DOCTOR OF SCIENCES

Modelling and model checking variability-intensive systems

Classen, Andreas

Award date:
2011

Awarding institution:
University of Namur

DOCTOR OF SCIENCE

Modelling and model checking variability-intensive systems

Andreas Classen

2011

Facultés universitaires Notre-Dame de la Paix

Thèse (Dissertation)

"Modelling and Model Checking Variability-Intensive Systems"

Classen, Andreas

Abstract

The premise of variability-intensive systems, specifically in software product line engineering, is the ability to produce a large family of different systems efficiently. However, in the case of safety critical systems, only a tiny fraction of the possible systems are generally offered to customers. One reason for this is that current quality assurance techniques used in the development of these systems, such as model checking, are designed for systems with no variability. In consequence, they are costly to apply to the whole system family. More specifically, when differences between systems are expressed in terms of features, the number of possible systems in a family is exponential in the number of features. Two major challenges for quality assurance techniques are thus scalable models and efficient algorithms. We address these challenges for a specific quality assurance technique: model checking. The proposed model checking approach is based on Featured Transition Systems (FTS), a novel formalism introduced in this thesis. FTS are a compact mathematical model for repr[...]

Référence bibliographique

Classen, Andreas. *Modelling and Model Checking Variability-Intensive Systems*. Prom. : Heymans, Patrick ; Schobbens, Pierre-Yves (2011)

Modelling and Model Checking Variability-Intensive Systems

Andreas Classen

Thèse présentée en vue de l'obtention
du titre de Docteur en Sciences.

PReCISE Research Centre
Faculty of Computer Science
University of Namur (FUNDP)
5000 Namur, Belgium

October 2011

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

Imprimé en Belgique
ISBN : 978-2-87037-731-4
Dépôt légal: D / 2011 / 1881 / 36

Jury

Prof. Naji Habra, University of Namur, Belgium
Prof. Patrick Heymans, University of Namur, Belgium (advisor)
Prof. Jean-Marie Jacquet, University of Namur, Belgium (chair)
Prof. Joost-Pieter Katoen, RWTH Aachen University, Germany
Prof. Bashar Nuseibeh, The Open University, UK and LERO, Ireland
Prof. Charles Pecheur, Catholic University of Louvain, Belgium
Prof. Pierre-Yves Schobbens, University of Namur, Belgium (co-advisor)

Abstract

The premise of variability-intensive systems, specifically in software product line engineering, is the ability to produce a large family of different systems efficiently. However, in the case of safety critical systems, only a tiny fraction of the possible systems are generally offered to customers. One reason for this is that current quality assurance techniques used in the development of these systems, such as model checking, are designed for systems with no variability. In consequence, they are costly to apply to the whole system family.

More specifically, when differences between systems are expressed in terms of *features*, the number of possible systems in a family is exponential in the number of features. Two major challenges for quality assurance techniques are thus scalable models and efficient algorithms. We address these challenges for a specific quality assurance technique: model checking.

The proposed model checking approach is based on *Featured Transition Systems* (FTS), a novel formalism introduced in this thesis. FTS are a compact mathematical model for representing the behaviours of a variability-intensive system. Basically, they are transition systems in which the presence of a transition depends on the features of the system. We define and study model checking algorithms that allow to verify FTS against temporal properties. They either prove that all systems of the family satisfy the property, or identify those that do not. Properties can be specified in *feature LTL* and *feature CTL*, extensions of the well known linear and branching time temporal logics.

In addition to the mathematical foundation, we discuss two implementations of FTS that can be used by non-experts. A first uses a symbolic representation of the state space and is implemented as part of the NuSMV model checker. The second, SNIP, uses a semi-symbolic on-the-fly algorithm. SNIP comes with an intuitive specification language based on Promela. Finally, we propose theoretical and empirical evaluations of our results. The baseline for our empirical evaluation is the application of classical model checking algorithms to each system of the family. Experiments conducted with both model checkers show that our algorithms can achieve order-of-magnitude speedups.

Résumé

L'ingénierie des lignes de produits logiciels est un paradigme d'ingénierie du logiciel dont le but est de permettre le développement efficace de grandes familles de logiciels. Cependant, dans le cas des systèmes critiques, peu de membres d'une famille de produits sont généralement vendus aux clients. En effet, les techniques d'assurance qualité utilisées dans le développement de ce type de systèmes, telles que le *model checking*, ne peuvent traiter que des systèmes sans variabilité. En conséquence, il est très coûteux d'appliquer ces techniques à tous les systèmes d'une même famille.

Si les différences entres les systèmes sont exprimées en terme de *features*, le nombre de systèmes possibles est exponentiel en fonction du nombre de features. Deux défis importants pour les techniques d'assurance qualité sont donc le développement de modèles et d'algorithmes qui passent à l'échelle. Nous traitons ces défis pour une technique spécifique: le model checking.

La technique de model checking proposée est basée sur les *Featured Transition Systems* (FTS), un nouveau formalisme introduit dans cette thèse. Les FTS sont un modèle mathématique pour représenter le comportement d'une famille de systèmes de façon compacte. Un FTS est essentiellement un système de transitions dans lequel la présence d'une transition dépend des features. Nous proposons et étudions des algorithmes permettant de vérifier des propriétés temporelles sur des FTS. Ces algorithmes peuvent prouver que tous les systèmes d'une famille satisfont une propriété donnée, ou bien ils identifient ceux qui ne le font pas. Pour spécifier les propriétés, nous proposons *feature LTL* and *feature CTL*, des extensions des deux logiques temporelles classiques.

En plus des fondements mathématiques, nous décrivons deux implémentations des FTS utilisables par des non-experts. La première utilise une représentation symbolique de l'espace d'états et a été implémentée au sein du model checker NuSMV. La deuxième, appelée SNIP, utilise un algorithme semi-symbolique à la volée. Le langage de modélisation de SNIP est basé sur le langage Promela. Enfin, nous discutons une évaluation théorique et empirique de nos résultats. Le cas de base utilisé pour l'évaluation empirique est l'application d'un algorithme de model checking classique à chaque produit. Les expériences conduites avec les deux outils montrent que nos algorithmes peuvent atteindre des gains en vitesse de plusieurs ordres de grandeur par rapport au cas de base.

Contents

Preface

> *Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are—by definition—not smart enough to debug it.*
>
> Brian Kernighan

Variability becomes more and more commonplace in today's software, be it in the form of configuration options or as an integral part of the development process, e.g., in *Software Product Lines* (SPLs) [Parnas, 1976, Clements and Northrop, 2001, Pohl et al., 2005]. Configuration options make it possible to adapt software to customers' needs and thereby to cover a wider market segment. In the case of *Software Product Line Engineering* (SPLE), multiple software systems (called *products*) are developed at once in order to benefit from economies of scale by systematically reusing common parts. Efficient management of the differences between the products of an SPL, i.e. its *variability*, is crucial. This type of system, in which variability plays an important role, is termed *variability-intensive*. Building variability-intensive systems has inherent advantages, such as productivity gains, shorter times to market and greater market coverage [Clements and Northrop, 2001, Pohl et al., 2005].

Unfortunately, the complexity created by variability and reuse also leads to problems [Leveson and Weiss, 2004]. Most software engineering techniques have to be adapted to cope with variability. In particular, techniques that analyse development artefacts, e.g., testing or static analysis, are only able to deal with one instance of the artefact (files with program code in the case of testing or static analysis) at a time. In the current state of SPLE, most analyses are thus carried out when building a product, i.e., during *application engineering* [Pohl et al., 2005]. Only few are conducted during *domain engineering*, i.e., when building the assets from which products are derived.

Many variability-intensive systems are safety critical. Embedded systems, for instance, are often developed as product lines [Ebert and Jones, 2009]. Verification, as a form of quality assurance, is important in this context. Often, due to quality considerations, only a fraction of the products that could be produced can be offered to customers. Existing verification techniques were mostly developed for single systems. Using those techniques to verify all prod-

ucts is very costly, since the number of possible products is exponential in the number of options or parameter values.

The verification process itself encompasses a wide range of activities. In the case of variability-intensive systems, there is a clear need to automate this task as much as possible due to the number of products that have to be considered. Furthermore, as critical systems tend to be distributed (e.g., electronic control units in modern cars), an important class of problems are related to the control flow, i.e., absence of deadlocks, race conditions and similar errors. Model checking [Clarke and Emerson, 1982, Queille and Sifakis, 1982] has proven to be a powerful technique for uncovering such errors. Model checking is one of the verification techniques that are currently restricted to the realm of single systems. It will be the focus of this thesis.

Problem statement

The model checking problem consists in deciding whether an abstract description of the system behaviour satisfies some specification. Specifications are usually expressed in a temporal logic. A model checking approach thus comprises three elements: a modelling language, a specification language and verification algorithms that check satisfaction. The two principal challenges to applying model checking to variability-intensive systems are the modelling language and the verification algorithms. These are most affected by variability.

Variability in SPLs is typically expressed in terms of *features*. Features are first-class abstractions that shape the reasoning of the engineers and other stakeholders [Classen et al., 2008a]. As such they can be used to express variability in any variability-intensive system. A set of features specifies a product of the variability-intensive system. The number of possible products is exponential in the number of features, i.e., $O(2^n)$, where n is the number of features. This is the source of much of the complexity of the problem at hand. The exact number of products depends on the constraints that exist between features, which are commonly recorded in a feature diagram [Kang et al., 1990].

A modelling language needs to be able to express $O(2^n)$ behaviours concisely. Current proposals are based on UML [Ziadi et al., 2003], modal transition systems [Fischbein et al., 2006, Fantechi and Gnesi, 2008, Asirelli et al., 2010a], modal I/O automata [Larsen et al., 2007, Lauenroth et al., 2009], deontic logics [Asirelli et al., 2009] and CCS [Gruler et al., 2008b]. With the exception of [Lauenroth et al., 2009], all these approaches fail to recognise the importance of features as a unit of difference. This means that they capture different behaviours, but offer little to no means to relate products and their behavioural descriptions. They also cannot make use of information contained in a feature diagram, such as the co-occurrence or mutual exclusion of two or more features. A first challenge is thus to propose an approach for scalable modelling of behaviour which overcomes the limitations of existing approaches.

The notion of satisfaction in variability-intensive systems can be reduced to satisfaction in single systems. Intuitively, if the behavioural model of a

variability-intensive system satisfies a property, this means that every product satisfies the property. The verification algorithm should thus be able to identify the products for which a property is satisfied and those by which it is violated. This means that is has to check property satisfaction for $O(2^n)$ products. None of the proposals mentioned above provides a model checking algorithm capable of doing this efficiently. The second challenge is thus the efficient verification of variability-intensive system behaviour.

Research questions

From this, we can derive the central research question of this thesis:

How can model checking be accomplished in the presence of variability?

This question touches upon many topics, and requires a comprehensive answer. We thus decompose and refine it into three smaller research questions.

Several general observations can be made from the preceding discussion. First, there is currently no understanding of what behaviour in an SPL context means. Modelling languages for behaviour either describe the behaviour of individual products rather than that of the product line; or they describe the behaviour of all products, without the ability to identify which behaviour pertains to which product; or they do not have a formal semantics which is required for model checking. In [Lauenroth et al., 2009], the authors describe a modelling language, but do not study its properties. A fundamental challenge, to be addressed first, is thus the following.

RQ1 *How can the behaviour of an SPL be described formally, and what does model checking of SPLs mean?*

The second step in answering the central research question is to provide and study solutions to the model checking problem identified as part of RQ1. To date, model checking of SPLs has only been treated in [Lauenroth et al., 2009] and [Asirelli et al., 2010a]. However, both approaches are limited. The first just proposes a rather inefficient proof-of-concept algorithm. The algorithm of the second cannot even be used to yield information about products. This leads us to formulate the second research question as follows.

RQ2 *Is SPL model checking tractable? If so, how?*

Both of these research questions can be answered solely with theory and mathematics. While theory and mathematics are the prerequisites of any practical development, it is our explicit goal to evaluate their viability in practice. The last step in answering the central research question is thus to implement the theory as part of a model checker. There are currently no tools available for SPL model checking. The third research question is the following.

RQ3 *How can SPL model checking be applied in practice?*

In this thesis, we shall strive to provide answers to these three questions.

Proposed solution

The basis of our proposed model checking approach are *Featured Transition Systems* (FTS), a novel formalism introduced in this thesis. FTS are a compact mathematical model for representing the behaviour of a large number of products. Basically, FTS are transition systems in which the presence of a transition depends on a combination of features. Features are thus modelled by *annotation*. This corresponds to the way features are commonly implemented in industry (i.e., with `#ifdefs` [Kästner et al., 2008, Liebig et al., 2010]). The behaviour of a particular product is a transition system, obtained by removing transitions whose feature annotation is incompatible with the product. In addition, an FTS is linked to a feature diagram, which specifies constraints between features. Linking transitions to features solves the traceability problem most current proposals suffer from. In case of a property violation, it allows us to provide a precise statement about the conflicting features. FTS thus overcome the limitations of existing modelling languages. Moreover, FTS have two desirable properties: an FTS can represent any finite set of transition systems (i.e., products) and FTS are exponentially more succinct than transition systems.

The model checking problem for FTS corresponds to the intuition given above. If an FTS satisfies a temporal property, the transition system of every product satisfies it. Model checking an SPL against a property thus corresponds to model checking all its products against this property. In consequence, properties for an SPL can be specified in existing temporal logics, *Linear Time Logic* (LTL) [Pnueli, 1977] and *Computation Tree Logic* (CTL) [Clarke and Emerson, 1982]. As a property might not be relevant to all products, we propose *feature LTL* (fLTL) and *feature CTL* (fCTL). These logics extend LTL and CTL with an operator that specifies the set of products for which a property should hold.

We define model checking algorithms that allow to verify FTS against fLTL and fCTL properties. Our model checking algorithms can verify all products at once and pinpoint those that violate the property. The algorithms explore the FTS instead of exploring the transition system of every product. They can thus prevent the exponential blowup due to the number of products. The algorithms compute for each state the set of products in which it is reachable. A central concern is to minimise the overhead caused by this, which we do by using a symbolic encoding for sets of products. A symbolic encoding is a compact data structure for a large set of elements. Symbolic encodings have long been used in model checking [McMillan, 1993, Burch et al., 1992]. A novelty of our algorithm is that it is semi-symbolic. It visits system states one by one, but represents products symbolically. In addition to this algorithm, we also give a fully symbolic algorithm. This is motivated by the fact that for single systems, symbolic algorithms have been shown to be applicable in cases where explicit algorithms do not scale [Burch et al., 1992].

As this overview of the theoretical results shows, our treatment of FTS touches many of the well known concepts and debates in model checking: LTL vs. CTL, explicit vs. symbolic. In addition to these central results, we cover

aspects such as expressiveness, parallel composition, deadlock checking and vacuity detection. We also provide an in-depth treatment of the model checking algorithms, studying their complexity, properties, and various optimisations.

Following RQ3, we not only cover the theoretical foundations, but also tools that can be used by non-experts. FTS are to variability-intensive systems what transition systems are to single systems: a semantic model for behaviour. They are not meant to be used as an actual modelling language. To be usable in practice, FTS have to be abstracted by a modelling language which is close to the problems to be modelled. We propose two such languages, based on different philosophies for modelling features. The first one, fSMV [Plath and Ryan, 2001], is a feature-oriented extension of the SMV language [McMillan, 1993]. The other is fPromela, an extension of the Promela language from the popular model checker SPIN [Holzmann, 2004]. fSMV is based on superimposition: features are specified modularly as changes to be done to a base system. A product is constructed by *composition* of features. fPromela, in contrast, follows an annotative approach in which statements can be guarded by features. There, products are constructed by *removing* statements related to non-selected features. The semantics of both languages is given in terms of FTS, and both are shown to be expressively equivalent to FTS.

Each language is implemented as part of a separate model checker, using different logics and algorithms. The model checker for fSMV is fNuSMV, an extension of the NuSMV model checker [Cimatti et al., 2000], using a fully symbolic algorithm for fCTL. The model checker for fPromela, SNIP, was implemented from scratch. It uses the semi-symbolic algorithm and supports the verification of fLTL properties. Both tools can be used to verify properties over all the products of a variability-intensive system at once. While the languages provide a practical solution to the problem of scaleable modelling, the model checkers provide practical solutions to the challenge of efficient verification.

For each tool, we conduct an evaluation consisting of a series of experiments measuring runtime and state space when the tool verifies a property. The baseline for this evaluation is the application of a classical model checking algorithm to each product. The experiments show that our algorithms can achieve up to order-of-magnitude speedups and reductions in the state space.

FTS are a semantic model for SPL behaviour which overcomes the limitations of existing semantic models. As such it serves as the formal foundation of our work. However, most of our results, such as the model checking problems and the algorithmic principles developed in this thesis can by applied beyond FTS. This becomes clear in our discussion of the model checking tools fNuSMV and SNIP, where the relation to FTS is only visible in the inner workings, almost imperceptible to the user. Nevertheless, their user interface clearly reflects the SPL model checking problems we defined.

Moreover, the contributions made in this thesis do not only apply to SPLs, or variability-intensive systems. Even in the case where a model does not represent a variability-intensive system, our algorithms can be used for tasks such as model understanding, or to determine which fragment of the model is respon-

sible for a property violation. Also, the idea of our semi-symbolic algorithm, which combines symbolic execution with explicit state-space exploration, can be applied beyond features and Boolean variables. In a broader sense, our algorithms calculate a Boolean function over model parameters which characterises the parameter values for which the model violates a property. Interpreted this way, it is clear that the parameters do not have to be features of a variability-intensive system. For example, they could represent design alternatives that are being considered for a given system.

Contributions

In summary, the principal contributions of this thesis are the following.

- FTS, a new semantic model for variability-intensive system behaviour. FTS are formally defined and extensively studied, including expressiveness, relation to existing languages and parallel composition. The improvements of FTS over existing work are manifold. Variability in FTS is a first-class citizen, meaning that there is a clear notion of which product has which behaviour. Reasoning on an FTS is equivalent to reasoning about the whole product line, or subsets of it. In FTS, very detailed behavioural variations (e.g., single transitions) can be expressed. The combination of transition system and feature diagram allows an FTS to take feature dependencies and incompatibilities into account.
- A study of the model checking problem in variability-intensive systems. We identify the relevant model checking decision problems, provide formal definitions and study their complexity. These results are presented in terms of FTS but apply to variability-intensive systems in general.
- New logics, fLTL and fCTL, as small but important variations of the well-known existing temporal logics.
- Algorithms for model checking FTS against fLTL and fCTL properties, the second major contribution. Our algorithms are the first to attempt to solve the model checking problem for variability-intensive systems efficiently. We propose two algorithms, a semi-symbolic algorithm, and a fully symbolic algorithm. We study the properties and optimisations of the semi-symbolic algorithm and propose two symbolic encodings for sets of products. We further show how the fully symbolic algorithm can be reduced to classical symbolic model checking of specially crafted transition systems. A detailed complexity analysis of the algorithms is given.
- We study high level modelling languages for variability-intensive system behaviour. fSMV can be used to express symbolic FTS and is based on existing work [Plath and Ryan, 2001]. fPromela is a new language, based on Promela. In both cases, we give a semantics in terms of FTS and prove full expressiveness of the high-level language. We further discuss the different philosophies used for modelling features in both languages.

- Two tools for model checking variability-intensive systems. We present a tool-chain built around an extended version of NuSMV. It can be used for the verification of fSMV models and implements the symbolic algorithm. We also present SNIP, a model checker for fPromela. Both tools are evaluated empirically based on a number of models.

Structure

The presentation is divided into four parts as follows.

Part I introduces the background of this research. Chapter 1 discusses software product lines, a method for the development of variability-intensive systems. Chapter 2 introduces the model checking problem, and relevant notations, concepts and techniques. Chapter 3 surveys the state of the art in verification of variability-intensive systems and identifies limitations.

Part II is the heart of the thesis. It covers the foundational and theoretical results. FTS are presented and studied in Chapter 4. In Chapter 5 we discuss logics and define the model checking problem for variability-intensive systems in general, and for FTS in particular. The algorithms for these decision problems are given in Chapter 6, with the semi-symbolic algorithms, and in Chapter 7 with fully symbolic algorithms.

Part III presents the practical results. We implemented our theory as part of two model checking tools, fNuSMV and SNIP presented in Chapters 8 and 9 respectively. Each tool comes with a high-level modelling language, of which we study the expressiveness and present example models. We further report on an empirical evaluation of our results.

Part IV discusses limitations and perspectives for future work in Chapter 10 and concludes the thesis.

Bibliographic notes follow on page 193 and an index on page 213.

Publications

This thesis is largely based on the following articles and technical reports, of which the candidate is the first author.

Published

[Classen et al., 2009a] *Towards Safer Composition*, in Proceedings of the 31st International Conference on Software Engineering (ICSE), Companion Volume, New Ideas and Emerging Results, IEEE, 2009, pages 227–230; with Patrick Heymans, Thein T. Tun and Bashar Nuseibeh. (Chapter 3)

[Classen et al., 2010b] *Model Checking Lots of Systems: Efficient Verification of Temporal Properties in Software Product Lines*, in Proceedings of the 32nd International Conference on Software Engineering (ICSE), ACM, 2010, pages 335–344; with Patrick Heymans, Pierre-Yves Schobbens, Axel Legay and Jean-François Raskin. (Chapters 4, 5 and 6)

[Classen et al., 2011c] *Symbolic Model Checking of Software Product Lines*, in Proceedings of the 33rd International Conference on Software Engineering (ICSE), ACM, 2011, pages 321–330; with Patrick Heymans, Pierre-Yves Schobbens and Axel Legay. (Chapters 5, 7 and 8)

Under review

[Classen et al., 2011d] *Modelling and Model Checking Variability-Intensive Systems with FTS*, submitted to IEEE Transactions on Software Engineering; with Patrick Heymans, Pierre-Yves Schobbens, Axel Legay and Jean-François Raskin. (Chapters 4, 5, 6 and 9)

Technical reports

[Classen, 2010c] *Modelling with FTS: a Collection of Illustrative Examples*, Technical report nb. P-CS-TR SPLMC-00000001, PReCISE Research Centre, University of Namur, 2010. (Chapter 4)

[Classen, 2010a] *CTL Model Checking for Software Product Lines in NuSMV*, Technical report nb. P-CS-TR SPLMC-00000002, PReCISE Research Centre, University of Namur, 2010. (Chapter 8)

[Classen et al., 2011b] *SNIP: An Efficient Model Checker for Software Product Lines*, Technical report nb. P-CS-TR SPLMC-00000003, PReCISE Research Centre, University of Namur, 2010; with Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. (Chapter 9)

Acknowledgements

Firstly, I would like to thank my advisor Prof. Patrick Heymans. I have been very fortunate to have an advisor who left me the freedom to define my own research agenda, who has always an open ear, and who works day and night to read drafts and provide feedback. I also thank my co-advisor Prof. Pierre-Yves Schobbens, whose rigour has helped improve this work tremendously and whose ability to predict theoretic results and validate theorem proofs is invaluable.

Many people contributed to the results presented in this thesis. In addition to my advisors, I have to thank Axel Legay (INRIA, Rennes), with whom I collaborated on most of the results presented herein; Prof. Jean-François Raskin (ULB, Brussels), who helped to bootstrap this work; Maxime Cordy (FUNDP, Namur), who implemented large parts of SNIP and helped with the benchmarks; and Nicolas Maquet (ULB, Brussels) and Marco Roveri (FBK, Trento, Italy) who helped me extend NuSMV. I also thank all the members of the jury whose comments were of great help. Finally, I thank Prof. Joost-Pieter Katoen for his hospitality during my stay at RWTH Aachen University, Germany.

The thesis was funded by the National Fund for Scientific Research (FNRS), through a research fellow grant. I benefited greatly from visits to international conferences, which would not have been possible without the funding received through FNRS travel grants, an ACM CAPS award and the MoVES project, funded by the IAP Programme of the Belgian Federal Science Policy Office.

Furthermore, I am grateful to my colleagues at the University of Namur: Arnaud Hubaux, with whom I collaborated on feature diagram configuration before I started working on model checking; Quentin Boucher and Raphaël Michel which whom I collaborated on TVL; Ebrahim Abbasi, Gilles Perrouin, Germain Saval, Thein Than Tun and Jean-Christophe Trigaux for their support and for many interesting discussions. I would also like to thank Kim Lauenroth (Paluno, Essen, Germany) and Andrzej Wąsowski (ITU, Copenhagen, Denmark) whom I had the pleasure of collaborating with on the writing of an EU FP7 research project proposal.

I thank my parents who made all of this possible; my family and my friends for their company and encouragement. Finally, I sincerely thank Sarah, who has to put up with my quirks and strange tastes every day. She has been a source of constant love and support during all these years. Thank you, Sarah.

Part I

Background

Chapter 1

Software Product Lines

> *The computer industry is the only industry that is more fashion driven than women's fashion.*
>
> Richard M. Stallman, in *The Guardian*, 2008

The discipline the present thesis should be attributed to is *software engineering*. Software engineering is the *"application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software."* [IEEE, 1990, Abran et al., 2004]. Within the vast area of computer science, it stands as a perfect example for the integration of two different paradigms [Wegner, 1976]. First and foremost, software engineering is *technocratic*, in that its purpose is the systematic and cost-effective design of software. It is also *mathematical* (the *rationalist* paradigm in [Eden, 2007]), in that it treats pieces of software as mathematical objects and uses abstraction or deductive reasoning to study these objects. Most disciplines in software engineering tend to focus primarily on one of these paradigms, with the technocratic paradigm dominating [Wegner, 1976]. This thesis is placed at their intersection. We explore theories and mathematical models with the ultimate goal to create tools that support the development of reliable software.

Software engineering as a discipline emerged in response to the *software crisis* of the 1960-1970s. The term itself first appeared in the 1968 NATO Software Engineering Conference [Naur and Randell, 1968]. As computers started to become orders of magnitude more powerful, the problems solvable and to be solved by computers started to become more complex, too [Dijkstra, 1972]. Also, the more powerful computers became, the more complex it became to program them [Dijkstra, 1972]. Furthermore, the development of systems with large teams of programmers created enormous organisational problems, which did not exist before [Brooks, 1975]. The results were budget overruns, canceled projects, low quality software and evan lethal accidents as in the case of the Therac-25 radiation therapy machine [Leveson and Turner, 1993].

Over the years, numerous techniques have been developed to address these problems, the most important of which were high-level programming languages. Whereas initial attempts were meant to solve the problem altogether, it became clear that there was no 'silver bullet' [Brooks, 1987]. Among the technologies initially lauded as silver bullets were Ada and other new programming languages, expert systems or program verification. These techniques provide incremental rather than order-of-magnitude improvements in productivity or quality. Most of them remain active subjects of research.

A key concept in software engineering, also introduced during the previously mentioned NATO conference, is that of reusable *software components* [McIlroy, 1968]. As described by McIlroy, software back then was *"produced by backward techniques"*, as many recurring problems were solved over and over again from scratch. The availability of standardised components solving these problems would allow for software development to be more industrialised. This idea is complemented by Parnas' introduction of *information hiding* [Parnas, 1971, Parnas, 1972], i.e., the idea that an interface should hide the design decisions that underly its implementation. A notable development of these ideas is the notion of *program family* [Dijkstra, 1969, Dijkstra, 1970]. According to Parnas, a set of programs is considered *"to constitute a family, whenever it is worthwhile to study programs from the set by first studying the common properties of the set and then determining the special properties of the individual family members"* [Parnas, 1976]. The motivation for this is that the development cost can be reduced when the *"designer/programmer pays conscious attention to the family rather than a sequence of individual programs"* [Parnas, 1976].

These developments culminated in what is today known as *Software Product Line Engineering* (SPLE), a discipline within software engineering which emphasises systematic and planned reuse. A *Software Product Line* (SPL) is traditionally defined as *"a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way"* [Clements and Northrop, 2001]. SPLs are a particular class of variability-intensive systems, those in which variability is systematically planned. As we shall see, the theories proposed herein apply to variability-intensive systems in general. We focus on SPLs because they provide a frame of reference, with well-defined concepts and notations.

In this chapter, we describe SPLE and its process in Section 1.1. We then introduce variability and feature diagrams in Section 1.2. In Section 1.3, we give an overview of feature-oriented software development. In Section 1.4, we discuss feature interactions, and the problem of feature interaction detection.

1.1 The software product lines paradigm

Whereas the idea of reuse goes back to early technical notions such as *software component*, SPLE as a paradigm did not appear until the advent of

widespread *Commercial Off-The-Shelf* (COTS) software in the 1990s. Compared to the mostly tailor-made software systems in use until then, COTS software is cheaper because it can be sold to more than one customer. A disadvantage that comes with this, of course, is that COTS software is not always well adapted to the requirements of the individual customers. There is a trade-off between price and fitness-for-purpose. As a mitigation, COTS software is made configurable. The configurable aspects range from run-time or compile-time options to extension mechanisms and scripting languages. Since then, configurability has become a characteristic shared by many software systems.

SPLE is an attempt at achieving a middle ground between tailor-made software and configurable COTS software. The goal is to achieve *mass customisation* [Pohl et al., 2005], i.e., *"the large-scale production of goods tailored to individual customers' needs"* [Davis, 1987]. In SPLE, this is achieved by developing several similar systems (called "products") at once. These products are designed as a family from the outset, and their development relies heavily on *reuse*. Reuse in SPLE is *planned and systematic* rather than opportunistic. The different products are identified upfront and a model of their variability and commonality is created. Commonality denotes all aspects the products have in common, whereas variability denotes those in which they differ.

Throughout most the thesis, we use a product line of beverage vending machines (inspired from [Fantechi and Gnesi, 2008]) as the running example. An example for commonality in these machines is that they all have an availability indicator for each beverage. There are several sources of variability. The vending machines serve two kinds of drink: soda and tea, and accept two kinds of currency. There are also vending machines that distribute drinks for free (in an office, for instance). All machines except for those that distribute free drinks have a protected beverage compartment. Finally, some machines allow a purchase to be cancelled, and others do not. When these variations are combined, a large number of potential vending machines is obtained. Since each of these requires a different controller, it seems natural to develop these controllers as an SPL (e.g., as a parameterised controller), and not each time anew.

Commonality and variability are expressed in terms of *features*. Features intuitively characterise pieces of functionality in a software system. Many definitions exist for the term 'feature' [Classen et al., 2008a]. In this thesis, we stick to the one of [Batory et al., 2006], who defines a feature as *"an increment in functionality"*. This means that a feature can range from an easily perceivable key characteristic of the system to a very subtle change in functionality. Moreover, features can denote technical characteristics of a system (like components, or classes), as well as management or customer-oriented characteristics [Metzger et al., 2007]. Since commonality and variability are expressed in terms of features, a product can be defined as a set of features [Schobbens et al., 2007] (those that are part of the product). In general, not all feature combinations are considered valid products. For example, some features might be incompatible. To capture the set of valid products, feature diagrams can be be used [Kang et al., 1990, Czarnecki and Eisenecker, 2000, Schobbens et al.,

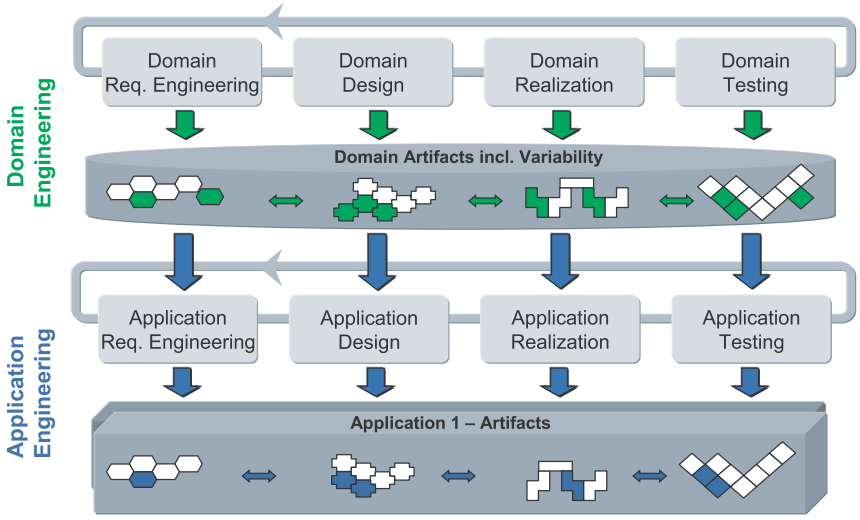


Figure 1.1: Schema of the SPLE process [Pohl and Metzger, 2006].

2007]. A feature diagram formally specifies the set of valid products by means of constraints between features. Feature diagrams will be presented in more detail in Section 1.2.

The key to SPLE is systematic reuse, which occurs throughout all phases of the software life cycle. More precisely, SPLE distinguishes between two development processes: *domain engineering* and *application engineering* [Pohl et al., 2005]. Both processes can be thought of as full-fledged software development processes, each with a different goal. The overall process is illustrated in Figure 1.1, taken from [Pohl and Metzger, 2006]. Domain engineering is concerned with producing reusable artefacts. These artefacts can range from requirements, design documents and models to code. The domain artefacts are created to be reused. They contain variability and can be parameterised or composed to create products. Generally, features correspond to such artefacts or their parameters. In this respect, domain artefacts are different from those found in the classical development processes of single systems. A further particularity is that domain engineering generally does not produce a running system. A concrete system is created during the application engineering process. This process is repeated for each product. Its starting point is thus the selection of features that make up a product. The processes of selecting features is called *configuration* [Czarnecki et al., 2005] and might reach into, or even beyond the application engineering phase (depending on the *binding time* of the various choices). During application engineering, a particular product is built using the domain artefacts. Depending on the techniques used to implement the domain artefacts, this can take different forms. For example, if

domain engineering has produced a large body of configurable code, then application engineering amounts to reducing this code to the selected features. On the other hand, if domain engineering just produced a library of reusable functions, then application engineering might be more involved, requiring new code to be written.

The motivation for adopting SPLE is that systematic reuse of development artefacts leads to economies of scale [Clements and Northrop, 2001]. Basically, domain artefacts can be developed in mass production mode, with all the benefits and economies this implies for the producer. Systems adapted to the requirements of a customer are derived from these artefacts, allowing the producer to benefit from the economic advantages of offering individualised products to the customer.

1.2 Feature diagrams

Feature Diagrams (FDs) are a common means to model the variability of an SPL. In this context, they have proven to be useful for a variety of tasks such as project scoping, requirements engineering and configuration [Bosch, 2005, Lee et al., 2002, Thiel and Hein, 2002, Pohl et al., 2005, Benavides et al., 2005]. FDs were introduced in the 1990s by [Kang et al., 1990]. They rose to prominence later through the work of [Czarnecki and Eisenecker, 2000, Batory, 2005, Batory et al., 2006, Schobbens et al., 2006, Benavides et al., 2010].

1.2.1 Syntax

To illustrate the syntax of FDs, we elaborate on the vending machine example.

From the description of the product line of beverage vending machines given above, we can derive the following features. *Soda* and *Tea* represent the possible beverages served. To capture the constraint that at least one beverage has to be sold, they are grouped under a feature *Beverage* with an *or*-decomposition. Features *Euro* and *Dollar* represent the currencies. They are also grouped under a corresponding feature, with an *xor*-decomposition, meaning that they are alternatives. The *FreeDrinks* and *CancelPurchase* features are independently optional. Finally, FDs require a root feature which represents the system itself, hence *VendingMachine* in our case. The resulting FD is shown in Figure 1.2. For convenience, each feature is given a one-letter acronym. Some of the features are coloured. These colours will be used in a later chapter.

Basically, FDs are trees (or directed acyclic graphs, DAGs [Kang et al., 1998]) whose nodes denote features and whose edges represent top-down hierarchical decomposition of features. Each decomposition tells that, given the presence of the parent feature in a product, some combination of its children should be present in the product, too. Which combinations are allowed depends on the type of the decomposition. In addition to the tree-shaped decomposition structure, FDs can also contain cross-cutting constraints (usually *requires*,

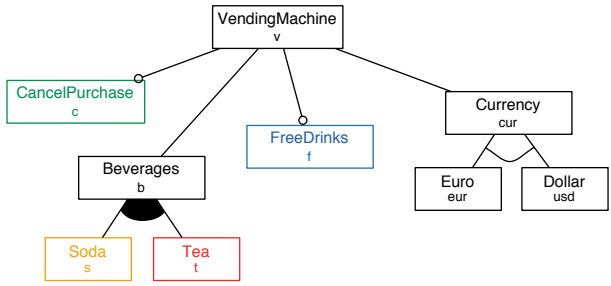


Figure 1.2: Feature diagram of an SPL of beverage vending machines.

Table 1.1: Graphical elements of FDs and their meaning

Concrete syntax	Boolen operator	Cardinality	Meaning
	*and* : $\wedge$	$\langle n..n \rangle$	mandatory children
	*or* : $\vee$	$\langle 1..n \rangle$	one or more children required
	*xor* : $\oplus$	$\langle 1..1 \rangle$	alternative choice of children
		$\langle i..j \rangle$	custom cardinality
			optional child feature (under *and*)

expressing dependency, or *excludes*, expressing incompatibility) as well as side constraints in a textual language such as propositional logic [Batory, 2005]. FDs are thus a constraint language that can be used to define the set of valid products of an SPL concisely.

Let N be the set of all features of an SPL. A specific set of features $p \subseteq N$ specifies a *product*. An FD d formally defines a set of products, i.e., a set of sets of features $[\![d]\!]_{FD} \subseteq \mathcal{P}(N)$. The formal definition of FDs is due to [Schobbens et al., 2006] and follows the guidelines of Harel and Rumpe [Harel and Rumpe, 2000]. They argue that each modelling language must possess an unambiguous mathematical definition of three distinct elements: the *syntactic domain*, the *semantic domain* and the *semantic function*, traditionally written $[\![\ ]\!]$.

Formally, an FD is defined as follows.

Definition 1.1. *An FD d is a tuple $(N, r, DE, \lambda, \Phi)$, where*

- *N is the set of features (nodes).*
- *$r \in N$ is the root.*

- *$DE \subseteq N \times N$ is the decomposition relation between features which must form a tree or a DAG with r as its root. For convenience, we write $children(f) \triangleq \{f' \mid (f, f') \in DE\}$.*
- *$\lambda : N \to \mathbb{N} \times \mathbb{N}$ indicates the decomposition type of a feature, represented as a cardinality $\langle i, j \rangle$ where i indicates the minimum number of children required in a product and j the maximum.*
- *Φ is a formula that captures additional constraints such as* requires *and* excludes*. Without loss of generality, we consider Φ to be a formula in propositional logic, $\Phi \in \mathbb{B}(N)$, as defined in Definition 1.2 below.*

The syntactic domain of the FD language is the set of all FDs conforming to this definition. The semantic domain is $\mathcal{PP}(N)$, meaning that each correct diagram is interpreted as a set of products*. The semantic function, $[\![d]\!]_{FD}$, returns the* valid products *of d, that is, any $p \subseteq N$ that*

- *contains the root: $r \in p$;*
- *satisfies the decomposition types: $m \leq |children(f) \cap p| \leq n$, for any $f \in p$ with $\lambda(f) = \langle m..n \rangle$;*
- *satisfies the additional constraints: $p \models \Phi$;*
- *contains the parents of selected features: $g \in p \wedge g \in children(f) \Rightarrow f \in p$.*

An optional feature f with parent g is syntactic sugar for two features f and f' with $(g, f'), (f', f) \in DE$ and $\lambda(f) = \langle 0, 1 \rangle$.

For example, the FD in Figure 1.2 admits 24 valid products out of the 2^9 possible feature combinations (or 2^6 if the structuring features are omitted).

Given an FD, *configuration* is the process of gradually making the choices defined in the FD with the purpose of determining the product that is going to be built. In a realistic development, the configuration process is a small project itself, involving many people and taking up to several months [Rabiser et al., 2007]. Czarnecki *et al.* [Czarnecki et al., 2005] proposed *multi-level staged configuration*, in which configuration is carried out by different stakeholders at different levels of product development or customisation. At each stage some variability is removed from the FD until none is left. This addresses problems that occur when different abstraction levels are present in the same FD and also allows for more realism since a realistic project would have several related FDs rather than a single big one [Reiser and Weber, 2006, Rabiser et al., 2007]. In [Classen et al., 2009c], we formalised and studied the properties of configuration processes [Classen et al., 2009c] which led to the proposal of *feature configuration workflows* [Hubaux et al., 2009, Classen et al., 2009b].

The notion of configuration is closely related to the *binding time* of a feature, which is the point in the development cycle at which a feature selection is fixed. One generally distinguishes between design-time variability, i.e., variability that is bound at design-time, and run-time variability i.e., variability that is only bound at run-time (using configuration files, or other parameters). The scope

of this thesis will be limited to design-time variability. The model checking technique proposed in this thesis does not directly apply to cases in which features are activated or deactivated at runtime.

The FDs conforming to Definition 1.1 are rather basic but largely sufficient for the purpose of this thesis. Various extensions to FDs have been proposed, such as feature attributes [Benavides et al., 2005], binding times [van Gurp et al., 2001], the use of of multi-level feature trees to improve scalability [Reiser and Weber, 2006, Reiser et al., 2007]. An alternative to the graphical FD notation shown in Figure 1.2 are textual languages such as FDL [van Deursen and Klint, 2002], GUIDSL [Batory, 2005], VSL [Abele et al., 2010, Reiser, 2009] or our own TVL [Classen et al., 2011a, Classen et al., 2010a, Boucher et al., 2010a]. In TVL, the FD of the vending machine from Figure 1.2 would be written as follows.

```
1  root VendingMachine
2    group allOf {
3      opt CancelPurchase,
4      Beverages group someOf {
5        Soda,
6        Tea
7      },
8      opt FreeDrinks,
9      Currency group oneOf {
10       Euro,
11       Dollar
12     }
13   }
```

The keyword `allOf` denotes an *and*-decomposition, `someOf` an *or*-decomposition and `oneOf` an *xor*-decomposition. Optional features are preceded by the `opt` keyword, and the hierarchy is represented by nested `groups`. TVL is used by the tools developed as part of this thesis. In the chapters of Part III, we will generally use TVL instead of the graphical representation.

1.2.2 Analysis and reasoning

FDs lend themselves well to automated analysis. Examples of such analyses are *satisfiability*, i.e., whether the FD admits at least one valid product; *product inclusion*, i.e. whether the FD admits a particular product, or a product containing certain features, and *product listing*, i.e. to list the valid products. The straightforward semantics of FDs can easily be encoded in the form of an expression in propositional logic [Mannion, 2002, Batory, 2005].

Definition 1.2. *Given a set of variables vx, $\mathbb{B}(vx)$ denotes the set of all possible expressions ϕ, with $\phi ::= 1 \mid v\ (\in vx) \mid \phi_1 \wedge \phi_2 \mid \neg\phi$. The usual operators*

are derived from this: $0 \triangleq \neg 1$, $\phi_1 \vee \phi_2 \triangleq \neg(\neg\phi_1 \wedge \neg\phi_2)$. *In the remainder of this thesis, we write* 1 *to denote true and* 0 *to denote false.*

The semantics of an expression $\phi \in \mathbb{B}(vx)$*, written* $[\![\phi]\!]$*, is the set of functions* $\sigma : vx \to \{0,1\}$ *so that* $\sigma \models \phi$*, where:* $\sigma \models 1$, $\sigma \models v \iff \sigma(v) = 1$, $\sigma \models \phi_1 \wedge \phi_2 \iff \sigma \models \phi_1 \wedge \sigma \models \phi_2$ *and* $\sigma \models \neg\phi \iff \sigma \not\models \phi$. *Regarding notation:* $\sigma \in [\![\phi]\!]$, $\sigma \models \phi$ *and* $\phi(\sigma) = 1$ *(function application) are equivalent.*

In a slight abuse of the language we just specified, we note $\mathbb{B}(d)$ the propositional logic encoding of an FD d. When N is the set of features in d, then $\mathbb{B}(d) \in \mathbb{B}(N)$, that is, the variables in the expression correspond to the features. An assignment σ thus defines a product (1 corresponds to selection and 0 to deselection of a feature), and $[\![\mathbb{B}(d)]\!]$ defines a set of products, namely $[\![d]\!]_{FD}$.

With this encoding, satisfiability of an FD d corresponds to satisfiability of $\mathbb{B}(d)$. The inclusion of a product p corresponds to satisfiability of $\phi_p \wedge \mathbb{B}(d)$, where ϕ_p is a conjunction of all literals positive for features in p and negative otherwise. Such analyses can be conducted using a satisfiability (SAT) solver and easily scale up to FDs with 10,000 of features [Mendonca et al., 2009]. An alternative representation for Boolean functions are binary decision diagrams [Bryant, 1992], which can also be used to conduct these analyses.

For the remainder of the thesis, unless otherwise stated, we always assume d to denote an FD, and N a set of features.

1.3 Feature-oriented software development

The SPL implementation techniques that use features (or similar concepts [Pohl et al., 2005]) as the building blocks of a product fall under the umbrella of *Feature-Oriented Software Development* (FOSD) [Prehofer, 1997, Batory, 2004, Apel and Kästner, 2009, Apel et al., 2011]. FOSD techniques range from model-driven approaches to pure implementation techniques and can be categorised broadly into two groups: annotative and compositional [Kästner et al., 2008].

Annotative techniques consist in implementing a product containing all features, which are clearly identified by annotations (e.g. using `#ifdef`s in the C programming language [Kästner et al., 2008, Liebig et al., 2010], or code tags [Boucher et al., 2010b]). Creating a product consists in removing code or artefacts pertaining to non-selected features (which we call *pruning*). This is probably the FOSD technique most commonly used in industry. A claimed disadvantage of annotative techniques is that they do not guarantee modular feature implementations, i.e., implementations in which features are independent from each other and interact in well-defined ways. Most annotative techniques allow to intertwine features arbitrarily in the code, which breaks modularity.

Compositional techniques, in contrast, separate the individual features and only allow for modular implementations. Features are specified individually as building blocks of a system and a *base system* provides a common core to which features can be added. A product is obtained by composing the base system

with the implementations of selected features. A number of composition mechanisms have been proposed in the literature. Programming language based approaches to FOSD generally rely on *superimposition* [Francez and Forman, 1990]. Examples of such techniques are aspect-oriented programming [Kiczales et al., 1997, Mezini and Ostermann, 2004, Voelter and Groher, 2007], mixins [Smaragdakis and Batory, 2002] and FeatureC++ [Apel et al., 2005], which are supported by tools such as AspectJ [Laddad, 2003], AHEAD [Batory, 2004], or FeatureHouse [Apel et al., 2009]. Approaches based on composition are most commonly used in research.

1.4 Feature interactions

In practice, features are seldom independent. FOSD techniques generally allow features to share variables, or to override changes made by other features. In these cases, features are said to *interact*, and a particular case is called a *feature interaction* [Calder et al., 2003]. An interaction might be desired (e.g., two features collaborating) or undesired (e.g., incompatible features). Undesired feature interactions are sometimes called *interferences*. When we use the term *feature interaction*, unless otherwise stated, we mean an undesired interaction.

Feature interactions were originally a research topic in the telecommunications domain [Keck and Kuehn, 1998]. The various services offered in a telephone network are all centralised (hence the concept of *intelligent network*). With a large number of centralised services, the likelihood of some of them interacting increases considerably. Furthermore, the exponential number of possible service combinations makes it hard to identify interactions by testing.

With the advent of SPLE and FOSD, feature interactions have once more become an important research problem. A large number of approaches for detecting and managing interactions, largely in telecommunication systems, have been proposed [Calder et al., 2003]. Among these approaches are approaches for detection (often based on verification techniques [Plath and Ryan, 2001, Calder and Miller, 2001]) and approaches for avoidance (based on architecture [Jackson and Zave, 1998], or particular composition operators [Hay and Atlee, 2000]). Detection approaches are generally limited to testing a subset of the potential systems, by considering pairs of features only. This is a major difference wrt. the work presented in this thesis.

In the past [Classen, 2007, Classen et al., 2008a, Classen et al., 2008c], we have studied the formal properties of feature interactions in the requirements engineering framework of [Zave and Jackson, 1997]. Our focus there were feature interactions that take the physical environment into account, which are especially relevant for SPLs of embedded control systems [Metzger et al., 2005].

Chapter 2

Model Checking

> *It is all a matter of time scale. An event that would be unthinkable in a hundred years may be inevitable in a hundred million.*
>
> Carl Sagan, *Cosmos*, 1998

Information and communication technology permeates our daily lives. As software systems become increasingly complex and play more important roles, reliability has become a prime concern in their development. Embedded systems, in particular, are pervasive and often control safety-critical processes. Failures in systems such as automobile controllers [Koscher et al., 2010], network protocols [Zave, 2008], or spacecraft [Havelund et al., 2001] can cause death, injury or major financial loss.

An important activity in the development of such systems is thus quality assurance. Quality assurance comprises two aspects: validation and verification [Easterbrook, 1996]. Validation considers the system in its context. Its goal is to make sure that the system being built conforms to initial requirements, i.e., that the *right system is built*. Verification, in contrast, makes assumptions about the environment and tests whether the system exhibits certain properties under these assumptions, i.e., whether the *system is built right* [Boehm, 1981].

In this thesis, we focus on verification, and one method in particular: model checking. Section 2.1 introduces model checking while Section 2.2 covers its fundamentals. In Section 2.3, we discuss model checking algorithms and in Section 2.4 the advanced techniques relevant to this work.

2.1 Introduction to model checking

Model checking is part of the subdomain of software engineering concerned with verification. A number of verification techniques were born out of the 1960s-70s software crisis. Many of them, such as Hoare logic [Hoare, 1969], consisted of proof systems for sequential programs. The goal of these methods was to

start from an assumption about the inputs of a program (the precondition) and systematically derive the guarantees it makes about the output (postcondition), or the other way round. Hoare logic has generic rules for each statement and derives the postcondition statement by statement. Loop statements are the most difficult, as they require a *loop invariant*, that is, a property that holds at the beginning of each iteration.

The early methods did not fare well as programs grew more complex. Since they were performed manually, they did not scale beyond simple programs, and the guarantees they could make were offset by the possibility of errors in their application. Furthermore, they had theoretical limitations. The search of invariants is hard to automate, and some properties, such as termination, are undecidable. Another problem came with the advent of concurrent execution of programs and reactive systems. Existing methods all worked under the assumption that a program was a sequence of steps which calculated a function of the input. They did not consider the case that execution might be interleaved in various ways with that of other programs, and that the programs might be interacting. They also did not consider the case of systems that reacted to inputs and produced outputs continuously, as a normal controller does.

This led to developments such as process algebra [Milner, 1980], which allows to describe distributed systems, and model checking, which allows to verify them, in the early 1980s. Model checking was developed independently by Clarke and Emerson [Clarke and Emerson, 1982] and Queille and Sifakis [Queille and Sifakis, 1982]. Contrary to other verification techniques at the time (e.g., testing, code review, or theorem proving), model checking is fully automatic and does not require human intervention. Model checking targets *reactive systems*; that is, systems with little to no data manipulation, but with a lot of non-determinism and interaction with the environment. Those properties make reactive systems extremely hard to test. Many behaviours are just exhibited under very special circumstances. While they are highly unlikely to be reproduced during a test session, they are rather likely to occur during years of operation. Reactive systems are also rather easy to model check. As they only manipulate small amounts of data, their state space is often finite and of reasonable size. Reactive systems are pervasive today. Most embedded systems, ranging from controllers in automobiles to pace makers, are of this kind. They observe a set of phenomena in the real world, and react accordingly.

The role of verification in software engineering has changed over time. The initial goal of most approaches was to prove some sort of correctness. This turned out to be overly ambitious, not least because there is no generic notion of correctness. Furthermore, for complex systems it is considered impossible, or at least impracticable, to provide a full verifiable specification of the system. Nowadays, model checking and other verification techniques are used in concert. Their goal is to provide confidence in designs and assistance in the development.

Model checking is highly relevant today. It is used in the development of critical systems [Giannakopoulou et al., 2005, Havelund et al., 2001], commercial software [Ball and Rajamani, 2001, Ball et al., 2004], and in the development

of hardware circuits [Burch et al., 1990]. In 2007, Clarke, Emerson and Sifakis received the Turing Award for their work on model checking.

2.2 Fundamentals

Model checking essentially deals with two types of artefact: *models* which describe system behaviour, and *properties* which are specifications that one would like the system to satisfy. Given a model and a property, the model checker can determine whether or not the model *satisfies* the property.

Unfortunately, the terms 'model' and 'specification' are overloaded with interpretations, some overlapping. In this thesis, 'model' denotes an artificial abstraction of the real world created for the purpose of analysis. A 'specification' is a declarative assertion that expresses a desired property of an artefact. In this sense, a model might well serve as the specification for an engineer. However, when we write 'specification', we generally mean assertions about the system and the model. Moreover, in the study of formal languages (such as propositional logic), the term 'model' denotes an interpretation of a sentence in a language that evaluates to *true*. This is the meaning of 'model' in 'model checking', that is, model checking is appropriately named as the procedure of checking whether an interpretation (the behavioural model) is a model of a formula (the specification).

Let us use the vending machine to illustrate these concepts. The behaviour of a basic vending machine which just sells soda and tea is as follows. Initially, the machine waits for a coin to be inserted. Once a coin is inserted, it returns change (assuming all beverages have the same price) and waits for the user to select the beverage. Upon selecting soda or tea, the machine serves it and opens the beverage compartment. The customer can then take her beverage and the machine closes the beverage compartment.

An example of a property that a vending machines should satisfy is that *"After selecting a beverage, the machine will always open the beverage compartment to allow the customer to collect her purchase."*

2.2.1 Formal models

The starting point for model checking is a description of the behaviour of a system. Such a description generally abstracts away from the physical characteristics of the machine on which the system is executed. In the most basic form, the behaviour of a system can be considered as a sequence of *states*. For a system consisting of a single program, a state would be a snapshot of the block of memory allocated to the program. The set of all states of a system is its *state space*. If a state is a block of n bits of memory, the state space is given by the 2^n possible values that the block of memory can have. The program consists of a set of instructions that manipulate this memory. Its behaviour is thus given by a sequence of blocks of memory, one for each executed instruction.

As we shall see later, model checking algorithms make searches in the state space of a system. A limiting factor in the performance of these algorithms is the number of states that can be explored. As seen before, the size of the state space of a system is exponential in the size of a state. Already for states as small as 288 bits (or 36 bytes), the maximal size of the state space is about 10^{73} terabytes, or as large as the observable universe measured in numbers of atoms (which is about 10^{85} [Hawking, 1988]). Even if only a fraction of these states is ever taken by the system, the number of states would be huge. This problem is known as *state explosion*, and is one of the main obstacles for the adoption of model checking to verify program code.

One way to combat state explosion is abstraction. In practice, languages for describing system behaviour abstract away from execution details, such as the representation of programs and their variables in memory. In addition, the engineer creating the model will abstract away from details that he considers irrelevant to the properties he wants to analyse. E.g., the description of the vending machine behaviour given above does not mention prices, or describe the handling (identification, validation and counting) of the inserted coins.

Whatever the language for describing behaviour, it always boils down to sequences of states. The most basic model for the system behaviour, and the semantic basis for many behavioural modelling languages are *transition systems* and *Kripke structures* [Baier and Katoen, 2008, Kripke, 1963]. Basically, a transition system is a graph where vertices are states and edges transitions between states. A transition from one state to another represents the capability of the system to perform this state change. A number of states are designated initial states, which means that they represent the possible states of the system at launch. Every path through the graph corresponds to a possible execution of the system, and the semantics of the transition system is the set of such executions. Formally:

Definition 2.1. *A transition system is a tuple* $ts = (S, trans, I)$*, where*

- S *is a set of states;*
- $trans \subseteq S \times S$ *is a set of transitions;*
- $I \subseteq S$ *is a set of initial states.*

For $s \in S$*,* $paths(ts, s)$ *denotes the set of all non-empty (potentially infinite) sequences* $\pi = s_0 s_1 \ldots$ *with* $s_0 = s$ *and* $s_i \rightarrow s_{i+1}$ *for all* $0 \leq i$*.*

Definition 2.2. *The semantics of a transition system ts is its set of* executions *(also called behaviours),* $[\![ts]\!]_{TS} = \bigcup_{s_0 \in I} paths(ts, s_0)$*. The semantic domain of transition systems is thus the power set of the (infinite) set of all possible (finite and infinite) executions.*

Unlike automata, transition systems have no accepting states. While they may have finite executions, the most interesting cases are those where the executions are always infinite. This corresponds to systems that never terminate, which is often the case for reactive systems.

Transition systems serve as the model on which most of the model checking theory is based, as they are independent of the types or variables used in actual modelling languages. However, verification properties usually *do* refer to variables, or particular states. To this end, transition systems are generally augmented with one or two forms of labelling. Firstly, transitions have textual labels that convey the action that causes the state change. In our case, those labels serve mainly as documentation and make example models easier to understand. Secondly, states are labeled with atomic propositions. Atomic propositions are assertions about the system state, e.g., *'beverage compartment open'*, that are true in the states that are labelled with them. Atomic propositions are used to abstract away from variables and other high-level concepts generally used when specifying properties. This leads to the following definition, which is the proper definition used in the remainder of the thesis.

Definition 2.3. *A transition system is a tuple* $(S, Act, trans, I, AP, L)$*, where*

- S *is a set of states;*
- Act *is a set of actions;*
- $trans \subseteq S \times Act \times S$ *is a set of transitions, with* $(s_1, \alpha, s_2) \in trans$ *sometimes noted* $s_1 \xrightarrow{\alpha} s_2$*;*
- $I \subseteq S$ *is a set of initial states;*
- AP *is a set of atomic propositions;*
- $L : S \to 2^{AP}$ *is a labelling function.*

The semantics is defined as in Definition 2.2.

A Kripke structure is a transition system without actions, and in which every state has at least one outgoing transition (i.e., all executions are infinite).

For example, a transition system of the beverage vending machine is shown in Figure 2.1. Each step of the procedure described at the beginning of Section 2.2 has become a transition. To avoid clutter, atomic propositions are omitted in the figures.

Systems consisting of several interacting processes can be modelled with several transition systems. The transition system of the combined system is then defined as their *parallel composition*. The initial assumption is that both processes are independent. The state space after parallel composition is thus

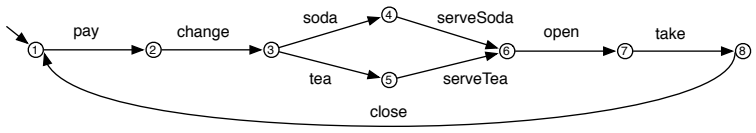


Figure 2.1: Transition system of the basic vending machine.

the product of the original state spaces (another reason for state explosion), so that a process can be in any state, independently from the other. The parallel composition of two transition systems is the interleaving of their executions. This means that only one process at a time can fire a transition, after which his part of the global system state changes. In the real world, processes may synchronise, e.g., when sending/receiving a message. Parallel composition takes this into account by forcing transitions with the same label to execute synchronously. In this case, both processes fire their transition at the same time, and the global system state changes for both. This is formalised as follows.

Definition 2.4. *Given two TS* $ts_i = (S_i, Act_i, trans_i, I_i, AP_i, L_i)$*, with* $i \in \{1, 2\}$*, the parallel composition of* ts_1 *and* ts_2 *synchronised over the set of shared actions* $Act_1 \cap Act_2$*, written* $ts_1 || ts_2$*, is the TS*

$$(S_1 \times S_2, Act_1 \cup Act_2, trans, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where

- $L((s_1, s_2)) = L(s_1) \cup L(s_2)$
- *trans is the smallest relation satisfying*
  - *for* $\alpha \notin Act_1 \cap Act_2$ *(interleaving):* $\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2)}$ $\frac{s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1,s_2')}$
  - *for* $\alpha \in Act_1 \cap Act_2$ *(synchronisation):* $\frac{s_1 \xrightarrow{\alpha}_1 s_1' \wedge s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2')}$

This form of parallel composition is just one of many. However, it is widely used and captures the key concept of parallel composition: asynchronous execution.

Transition systems are very basic, and are generally not meant to be used as a modelling language directly. There exist a number of high-level languages in which the engineer can use familiar concepts such as variables to specify behaviour. Such languages can often be generalised to *program graphs* [Baier and Katoen, 2008], whose semantics is expressed in terms of transition systems.

2.2.2 Properties

The second key concept in model checking are specification languages for the properties that the models are checked against. Since transition systems as defined in the previous sections abstract away from time, properties can only refer to the inclusion and the ordering of certain states in executions. Such properties can be expressed in temporal logics [Prior, 1967], which are propositional languages that include modalities for time.

The first use of temporal logic in computer science was due to [Pnueli, 1977] who proposed *linear temporal logic* (LTL) to encode properties of executions (see Definition 2.2). An execution satisfies a property if its initial state satisfies it. In addition to the usual Boolean connectives, LTL has temporal operators. Those are *next*, $\bigcirc\phi$, which requires that the next state satisfies ϕ; and *until*, $\phi_1 U \phi_2$, which requires that ϕ_2 is satisfied in some future state and that ϕ_1 holds until then.

Definition 2.5. *An LTL formula ϕ is an expression*

$$\phi ::= 1 \mid a \ (\in AP) \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid \bigcirc \phi \mid \phi_1 U \phi_2.$$

LTL formulas are interpreted over infinite executions. Satisfaction for an execution π is defined as follows:

$$\begin{array}{ll}
\pi \models 1 & \\
\pi \models a & \iff a \in L(head(\pi)) \\
\pi \models \neg\phi & \iff \pi \not\models \phi \\
\pi \models \phi_1 \wedge \phi_2 & \iff \pi \models \phi_1 \text{ and } \pi \models \phi_2 \\
\pi \models \bigcirc\phi & \iff \pi_1 \models \phi \\
\pi \models \phi_1 U \phi_2 & \iff \exists i \geq 0 \bullet \pi_i \models \phi_2 \wedge \forall j \in [0, i-1] \bullet \pi_j \models \phi_1
\end{array}$$

Where $head(\pi)$ denotes the first state in π and π_i denotes the tail of π starting at the ith state (with $\pi_0 = \pi$). Satisfaction for a transition system ts is given by $ts \models_{LTL} \phi \iff \forall \pi \in [\![ts]\!]_{TS} \bullet \pi \models \phi$.

Other Boolean connectives can be derived from $\wedge$ and $\neg$. Other temporal operators can be derived from U: *eventually*, $\Diamond\phi \triangleq 1U\phi$, requires that the ϕ holds in some future state; and *globally* $\Box\phi \triangleq \neg\Diamond\neg\phi$, which requires that ϕ holds on all future states.

In the case of the vending machine, suppose we have two propositions *selected* (denoting the fact that the user has selected a beverage) and *open* (denoting the fact that the beverage compartment is open). The example property given before could then be translated to the following LTL formula: $\Box(selected \Rightarrow \Diamond open)$. An execution that satisfies the property is one in which the following pattern of propositions is always repeated

$$\{\} \rightarrow \{selected\} \rightarrow \{open\}$$

A violating execution is an infinite execution in which no *open* appears after the last *selected*

$$\{\} \rightarrow \{selected\} \rightarrow \{open\} \rightarrow \{selected\} \rightarrow \{\} \rightarrow \ldots.$$

Lamport identified two types of temporal property [Lamport, 1977]. A *safety property* specifies that something bad does not happen, whereas a *liveness property* specifies that something good does eventually happen. While safety properties can be disproven by a finite execution leading to the undesired phenomenon, liveness properties can only be disproven by an infinite execution showing the absence of the desired phenomenon. The property discussed before is a liveness property. Alpern and Schneider showed that every property is the intersection of a safety and a liveness property [Alpern and Schneider, 1987].

Another temporal logic commonly used in model checking is *computation tree logic* (CTL) [Clarke and Emerson, 1982]. It is distinct from LTL in that it does not consider individual executions of the system, but a tree of executions.

This tree is obtained when the common prefixes of all executions of a transition system are collapsed. It branches every time there is a choice between two transitions. In CTL, each temporal operator is preceded by a path quantifier. The two quantifiers are E, requiring at least one path to satisfy a property; and A, requiring all paths to satisfy a property. The A quantifier can be obtained from E [Baier and Katoen, 2008], and will thus not be used in definitions or algorithms. The temporal operators *next*, *until* and *globally* have a similar meaning as in LTL.

Definition 2.6. *A CTL formula ϕ is an expression*

$$\phi ::= \quad 1 \mid a\ (\in AP) \mid \phi_1 \wedge \phi_2 \mid \neg\phi \mid E \bigcirc \phi \mid E(\phi_1 U \phi_2) \mid E\Box\phi$$

CTL formulas are interpreted over a transition system ts (or a Kripke structure) and a state s

$$\begin{array}{lll}
ts, s \models 1 & & \\
ts, s \models a & \iff a \in L(s) & \\
ts, s \models \neg\phi & \iff ts, s \not\models \phi & \\
ts, s \models \phi_1 \wedge \phi_2 & \iff (ts, s \models \phi_1) \wedge (ts, s \models \phi_2) & \\
ts, s \models E \bigcirc \phi & \iff \exists \pi \in paths(ts, s) \bullet ts, head(\pi_1) \models \phi & \\
ts, s \models E(\phi_1 U \phi_2) & \iff \exists \pi \in paths(ts, s) \bullet \exists j \geq 0 \bullet ts, head(\pi_j) \models \phi_2 & \\
 & \qquad \wedge\ \forall 0 \geq i < j \bullet ts, head(\pi_i) \models \phi_1 & \\
ts, s \models E\Box\phi & \iff \exists \pi \in paths(ts, s) \bullet \forall i \geq 0 \bullet ts, head(\pi_i) \models \phi &
\end{array}$$

Satisfaction for a transition system ts with a set of initial states I is given by $ts \models_{CTL} \phi \iff \forall s \in I \bullet ts, s \models \phi$.

Note that LTL and CTL are not equivalent. Both logics are subsets of the logic CTL*, which allows the path quantification operators to be mixed with the Boolean connectives.

In model checking there is a series of *"Great Debates"* [Holzmann, 2004], one of them being whether to use branching or linear time logics. In a linear time logic, such as LTL, each possible execution of the system is interpreted as a line of events. In a branching time logic, such as CTL, there is instead a single execution tree that branches each time different events can occur. A branching time logic allows to quantify over the branches at each point in the execution tree. There are various arguments for either of these logics [Vardi, 2001]. The theoretical worst-case complexity of CTL model checking is linear in size of the formula, whereas it is exponential for LTL. On the other hand, LTL formulae are considered more intuitive to write and understand. This debate also applies to SPLs. It thus seems natural for us to consider both CTL and LTL.

2.2.3 The model checking problem

The model checking problem is the decision problem that consists in determining whether a transition system satisfies a property. For LTL, a transition

system satisfies a property if all its executions satisfy the property. In this case, the output is *true*. Should the property be violated, the output is $false$. In practice, a model checker will also return an execution that demonstrates the property violation. This execution is commonly called *counterexample*.

Definition 2.7. *For a transition system t and an LTL formula ϕ,* $\text{SMC}_{LTL}(t, \phi)$ *returns true iff $t \models_{LTL} \phi$, and $false$ otherwise.*

The 'S' in SMC_{LTL} stands for 'single-system', to distinguish it from the model checking problem in SPLs, which we introduce later. For CTL, satisfaction means that the property is satisfied in all initial states.

Definition 2.8. *For a transition system t and a CTL formula ϕ,* $\text{SMC}_{CTL}(t, \phi)$ *returns true iff $t \models_{CTL} \phi$, and $false$ otherwise.*

In the example of Figure 2.1, suppose that state ⑥ is labelled with the proposition *selected* and state ⑦ with the proposition *open*. The example LTL property of the vending machine discussed before, $\square(selected \Rightarrow \Diamond open)$, indeed holds for the transition system. It is clear that in every execution of the transition system, state ⑥ is followed eventually by state ⑦, i.e., every *selected* is followed by an *open*.

2.3 Model checking algorithms

To check the satisfaction of a formula, model checking algorithms perform a systematic search in the state space. A number of techniques for this have been proposed in the past decades. One of the more prominent algorithms for checking the satisfaction of LTL properties is due to [Vardi and Wolper, 1986, Courcoubetis et al., 1992] and based on automata theory. An automaton is a concise representation for a set of words (its *language*). The basic principle of automata-theoretic model checking is that it is possible to construct an automaton whose language corresponds to the executions allowed by a property.

For safety properties, automata over finite words are sufficient.

Definition 2.9. *A finite automaton is a tuple $(Q, \Sigma, \delta, Q_0, F)$ where*

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

The language of a finite automaton A, $[\![A]\!]$, is the set of all words $\sigma_0 \dots \sigma_k$ with $\forall i \in [0, k] \bullet \sigma_i \in \Sigma$ so that there is an execution of the automaton $q_0 \dots q_{k+1}$ which starts in an initial state $q_0 \in Q_0$, respects the transition relation $\forall i \in [0, k] \bullet (q_i, \sigma_i, q_{i+1}) \in \delta$ and ends in a final state $q_{k+1} \in F$.

Liveness and LTL properties require automata over infinite words.

Definition 2.10. *A Büchi automaton is defined in the same way as a finite automaton. The language of a Büchi automaton A, $[\![A]\!]$, is the set of all* infinite *words $\sigma_0 \ldots$ with $\forall i \geq 0 \bullet \sigma_i \in \Sigma$ so that there is an execution of the automaton $q_0 q_1 \ldots$ which starts in an initial state $q_0 \in Q_0$, respects the transition relation $\forall i \geq 0 \bullet (q_i, \sigma_i, q_{i+1}) \in \delta$ and contains one or more accepting states infinitely often $I \cap F \neq \emptyset$ (where I is the set of states appearing infinitely often).*

This results in the following algorithm for LTL.

Algorithm 2.11. *Given a transition system ts and an LTL property ϕ, calculate $A_{\neg\phi}$, the Büchi automaton corresponding to $\neg\phi$ [Vardi and Wolper, 1986, Gastin and Oddoux, 2001]. Now test whether $[\![ts]\!]_{TS} \cap [\![A_{\neg\phi}]\!] = \emptyset$ [Vardi and Wolper, 1986] by executing the automaton jointly with the transition system. When an accepting execution is found, the property is violated. The algorithm then returns $false$ and the path to the current state. When all possible executions were considered and no accepting one was found, the property is satisfied and the algorithm returns $true$.*

The algorithm is essentially a search in a graph, which can be implemented with a nested depth-first search or a depth-first search combined with a breadth-first search. Such an algorithm is called *explicit*, as it visits states *one by one*. The computational complexity of this procedure is $O(2^{|\phi|}.|ts|)$, where $|\phi|$ is the size of the formula (number of operators) and $|ts|$ the size of the transition system (number of states and transitions). Altogether, the decision problem SMC_{LTL} is PSPACE-Complete [Sistla and Clarke, 1985, Schnoebelen, 2002].

The LTL algorithm is exponential in the size of the property, which was an obstacle for early approaches to this problem. Hence, initial research into model checking focussed on branching time logics similar to CTL (notably [Clarke and Emerson, 1982, Queille and Sifakis, 1982]), which are comparatively easier to compute. The CTL algorithm computes directly on the property and the transition system. It uses the parse tree of a formula to decompose it into subproblems that can be handled independently.

Definition 2.12. *Given a CTL formula ϕ, its* parse tree *is a graph, with ϕ as the root, where leaf nodes correspond to terminal formulae (i.e., 1 or a) and where each intermediate node corresponds to a production of Definition 2.6 with its children being the sub-formulae of this production.*

The CTL algorithm traverses the parse tree of the formula bottom-up. First, the states satisfying the formulae of the leaves are computed, then the information is used to compute the states satisfying the formulae of their parents and so on. The last step is to compute the states satisfying the whole formula. The intermediate results are referred to as *satisfaction sets*; they are sets of states that satisfy a particular sub-formula. The algorithm is given by a recursive specification of the satisfaction set calculation.

Algorithm 2.13. *Given are a transition system ts and an LTL property ϕ. For each sub-formula ϕ', starting with the smallest, calculate $Sat(\phi') \subseteq S$, the set of states that satisfy ϕ'. If $I \subseteq Sat(\phi)$, the property is satisfied and the algorithm returns true, otherwise it returns false.*

The crux of this algorithm is the calculation of $Sat(\phi_1 U \phi_2)$ and $Sat(\Box\phi)$. They can be computed by a backward search and by a strongly connected components analysis respectively [Clarke et al., 1986]. The computational complexity of the CTL algorithm is $O(|\phi|.|ts|)$, i.e., linear in the size of the formula. The decision problem SMC_{CTL} is P-Hard [Schnoebelen, 2002]. Note that the CTL algorithm does not directly produce a counterexample. It has to be calculated *a posteriori* [Clarke et al., 1995].

2.4 Advanced techniques in model checking

The model checking algorithms discussed above are part of the early results published about model checking. Research has since focussed on making model checking scale to systems of non-trivial size, and on making models and logics more expressive (e.g., by taking time into account [Alur and Dill, 1990], or replacing transition systems by Markov chains or Markov decision processes [Vardi, 1985, Vardi and Wolper, 1986, Baier et al., 2009]).

Among the topics that are currently the focus of the research community are bounded, probabilistic and software model checking. Bounded model checking [Biere et al., 1999] trades soundness for scalability and encodes the model checking problem as a SAT problem by unrolling the transition relation a finite number of times (i.e., the length of the considered executions is bounded). Probabilistic model checking enables the use of probabilistic models, such as Markov chains or Markov decision processes, to describe system behaviour. This allows to capture quantitative properties and model uncertainty. Software model checking is the application of model checking to program code rather than models [Corbett et al., 2000, Visser et al., 2000]. It addresses the problem of model construction. Constructing models is tedious, error prone, and it is hard to prove that the implementation conforms to the analysed models. Software model checking faces problems with scalability due to state explosion.

The goal of this thesis is to provide a basis for the modelling and verification in SPLs. We will thus revisit the basic theories and algorithms in model checking, and see how they apply to SPLs. Most of the topics that are currently fashionable in model checking research will thus not be touched upon, and are reserved for future work. Let us briefly discuss the advanced techniques in model checking that are relevant to this work.

2.4.1 Symbolic model checking

A number of solutions have been proposed to the state explosion problem. An important one is the use of symbolic encodings of the state space [McMillan,

1993]. Given a transition system, a symbolic representation is a compact data structure for a large set of states and transitions. Symbolic algorithms have been shown to be applicable in many cases where exhaustive techniques do not scale [Burch et al., 1992].

In model checking, a widely used data structure for this is the *Reduced Ordered Binary Decision Diagram* (BDD), first proposed by Bryant [Bryant, 1992]. In the symbolic setting, sets of states and the transition relation are encoded directly with their characteristic functions. The characteristic functions are then turned into Boolean functions, which are represented by BDDs. BDDs are often compact and Boolean operators such as conjunction, disjunction and negation (corresponding to intersection, union and complementation of the sets they represent) can be computed efficiently on them.

As a starting point we assume the existence of a binary encoding of states, that is, a function $enc : S \to \{0,1\}^k$, where k is chosen large enough to encode all states. With this encoding, $\{0,1\}^k$ implicitly denotes the sets of all (encoded) states. Any subset of states $T \subseteq S$ can be represented by its characteristic function, χ_T, that is

$$\chi_T(\overline{s}) : \{0,1\}^k \to \{0,1\} \bullet \chi_T(enc(s)) = 1 \iff s \in T$$

χ stands for *characteristic function* [Baier and Katoen, 2008]. The subscript of χ, e.g., '$X \cup Y$' in $\chi_{X \cup Y}$, denotes the set for which this is the characteristic function. In parentheses follow the variables on which the function is defined. By convention, $\overline{s}$ denotes a vector of variables encoding a state. The *cofactor* $\chi_{T[\overline{s} \leftarrow enc(x)]}$ of a Boolean function $\chi_T(\overline{s}, \overline{p}, \ldots)$ is the function over the variables $\overline{p}, \ldots$ obtained by replacing the variables $\overline{s}$ by the value they take in $enc(x)$. In a tool implementation, each $\chi(x_1, .., x_k)$ becomes a BDD over variables $x_1, .., x_k$.

A BDD is a directed acyclic graph with a single root. Each vertex represents a variable, and two terminal vertices represent 1 and 0. The variable vertices all have two outgoing edges, one meaning that its value is 1 and one 0. Given a value assignment, these edges are such that following them from the root leads either to the 1 or to the 0 vertex, denoting the output of the function for this assignment. BDDs can be minimised, and the minimal size of a BDD depends on the ordering chosen for its variables.

Symbolic model checking algorithms compute directly on the BDDs, that is, on sets of states rather than on individual states. The CTL algorithm can easily be given in terms of set computations, which is why most symbolic model checking approaches focus on CTL. Algorithm 2.13 is already specified in terms of set operations, except for the computation of EU and $E\Box$. These can be computed by a fixed-point algorithm over the set of states [Clarke et al., 1999].

2.4.2 Generalised and parameterised model checking

A number of approaches have been proposed whose strategy is to prove satisfaction or violation of a property with the smallest possible state space. An example of such an approach that has become rather successful is *counterexample-*

guided abstraction refinement [Clarke et al., 2000]. The technique starts with a coarse abstraction of a system to be verified (generally program code), and then refines this abstraction based on inconclusive model checking results, until the property is either proven satisfied or violated.

Generalised model checking addresses state-space reduction in a different way [Bruns and Godefroid, 2000]. The idea is to check a property against a partial state space. As the state space is partial, the algorithm might be unable to determine whether or not the property is satisfied. This leads to a three-valued model checking result: satisfied, violated or unknown. The generalised model checking problem can be solved by two normal model checks [Bruns and Godefroid, 2000,Godefroid and Piterman, 2009]. In one check, the partial state space is completed optimistically, meaning that if the property is violated in this case, it is also violated by the full state space. In the other check, the partial state space is completed pessimistically so that if the property is satisfied in this case, it would also be satisfied by the full state space. If a property is satisfied by the optimistic check and violated by the pessimistic check, it is unknown whether the property is satisfied or violated by the full state space. Generalised model checking can be seen as a middle ground between classical model checking and satisfiability checking, that is, checking whether a model exists that satisfies a given temporal property. It generalises both [Bruns and Godefroid, 2000]. The algorithms presented in Part II are similar in that they are capable of determining the values of a number of parameters for which a given model satisfies a temporal property. However, they always produce an answer (there is no 'unknown').

There is a body of research into the problem of *parameterised model checking* [Emerson and Namjoshi, 1996]. The goal of these techniques is to model check an arbitrary number of parallel processes. The parameter thus determines the number of processes, which is different from our work where parameters are Boolean variables that determine the behaviour of individual processes.

A related approach is that of *temporal logic queries* [Chan, 2000, Bruns and Godefroid, 2001]. A temporal logic query is a temporal logic formula with a placeholders. Solving a query for a given transition system yields a propositional expression which when substituted for the placeholder makes the property true in the transition system. The model checking problem for SPLs as defined in Part II can be expressed as a particular kind of temporal query. Our algorithms compute an expression over the feature variables characterising the products for which a temporal logic formula holds. Our work is original in that we use a semi-symbolic model checking algorithm and apply it to SPLs.

2.4.3 Symbolic execution

With the advent of software model checking [Holzmann and Smith, 1999a, Holzmann and Smith, 1999b, Corbett et al., 2000, Visser et al., 2000, Visser et al., 2003], a number of static analysis techniques found their way into model checking. The biggest problems in software model checking are state explosion

due to the large number of variables, and infinite state spaces due to language features like dynamic allocation. Static analyses, such as program slicing can be used to filter out variables and threads that are not relevant to properties being analysed [Corbett et al., 2000]. Furthermore, compression techniques have to be used to be to store large state spaces in memory [Holzmann, 1997].

Another technique to reduce the state space is to abstract the domains of variables into the properties of interest. E.g., instead of considering all possible lists, only the case of an empty list, and that of a non-empty list are considered [Corbett et al., 2000, Visser et al., 2000, Visser et al., 2003]. A natural extension of this is *symbolic execution*, a type of abstract interpretation in which program code (or some high-level model) is executed with symbolic values [King, 1976]. This means that instead of having fixed values, the variables are defined by expressions, and all statements of the program are computed over these expressions. Symbolic execution predates model checking. However, it has become highly relevant for model checking high-level modelling languages or program code.

Symbolic execution builds a tree of the various paths through the code. Each path has a *path condition*, an expression that symbolically represents the input values for which the path can be taken. Such expressions can then be checked for satisfiability with a SAT solver (or similar) to yield information about reachable paths, or erroneous paths. In [Khurshid et al., 2003], the authors introduce *lazy initialisation* which uses symbolic execution to reduce the state space by initialising variables as late as possible.

The algorithms presented in Part II use symbolic execution limited to certain variables. However, unlike in existing approaches, the goal is to compute sound and complete symbolic expressions that are of interest to the engineer.

Chapter 3

Related Work

> *The greatest triumph that modern PR can offer is the transcendent success of having your words and actions judged by your reputation, rather than the other way about.*
>
> Christopher Hitchens, in *Salon*, 1998

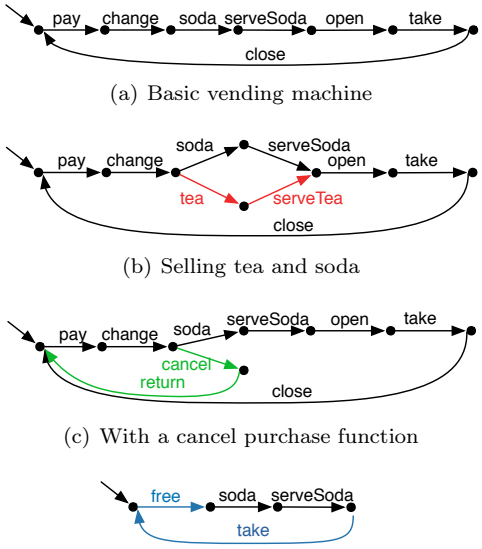
Now that we introduced the context of this work, we are equipped with the necessary concepts to present the motivation given at the very beginning more precisely. We revisit the problem and its motivation in Section 3.1. In Section 3.2, we discuss state-of-the-art approaches that aim to tackle this problem.

3.1 Motivation

The starting point of this thesis is the observation that quality assurance in SPLs should already be conducted during domain engineering, not only during application engineering. Current techniques for quality assurance, however, were not designed to deal with variability inherent in domain artefacts. Our ultimate goal is thus to study quality assurance at the domain engineering level. Since quality assurance comprises a vast range of activities and methods, we refined the scope of this thesis to a particular technique: model checking.

Let us illustrate the challenges faced for model checking approaches in domain engineering with the example of the vending machine. As formalised with the FD of Figure 1.2, the product line of beverage vending machines has 24 products. In terms of model checking, the behaviour of each one of them is a transition system. In Figure 3.1, we give four of these transition systems.

The one in Figure 3.1(b) was already discussed in Section 2.2. The other variants shown are a machine that only sells soda, in Figure 3.1(a). One that lets the customer cancel her purchase after entering a coin, in Figure 3.1(c). A third one offers free drinks and has no closing beverage compartment, see Figure 3.1(d). Each of them highlights one of the four central features of the



(a) Basic vending machine

(b) Selling tea and soda

(c) With a cancel purchase function

(d) Distributing soda for free

Figure 3.1: Several variants of a vending machine.

vending machine product line: *Soda*, *Tea*, *CancelPurchase* and *FreeDrinks*. The other features do not cause behavioural variations. As explained before, we abstract away from coins and currency in the behavioural model.

By combining these variants, the behaviour of the other 20 vending machines can be obtained. That is, a model of the behaviour of a small example such as this would already require 24, largely identical, behavioural descriptions. For realistic cases, this number is so high that it is outright impossible to model each product individually. Furthermore, conducting model checking at the domain engineering level means that checks should be performed over all those 24 products. Again, given that most of them are largely identical or have substantial overlap, executing a classical model checking algorithm on each one of them would seem to be inefficient.

This example illustrates the two main challenges to be addressed in this work: (a) scalable modelling and (b) efficient verification of variability-intensive system behaviour. Existing methods in mainstream model checking, especially state space reduction techniques, do not consider the case of variability at all. Still, a number of approaches for modelling and verification of product lines have been proposed in the past years. However, these proposals generally suffer from two main limitations.

Firstly, their behavioural models often fail to recognise the importance of features as a unit of difference. This means that they capture the behaviours of different products, but offer little to no means to relate these behaviours to

products or features. In many cases, features are not even first-class entities in the models. A consequence of this is that these approaches cannot make use of information contained in models such as FDs, e.g., the co-occurrence or mutual exclusion of features. Secondly, none of the proposals provide concrete means for checking behavioural models against temporal properties.

Let us now review the state of the art in quality assurance of SPLs, including existing proposals that address the above challenges.

3.2 State of the art

Many of the early approaches to modelling variability in behaviour do not consider verification, offering at best means for structural integrity checks. These approaches are generally based on UML, like [Ziadi et al., 2003] a UML profile for variability with stereotypes for optionality, alternatives and refinement. Another UML-based approach is described in [Czarnecki and Antkiewicz, 2005, Czarnecki and Pietroszek, 2006] in which variability is modelled by specifying *presence conditions* of model fragments. These are Boolean expressions that define to which products a model fragment belongs. Both approaches allow modelling of SPL behaviour with the corresponding UML diagram types (sequence diagrams, state machines). Verification can be accomplished using existing methods and tools, as in [Kishi and Noda, 2006, Liu et al., 2007], by deriving the model of a specific product. If conducted during domain engineering, such approaches face a scaleability problem as the number of products is potentially huge.

The more formal approaches to modelling SPL behaviour are based on modal transition systems (MTS) [Larsen, 1989, Fischbein et al., 2006] and modal I/O automata [Larsen et al., 2007]. In these approaches, transitions can be mandatory (*required* transitions) or optional (*allowed* transitions). As expected, *allowed* transitions can be used to model variability, and an MTS essentially specifies a family of transition systems. Fantechi and Gnesi extend MTS by introducing variability operators that allow to specify cases in which a specific number of outgoing transitions may be taken [Fantechi and Gnesi, 2008, Fantechi and Gnesi, 2007]. In [Asirelli et al., 2009, Asirelli et al., 2010b, Asirelli et al., 2010a], the authors propose a deontic logic interpreted over MTS, that can be used to express both behavioural properties, and constraints over features. Gruler *et al.* adapt the CCS process algebra [Milner, 1980] in a similar way [Gruler et al., 2008b, Gruler et al., 2008a]. Their PL-CSS algebra has an operator that allows to model variability in the form of alternative choice between two processes.

All these approaches suffer from the limitation mentioned before: they do not allow to relate behaviours to products. An MTS, for instance, does not capture the features that make a transition optional; similarly, choices between processes in PL-CSS are not linked to features. In consequence, when a verification algorithm finds an execution that violates a property, there is no

information to determine the products that exhibit this execution (i.e., those that violate the property) or the features that enable such an execution. The more recent approach of Lauenroth *et al.*, which shares some similarity with this work, does overcome this limitation [Lauenroth and Pohl, 2008, Lauenroth et al., 2009]. Just as the algorithms we propose in this thesis, those in [Lauenroth et al., 2009] are capable of determining the products for which a certain temporal property holds. However, the algorithms are inefficient (exponential in the size of the state space, see Section 7.4), the modelling formalism not thoroughly studied, and no high-level languages are proposed.

Those are all the approaches to modelling and verification of SPLs that are currently proposed in the literature. There are a number of other relevant techniques, which do not address SPL verification directly.

Morin *et al.* propose a method to check for inconsistencies between features in adaptive systems [Morin et al., 2009]. Instead of verifying all possible combinations at design time, they verify a feature combination when it is activated at runtime, which is prevented in case of an inconsistency. This verification, however, only covers structural properties of the system. Furthermore, testing the validity of a product when it is chosen at run-time, and refusing the choice if it is invalid, is not a user-friendly solution and does not seem to be a viable approach in the case of embedded systems.

In the context of workflow modelling, van der Aalst *et al.* propose workflow templates that contain variation points [van der Aalst et al., 2008]. The authors propose a technique for configuring workflow models incrementally, continuously verifying that they are deadlock free. However, verifying temporal properties over all workflow instances upfront is not possible with their approach. Furthermore, the predefined properties being verified during configuration are specific to workflows and cannot be readily changed.

Fisler, Krishnamurthi and Li propose a compositional approach for CTL model checking of collaborations, a feature-like concept [Fisler and Krishnamurthi, 2001, Li et al., 2002b, Li et al., 2002a]. In this approach, the automaton of a feature can be attached to precisely defined interface states of a base system. The advantage of this is that each feature can be verified in isolation, which overcomes the problem of exponentially many feature combinations. The disadvantage is lost expressiveness: features can only add sequentially at the interface and cannot remove transitions or states. Compositional verification has also been studied in the context of aspect-oriented systems [Krishnamurthi et al., 2004, Goldman and Katz, 2007, Katz and Katz, 2008]. In [Goldman and Katz, 2007], the authors introduce the MAVEN approach, in which assumptions and guarantees of an aspect are specified and aspects can be verified in isolation. Such approaches have not yet been studied in the context of SPLs. Furthermore, compared to annotation-based approaches, aspects are only rarely used for implementing features in SPLs (beyond the research literature).

In addition to the above, there is a body of related research in the field of feature interaction detection [Calder et al., 2003]. Feature interaction research

lacks the product line perspective: their techniques generally focus on pairwise checks and do not deal with the problem of an exponential number of possible feature combinations. Contrary to this, we aim to provide complete techniques. Our results have the potential to supersede existing approaches in feature interaction detection.

When it comes to quality assurance in SPLE in general, the picture is also rather bleak. Unlike in normal (single systems) development, none of the implementation approaches in FOSD have any form of *built-in* correctness. Classical correctness notions, e.g., those in compilation and testing, only apply during application engineering, once a specific product is generated. This is true even for seemingly easy to catch errors like type errors [Thaker et al., 2007]. For annotative approaches, a fundamental challenge is also the parsing of annotated code [Garrido and Johnson, 2005, Kästner et al., 2010]. There has been some progress over the last years, in particular regarding type checking [Kästner and Apel, 2008]. However, type checking of annotated code is a structural verification problem and cannot be used to check specifications, for instance.

A number of approaches for testing in SPLE have been proposed [Cohen et al., 2006, Cohen et al., 2007, Oster et al., 2010, Perrouin et al., 2010]. In testing, just as in model checking currently, there are no techniques to deal with variability in domain artefacts. Testing always operates on particular products. To assure a reasonable coverage of the space of products, combinatorial techniques are used. Coverage criteria are, for instance, that every pair of features is covered. Testing in SPLE is thus a rather straightforward extension of current testing techniques.

Part II

Foundations

Chapter 4

Featured Transition Systems

> *. . . In that Empire, the craft of Cartography attained such Perfection that the Map of a Single province covered the space of an entire City, and the Map of the Empire itself an entire Province. In the course of Time, these Extensive maps were found somehow wanting, and so the College of Cartographers evolved a Map of the Empire that was of the same Scale as the Empire and that coincided with it point for point. Less attentive to the Study of Cartography, succeeding Generations came to judge a map of such Magnitude cumbersome, and, not without Irreverence, they abandoned it to the Rigours of sun and Rain. In the western Deserts, tattered Fragments of the Map are still to be found, Sheltering an occasional Beast or beggar; in the whole Nation, no other relic is left of the Discipline of Geography.*
>
> Jorge Luis Borges, *On Exactitude in Science*, 1946

The starting point and foundation of this work is a formalism for describing the behaviour of an SPL. There is one key criterion that such a formalism should have. It derives from the observation that transition systems are a generic and well studied fundamental model for the behaviour of single systems. They are the starting point of most developments in model checking. In consequence, a formalism for SPL behaviour should describe the behaviour of each product as a transition system.

In this chapter, we present one such formalism, *featured transition systems* (FTS), and study its properties. Section 4.1 gives an intuitive introduction to FTS, while formal definitions are given in Section 4.2. In Section 4.3, we discuss parallel composition of FTS. We study formal properties of FTS and compare them to other languages in Section 4.4. In Section 4.5, we give a number of sizeable examples before we conclude the chapter in Section 4.6.

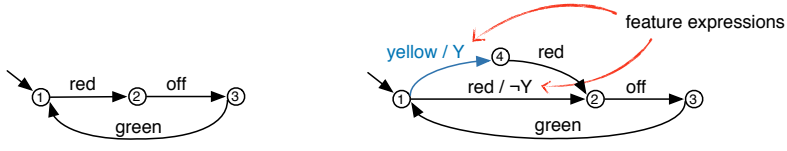
4.1 Introduction

The first challenge identified in Chapter 3 is that of scalable modelling. The goal is that the size of a model, measured in number of elements, increases linearly with the number of features; unlike the number of products which increases exponentially. This, of course, precludes a formalism in which products are modelled individually, or are otherwise first-class concepts. A basic red lights controller, for instance, will switch between red and green. So far, the controller can be modelled with a conventional transition system, as shown in Figure 4.1(a). Assume that another version (i.e., *product*) shows yellow before switching to red. It can be modelled by a second transition system with an additional state and transition ① $\xrightarrow{yellow}$ ④ $\xrightarrow{red}$ ②, and so forth for every product. This approach does not scale to a large number of different products, even if they only differ in small details. It is preferable that a single model represents these instances, in order to exploit the similarities between them.

Nevertheless, it should be possible to relate behaviours to products. A model which captures many behaviours, but offers no means to identify the product(s) that exhibit a particular behaviour, cannot be used to identify problematic products and would thus not be of much use in SPLE. A way of achieving this is by making features first-class concepts of the formalism. This has to be done in such a way that behaviours can be related to features, which, in turn, can be related to products.

The question thus becomes: how should features be captured as part of a transition system? A look at the transition systems in Figure 3.1 shows that the impact of adding a feature to a product is either to *add* states and transitions (as do features *Tea* and *CancelPurchase* in Figures 3.1(b) and 3.1(c) respectively) or to *remove* states and transitions (as does feature *FreeDrinks* in Figure 3.1(d)). In consequence, a formalism for SPL behaviour should be able to capture addition and removal of single transitions and states. Note that it is sufficient to capture the addition or removal of transitions. Through the addition or removal of incoming transitions, states become reachable or unreachable, which is as if they were added or removed themselves.

The formalism we propose for modelling SPL behaviour, *featured transition systems* (FTS), adopts this mechanism. In FTS, each transition has an anno-



(a) The basic version without variability. (b) With one additional feature.

Figure 4.1: FTS of the red lights controller.

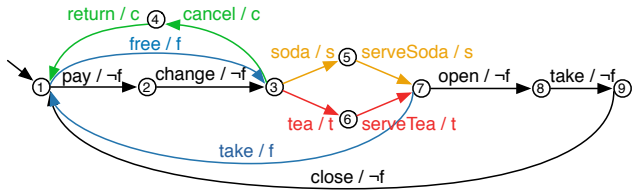


Figure 4.2: FTS of the vending machine.

tation (or *labelling*) that specifies to which products it belongs. This is similar to the annotative style in FOSD discussed in Section 1.3. To model both red lights systems in FTS, it is sufficient to add two transitions ① $\xrightarrow{yellow}$ ④ $\xrightarrow{red}$ ② as described before, and to label the first one with a different feature, say Y. The fact that there are two variants of the system, one with Y and one without, is documented in an FD.

This is how FTS handles the situation in which a feature *adds* transitions to a transition system. The Y feature also *removes* the ① $\xrightarrow{red}$ ② transition. In FTS, this is done by labelling ① $\xrightarrow{red}$ ② with the Boolean expression $\neg Y$. Such Boolean expressions are called *feature expressions*. This way, only one of the transitions leaving ① can exist in a product. The resulting FTS is shown in Figure 4.1(b). Features in FTS can thus be non-monotonic, i.e., remove transitions (and thus remove behaviours).

Now consider the vending machine example. Figures 3.1(b) and 3.1(c) show the impact of adding features *Tea* and *CancelPurchase* to a machine serving only soda: both add two transitions. In consequence, those transitions are labelled with the feature they belong to. The *FreeDrinks* feature replaces ① $\xrightarrow{pay}$ ② $\xrightarrow{change}$ ③ by the ① $\xrightarrow{free}$ ③ transition and ⑦ $\xrightarrow{open}$ ⑧ $\xrightarrow{take}$ ⑨ $\xrightarrow{close}$ ① by ⑦ $\xrightarrow{take}$ ①. The replaced transitions are thus labelled with $\neg f$, which means that they only belong to products without *FreeDrinks*, whereas the replacing transitions are labelled with f. The corresponding FTS is given in Figure 4.2. The feature label of a transition is shown next to its action label, separated by a slash. The colours of the transitions correspond to those used in Figure 1.2.

4.2 Syntax and semantics of FTS

As shown in the examples, an FTS is a transition system with an additional labelling function and an FD:

Definition 4.1. *An FTS is a tuple* $(S, Act, trans, I, AP, L, d, \gamma)$*, where*

- $(S, Act, trans, I, AP, L)$ *is a transition system, as defined in Definition 2.3,*
- d *is an FD as defined in Definition 1.1,*
- $\gamma : trans \to \mathbb{B}(N)$ *is a total function, labelling each transition with a* feature expression*, i.e., a Boolean expression over the features.*

Feature expressions work similarly to presence conditions in [Czarnecki and Antkiewicz, 2005]. The transition system of a particular product is obtained by removing all transitions whose feature expression is not satisfied. This operation is called *projection*.

Definition 4.2. *The projection of an FTS* $fts = (S, Act, trans, I, AP, L, d, \gamma)$ *to a product* $p \in [\![d]\!]_{FD}$*, noted* $fts_{|p}$*, is the transition system*

$$ts = (S, Act, trans', I, AP, L) \quad with \quad trans' = \{t \in trans \mid p \in [\![\gamma(t)]\!]\}.$$

The four transition systems of Figure 3.1 can be obtained from the vending machine FTS fts in Figure 4.2 with the projections:

$$(a)fts_{|\{v,b,s\}}, \quad (b)fts_{|\{v,b,s,t\}}, \quad (c)fts_{|\{v,b,s,c\}} \text{ and } \quad (d)fts_{|\{v,b,s,f\}}.$$

Each transition system obtained by projection describes the behaviour (as in Definition 2.2) of a particular product. This corresponds to the criterion discussed at the beginning. The FTS itself describes the behaviour of all products in the SPL. The semantics of an FTS is thus defined as the union of the behaviours of the projections on all valid products.

Definition 4.3. *The semantic domain of FTS is that of transition systems: the power set of the (infinite) set of all finite and infinite executions. The semantics of a particular FTS is given by*

$$[\![fts]\!]_{FTS} = \bigcup_{p \in [\![d]\!]_{FD}} [\![fts_{|p}]\!]_{TS}$$

It is important to point out that this is not equivalent to removing d and γ from an FTS and interpreting it as a transition system. In other words, the FTS semantics from Definition 4.3 is not equal to the transition system semantics of Definition 2.2. In general, the following theorem holds.

Theorem 4.4. *Let* $TS(fts)$ *be the transition system obtained by removing* d *and* γ *from an FTS* fts*, then for any* fts*,*

$$[\![fts]\!]_{FTS} \subseteq [\![TS(fts)]\!]_{TS}.$$

Proof. Let ts_1 and ts_2 be two transition systems that are identical except for $trans_1 \subseteq trans_2$. Clearly, $[\![ts_1]\!]_{TS} \subseteq [\![ts_2]\!]_{TS}$. The theorem follows from this, since all projected transition systems have less transitions than $TS(fts)$ and are otherwise identical. □

The transition system interpretation of an FTS has thus *more behaviours*. In the vending machine SPL, for example, a valid execution $e \in [\![TS(fts)]\!]_{TS}$ is one in which the vending machine would ask the first customer for a coin and offer a free drink to the next one. This is not admitted by the FTS semantics, according to which a machine should either always offer free drinks or always

require payment: $e \notin [\![fts]\!]_{FTS}$. The choice between the transitions ① $\xrightarrow{pay}$ ② and ① $\xrightarrow{free}$ ③ is non-deterministic in the transition system interpretation. In an FTS, they are alternatives depending on the *FreeDrinks* feature.

A corollary of Theorem 4.4 is that one cannot use classical model checking algorithms directly on an FTS. They would produce sound but incomplete results, i.e., they might find properties to be violated when they are in fact satisfied (a false positive).

Assuming that the behaviour of an SPL is given as a finite set of transition systems (one for each product), it is clear from Definition 4.1 that any such set can be represented as an FTS. By labelling transitions with feature expressions, FTS overcome the limitations of existing modelling languages for SPL behaviour: they have the ability to represent the behaviour of any SPL concisely, and any execution of an FTS can be traced to one or more products. Later on, we will further show that FTS are more concise than transition systems. Before we do this, let us discuss parallel composition of FTS.

4.3 Parallel composition

An important tool for the design of large scale systems is *parallel composition*. Generally, parallel composition is used if a system consists of parallel processes. Each process is modelled separately, and the parallel composition yields the final system. For example, the FTS of the red lights system by itself is rather limited. Vehicles have to be added to make it interesting. The vehicles are a typical example of parallel processes: they operate independently from each other. It would thus be natural to specify the FTS of such a system as the parallel composition of the red lights FTS and several FTS modelling the cars (see, e.g., [Magee and Kramer, 2006]).

There are several different execution models for parallel composition. Generally, the composed systems are executed asynchronously in parallel, that is, their executions are interleaved. For the purpose of this thesis, we adopt the handshake execution model, in which certain transitions are executed synchronously. The transitions that are executed synchronously are those with *shared actions*. This is a rather common model of parallel composition. However, other models exist and can be adapted to FTS in a similar way.

We already discussed the parallel composition of transition systems in Section 2.2.1 and Definition 2.4. To adapt it to FTS, the two additional elements of the FTS definition have to be taken into account: the FDs and the labelling with feature expressions. Before we do this, we should discuss what parallel composition in SPLs actually means.

4.3.1 Two types of parallel composition

Conceptually, an FTS captures the behaviour of an SPL. The parallel composition of two FTS thus entails some kind of composition of the SPLs they

represent. More precisely, we can distinguish several cases. One case is that each FTS represents a fragment of the behaviour of the same SPL, which means that their parallel composition represents the same SPL, but without the fragments. Finally, each FTS could also represent a different SPL (or a different instance of the same SPL), in which case their parallel composition represents yet another SPL, obtained by composing the initial SPLs.

An FD captures the essence of an SPL, its variability and its set of products. Furthermore, each FTS comes with its own FD. The two situations described in the previous paragraph can thus be characterised by whether or not the FDs of both FTS are identical. When two FTS have identical FDs, it is safe to assume that they belong to the same SPL. Their parallel composition thus happens *within this SPL*. We call this *Intra-SPL* composition. On the other hand, when the two FTS have different FDs, it is safe to assume that they represent different SPLs. In this case, the parallel composition has to combine two FDs into a new FD (hence two SPLs into a new SPL). We call this *Inter-SPL* composition. The case in which each FTS represents a different instance of the same SPL can be reduced to Inter-SPL composition by renaming the features in the FDs (e.g., by appending indices). Let us expand on this distinction.

Intra-SPL composition is a situation in which parallel composition occurs within the same SPL. In terms of FTS, this means that the FTS being composed all have the same FD, and that the result of the composition also has this FD. Intra-SPL composition thus does not create new products. Rather, each product is defined as the parallel composition of several distributed processes, each corresponding to one of the composed FTS. In the case of the red lights system with cars, Intra-SPL composition would correspond to a situation in which the cars do not have variability. They just provide a context for the red lights controller, and are hence part of its SPL.

Inter-SPL composition is a situation in which parallel composition occurs between different SPLs, that is, SPLs that do not share the same sets of products. The products of the resulting FTS are parallel compositions of products (hence of transition systems) of the operands. Product generation by composition is needed, for instance, in the automotive industry [Reiser and Weber, 2006]. Each electronic control unit in an automobile can be thought of as an SPL. These electronic control units are typically provided by different manufacturers and communicate over a bus. They form a distributed system. The difference wrt. the previous case is that parallel composition is no longer between two SPLs with the same products, but happens between SPLs with different products. Basically, each product of one SPL will be put in parallel with each product in the other SPL. The result of the parallel composition is thus defined over a set of products obtained by combining the sets of products of the original SPLs.

Parallel composition in FTS should support both of these cases. For Inter-SPL composition, we thus have to investigate the composition of the FDs that are associated to the FTS (in addition to parallel composition itself). Basically, we need a binary operator $\oplus : FD \times FD \to FD$ that combines FDs. It should be commutative and idempotent. If $\oplus$ is idempotent, Intra-SPL composition becomes a special case of Inter-SPL composition in which all FDs are equal.

Schobbens *et al.* specify three composition operations on the level of products [Schobbens et al., 2007]. *Intersection* of the sets of products, $[\![d_1]\!]_{FD} \cap [\![d_2]\!]_{FD}$: this is a very restrictive operation as it assumes that the sets of products of both SPLs overlap. This would require both SPLs to have the same set of features, which is most likely not a safe assumption. *Union* of the sets of products, $[\![d_1]\!]_{FD} \cup [\![d_2]\!]_{FD}$: this operation is problematic as it does not *combine* the products of both SPLs. The *reduced product* of the set of products, $\{p_1 \cup p_2 \mid p_1 \in [\![d_1]\!]_{FD}, p_2 \in [\![d_2]\!]_{FD}\}$: this operation combines the products of both SPLs. However, it is not idempotent.

None of these composition mechanisms corresponds to our needs. We thus propose the following alternative mechanism.

Definition 4.5. *Given a set of features N and two FDs d_1 and d_2 with sets of features $N_1, N_2 \subseteq N$, their* join $d_2 \oplus d_2$ *is an FD with*

$$[\![d_2 \oplus d_2]\!]_{FD} = \{p \in N_1 \cup N_2 \mid (p \cap N_1) \in [\![d_1]\!]_{FD} \wedge (p \cap N_2) \in [\![d_2]\!]_{FD}\}$$

The join of two SPLs is similar to the reduced product in that it combines the products of both SPLs. However, contrary to the reduced product, it conjoins constraints defined in both SPLs (if two features are exclusive in one SPL, but not in the other, they will be exclusive in their join). Moreover, the join operation is idempotent.

Like the definitions from [Schobbens et al., 2007], Definition 4.5 only characterises a composition operator. It does not immediately translate to a procedure that operates on FDs. Since the study of syntactical procedures for FD merging falls out of the scope of this thesis, we content ourselves with the specification of the join operator. The interested reader is referred to [Acher et al., 2009] who study generic merge procedures for FDs. Nevertheless, the join operator can readily be implemented on the Boolean function encoding of the FDs (see Section 1.2). The following theorem is a corollary of Definition 4.5.

Theorem 4.6. *Given a set of features N and two FDs d_1 and d_2 with sets of features $N_1, N_2 \subseteq N$, then $[\![d_2 \oplus d_2]\!]_{FD} = [\![\mathbb{B}(d_1) \wedge \mathbb{B}(d_2)]\!]$.*

In a tool implementation, this will most likely be sufficient, since tools generally operate on the Boolean function encoding of the FDs.

4.3.2 Parallel composition of FTS

The other question in adapting parallel composition to FTS is what happens to the feature expressions. This is rather straightforward. The feature expressions

of asynchronous transitions are those of the original transitions. The feature expressions of synchronous transitions are conjoined.

This leads to the following definition of parallel composition.

Definition 4.7. *Given two FTS* $fts_i = (S_i, Act_i, trans_i, I_i, AP_i, L_i, d_i, \gamma_i)$, *with* $i \in \{1, 2\}$, *their parallel composition synchronised over the set of shared actions* $Act_1 \cap Act_2$, *written* $fts_1 || fts_2$, *is the FTS*

$$(S_1 \times S_2, Act_1 \cup Act_2, trans, I_1 \times I_2, AP_1 \cup AP_2, L, d_1 \oplus d_2, \gamma)$$

where

- $L((s_1, s_2)) = L(s_1) \cup L(s_2)$
- *trans is the smallest relation satisfying*
  - *for* $\alpha \notin Act_1 \cap Act_2$ *(interleaving):* $\dfrac{s_1 \xrightarrow{\alpha}_1 s_1'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2)}$ $\dfrac{s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1,s_2')}$
  - *for* $\alpha \in Act_1 \cap Act_2$ *(synchronisation):* $\dfrac{s_1 \xrightarrow{\alpha}_1 s_1' \wedge s_2 \xrightarrow{\alpha}_2 s_2'}{(s_1,s_2) \xrightarrow{\alpha} (s_1',s_2')}$
- *For* $\alpha \notin Act_1 \cap Act_2$, $\begin{cases} \gamma\big((s_1, s_2) \xrightarrow{\alpha} (s_1', s_2)\big) = \chi_1 \\ \gamma\big((s_1, s_2) \xrightarrow{\alpha} (s_1, s_2')\big) = \chi_2 \end{cases}$
  for $\alpha \in Act_1 \cap Act_2$, $\gamma\big((s_1, s_2) \xrightarrow{\alpha} (s_1', s_2')\big) = \chi_1 \wedge \chi_2$,
  where for $i \in [1, 2]$, $\chi_i = \gamma(s_i \xrightarrow{\alpha}_i s_i')$.

An important property of this parallel composition operator is that it is equal to the one for transition systems (Definition 2.4) modulo projection. This is formalised by the following theorem.

Theorem 4.8. *For any two FTS,* fts_1 *and* fts_2, *with the same feature diagram* d, *and for any product* $p \in [\![d]\!]_{FD}$, $fts_{1\,|p} || fts_{2\,|p}$ *is syntactically equivalent to*

$$(fts_1 || fts_2)_{\,|p}.$$

This corresponds to the characterisation of Intra SPL composition given earlier: each product is the parallel composition of several transition systems.

Let us point out one special case of parallel composition. In a system which interacts with the environment (such as the mine pump system presented later in Section 4.5.2), elements of the environment are most likely modelled as parallel FTS. Since these FTS are not part of the SPL, they cannot be labeled with any feature. When composing the environment with the FTS of the system, each synchronised transition should simply inherit the labelling of the system FTS. This can be achieved if all transitions of the environment FTS are labelled with the feature expression *true*. In our first FTS implementation, this was the only kind of parallel composition supported. In this case, Definition 4.7 degenerates into something close to Definition 2.4.

4.4 Expressiveness

The *expressiveness* of a formalism is a measure of its ability to express the various subjects that the formalism is meant to model. Formally, the expressiveness of a language (viz. modelling language) is the set of words (viz. models) that can be expressed in the language [Hopcroft et al., 2000]. The expressiveness is generally measured in comparison to the set of words that are in the semantic domain of the language. A language it is said to be *fully expressive* if it covers the entire semantic domain [Trigaux, 2008]. Another notion used when evaluating characteristics of languages is *succinctness*. It is a measure of the relative size of the same model in two different languages.

In the following, we will study these properties in the context of FTS and other languages.

4.4.1 Preliminaries

Trigaux formally defined the notions of expressiveness and succinctness [Trigaux, 2008] in the context of the framework of [Harel and Rumpe, 2000]. Given a language L, its expressiveness is the codomain of the semantic function.

Definition 4.9. *The expressiveness of a language L with syntactic domain $syn(L)$ and semantic domain $sem(L)$ is the set $exp(L) \triangleq \{[\![l]\!] \mid l \in syn(L)\}$. A language L is more expressive than a language L' iff $exp(L) \supseteq exp(L')$. A language L is fully expressive iff $exp(L) \supseteq sem(L)$.*

Of course, languages can only be compared in terms of expressiveness if their semantic domains are equal. Definition 4.9 is declarative and does not give a practical method for comparing the expressiveness of two languages. This is done by providing a *translation* from one language to the other. If it is possible to translate any model of a language L into a model of another language L' with the same semantics, then L' is at least as expressive as L.

Definition 4.10. *A semantics-preserving translation T from a language L to a language L' is a total function $T : syn(L) \to syn(L')$ such that $[\![T(l)]\!]_{L'} = [\![l]\!]_L$ for any $l \in L$.*

Lemma 4.11. *If there exists a semantics-preserving translation from L to L', then L' is more expressive than L.*

Succinctness is a property of such a translation. It measures the factor by which the translation increases (or decreases) the size of the model. It is given as a function over the natural numbers which returns the size of the output model in function of the size of the input model (denoted by the variable x). Succinctness can also be given as a set of such functions, which is often specified with the big-O notation. E.g., a succinctness of $O(x^2)$ means that the translation increases the number of elements of the input model quadratically.

Definition 4.12. *The succinctness of a translation* $T : syn(L) \to syn(L')$ *is* S*, with* $S \subseteq \mathbb{N} \to \mathbb{N}$*, iff* $\exists s \in S \bullet \forall n \in \mathbb{N}, l \in syn(L) \bullet |l| \leq n \Rightarrow |T(l)| \leq s(n)$*. A language* L *is* S*-as succinct as* L' *iff there is a translation of succinctness* S*.*

Succinctness thus indirectly measures the ability of a language to represent a subject (i.e., an element the semantic domain) concisely. As the computational complexity of most analysis algorithms is related to the size of the model, a more concise model (everything else being equal) can be analysed more efficiently.

Finally, to be able to compare sizes of models, we need to define what the size of a transition system or an FTS is.

Definition 4.13. *The size of a transition system* t *is* $|t| \triangleq |S| + |trans|$*.*

Definition 4.14. *The size of an FTS* fts *is* $|fts| \triangleq |S| + |trans| + |expr| + |d|$*. Let* $n \triangleq |N|$*, the number of features.* $|expr| \triangleq \sum_{t \in trans} |\gamma(t)|$ *gives the size of all feature expressions, which is bounded by* $O(2^n|trans|)$*.* $|d|$ *gives the size of an FD,* $|d| \triangleq |N| + |\Phi|$*, where* $|\Phi|$ *gives the size of the additional constraint, which is bounded by* $O(2^n)$*.*

Unless otherwise stated, the size of a transition-system-like model is defined in the same way as the size of a transition system.

4.4.2 Comparing FTS and transition systems

The semantic domain of FTS and transition systems is the same: the set of all sets of sequences of states. This definition is somewhat inconvenient for the purpose of this discussion, mainly because states in two different models are not directly related. The semantic information exploited by algorithms lies in the labelling of states with atomic propositions. We therefore consider two sequences of states $s_1 s_2 \ldots$ and $s'_1 s'_2 \ldots$ equivalent if the respective sequences of labelings are equal, i.e., if $\forall i \geq 1 \bullet L(s_i) = L'(s'_i)$. By extension, the semantic domain can then be considered to be the power set of the set of all sequences of labelings. A translation is semantics-preserving if for each execution of the input model, the output model has an equivalent execution, and the other way round. This corresponds to the notation *trace equivalence* [Baier and Katoen, 2008]. Given this, a first observation is the following immediate result.

Theorem 4.15. *There is a semantics-preserving translation from transition systems to FTS of succinctness* $O(x)$*.*

Proof. The FTS corresponding to a transition system has an FD that just consists of a root r, and all transitions are labeled with r. It has one product, whose projection is syntactically equivalent to the transition system. By Definition 4.3, it is thus semantically equivalent. The translation does not add any transitions or states. □

As expected, FTS are as expressive as transition systems. The two following results are more interesting.

Theorem 4.16. *There is no semantics-preserving translation from FTS to transition systems with succinctness lower than $O(2^n x)$, where n is the number of features.*

Proof. Consider the class of FTS as shown in Figure 4.3(a). Its parameter is n, the number of features. For each feature f_i, the FTS contains two transitions labelled with f_i and $\neg f_i$ respectively. The first transition leads to a state labelled with the atomic proposition iT, the second to a state labelled with iF. From those states, transitions labeled with 1 lead to a state in which the same pattern is either repeated for the next feature, or a transition leads back to the initial state. The FD of these FTS is so that all feature combinations are valid products. The size of these FTS is $7n + 2$.

The semantics of such an FTS is a set of 2^n executions each corresponding to a single product, and capturing the truth assignment of the features of this product. For example, one execution consists in indefinitely repeating

$$\{\} \to \{1T\} \to \{\} \to \{2T\} \to \ldots \to \{\} \to \{nT\} \to \{\},$$

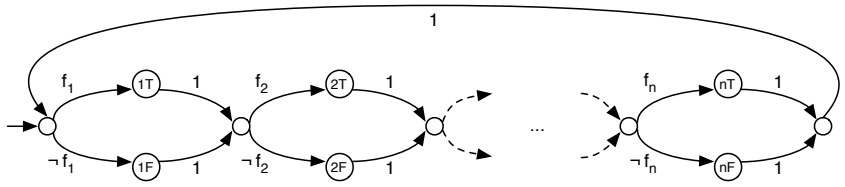
which corresponds to the product containing all features. It is impossible to have a semantically equivalent transition system with less than $O(2^n x)$ states and transitions. The only way to guarantee that each behaviour of the transition system repeats in the same truth-value pattern is to have 2^n loops with $2n + 1$ states each, as shown in Figure 4.3(b). There cannot be sharing between the loops, as there would have to be a state with two outgoing transitions, whose non-deterministic choice would allow behaviours that are not patterns of repeated truth assignments and thus not behaviours of the FTS.

Hence, no semantics-preserving translation from FTS to transition systems can have a lower succinctness than $O(2^n x)$. □

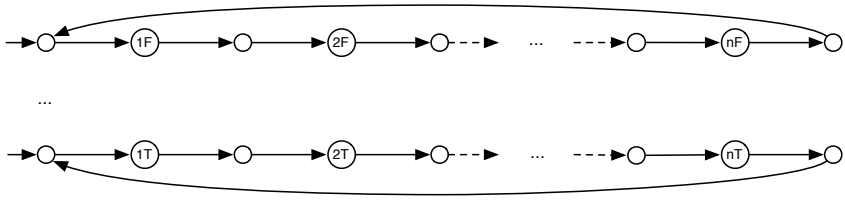
Theorem 4.17. *There is a semantics-preserving translation from FTS to transition systems of succinctness $O(2^n x)$, where n is the number of features.*

Proof. Create a transition system which contains the states and transitions of every projected transition system for every product. The set of initial states is the union of the sets of initial states of the projected transition systems. By construction, it will have the same behaviours as the FTS. Its size is $O(2^n)$ times the states and transitions of the FTS. This corresponds to the translation used in the proof of Theorem 4.16. □

The results of Theorems 4.16 and 4.17 are important, as they show that reducing FTS model checking to classical model checking of transition systems will be very costly. In the worst case, the transition system equivalent of an FTS is exponentially larger than the FTS. Combined with the algorithmic complexity of LTL and CTL model checking on transition systems given in Section 2.3, this yields lower bounds for the complexity of LTL and CTL model checking on FTS.



(a) A class of FTS. Action labels were omitted; all transition labels are feature expressions and all state labels are atomic propositions.



(b) The corresponding class of smallest equivalent transition systems.

Figure 4.3: The relative succinctness of a class of FTS and transition systems with parameter n, the number of features.

4.4.3 Comparing FTS and modal transition systems

One of the earliest attempts to modelling variability in behaviour are modal transition systems (MTS) [Larsen and Thomsen, 1988] or modal I/O automata [Larsen et al., 2007]. In these models, transitions can be mandatory (*required* transitions) or optional (*allowed* transitions). As expected, the approaches proposing the use of these models for SPL behaviour use *allowed* transitions to model variability [Fischbein et al., 2006, Larsen et al., 2007, Fantechi and Gnesi, 2007, Fantechi and Gnesi, 2008, Lauenroth et al., 2009, Asirelli et al., 2009, Asirelli et al., 2010b, Asirelli et al., 2010a]. The following definition is based on [Larsen and Thomsen, 1988].

Definition 4.18. *A model transition system is a tuple* $(S, Act, \rightarrow_{\Box}, \rightarrow_{\Diamond})$, *where*

- S *is a set of states,*
- Act *is a set of actions,*
- $\rightarrow_{\Diamond} \subseteq S \times Act \times S$ *is a set of* allowed *transitions,*
- $\rightarrow_{\Box} \subseteq \rightarrow_{\Diamond}$ *is a set of* required *transitions,*

A refinement relation can be defined for MTSs, so that an MTS m' refines an MTS m if it allows only transitions that were allowed in m, and if all mandatory transitions of m are mandatory. Note that refinement in Larsen *et al.* [Larsen and Thomsen, 1988, Larsen, 1989, Larsen et al., 2007] and also [Fischbein et al., 2006] is defined at the level of executions, whereas Fantechi and Asirelli *et*

al. [Fantechi and Gnesi, 2007, Fantechi and Gnesi, 2008, Asirelli et al., 2009, Asirelli et al., 2010b, Asirelli et al., 2010a] also define it syntactically. An *implementation* of an MTS is a refinement in which all allowed transitions are required, that is, essentially a transition system. An MTS thus specifies a (possibly infinite) family of transition systems, those that implement it.

The main distinction between FTS and MTS is that the *allowed* transitions in an MTS are all independently optional. In an FTS they are not. Consider the vending machine example from Figure 4.2. The transitions ① $\xrightarrow{free}$ ③, ① $\xrightarrow{pay}$ ② and ⑦ $\xrightarrow{open}$ ⑧, ⑦ $\xrightarrow{take}$ ① are not independent, on the contrary, the transition taken in state ① determines the transition to be taken in ⑦. In an FTS, the choice of a transition depends on the features (and hence the transitions taken before), whereas in an MTS the choice of taking an *allowed* transition is non-deterministic each time it comes up. This means that the succinctness argument from the proof of Theorem 4.16 can be made for MTS, too. A formal comparison of the expressiveness of FTS and MTS, however, cannot be done. They would need to have the same semantic domain, which is not the case. FTS always represent finite sets of transition systems, whereas the set of transition systems that an MTS represents is possibly infinite.

Another distinction between FTS and MTS, of course, is the lack of features or an FD in MTS. While the MTS does specify a family of transition systems, it offers no means to record the product to which a given transition system belongs. For this, for example, an MTS would need to have information about which features influence which *allowed* transitions. This is a key limitation. As a consequence, when the model check of a property on an MTS fails, it cannot reveal anything about the problematic products. In terms of variability, MTS do not carry more information than transition systems.

The most common use of MTS and other formalisms with *allowed/required* modalities is to abstract large systems or systems with infinitely many states. There, the modalities are used for over-approximating (allowed) and under-approximating (required) the state space [Wei et al., 2009]. This problem is fundamentally different from SPL model checking. In fact, it is an orthogonal problem, and we could imagine to extend FTS with *allowed/required* modalities.

Almost all model checking approaches for MTS and similar formalisms target the case in which the modalities are used to abstract large or infinite state spaces. They are generally three-valued model checking approaches, meaning that they can return 'unknown'. (In the case of an SPL, such a result would mean that it is unknown whether there are products that violate a property.) Model checking of MTS is discussed in [Godefroid et al., 2001]. It can be reduced to model checking two particular implementations of the MTS, in a method similar to the one used in generalised model checking [Bruns and Godefroid, 2000], discussed in Section 2.4.2. An example of a model checker for MTS is MTSA [D'Ippolito et al., 2008], which allows to check MTS against fluent LTL [Giannakopoulou and Magee, 2003].

The original papers studying MTS in the context of SPLs did not consider model checking. Instead, they focussed on refinement and related notions [Fis-

chbein et al., 2006, Larsen et al., 2007]. There are alternate model checking techniques, in which a temporal logic is interpreted directly over the MTS. Larsen [Larsen, 1989] originally proposed Hennessy-Milner logic [Hennessy and Milner, 1980], Asirelli *et al.* [Asirelli et al., 2010a] propose a variation thereof called MHML.

4.4.4 Different variants of FTS

When we initially proposed FTS in [Classen et al., 2010b], they did not have feature expressions on transitions. In [Classen et al., 2010b], the situation in which a feature f causes a transition t to be replaced by t' was modelled by labelling t with f and specifying that t' *has priority over* t. In the red lights example of Figure 4.1(b), transition ① $\xrightarrow{yellow}$ ④ would have priority over ① $\xrightarrow{red}$ ②. The FTS as defined in Definition 4.1 were only introduced later, in [Classen et al., 2011c], under the name 'FTS$^+$'. FTS$^+$ dropped priorities in favour of feature expressions which allow for more flexibility and simplify the notation. As we shall see later, both formalisms are expressively equivalent; we thus renamed 'FTS$^+$' back to 'FTS'.

To avoid ambiguity, we use the following generic definition, GenFTS.

Definition 4.19. *GenFTS(L), where* $L \subseteq \{FD, Exp, Prior\}$ *is a parameter, denotes the language of tuples* $(S, Act, trans, I, AP, L, d, \gamma, >)$ *where*

- $(S, Act, trans, I, AP, L)$ *is a transition system as defined in Definition 2.3,*
- d *is an FD as defined in Definition 1.1; depending on the 'FD' parameter:*
  - *if* $FD \in L$*, then* d *is an arbitrary FD,*
  - *if* $FD \notin L$*, then* $[\![d]\!]_{FD} = \{p \in \mathcal{P}(N) \mid r \in p\}$*, i.e., the FD allows all combinations, as if there were no FD at all but just a list of features,*[1]
- $\gamma : trans \to \mathbb{B}(N)$ *is a total function, labelling transitions with feature expressions; depending on the 'Exp' parameter:*
  - *if* $Exp \in L$*, then arbitrary feature expressions are allowed,*
  - *if* $Exp \notin L$*, then feature expressions have to be single positive literals, i.e., transitions are labelled with single features,*
- $> \subseteq trans \times trans$ *is a partial order, defining priorities between transitions; depending on the 'Prior' parameter:*
  - *if* $Prior \in L$*, there is no restriction on* $>$*,*
  - *if* $Prior \notin L$*, then* $>$ *must be empty, i.e., no priorities are allowed.*

[1] We cannot write $\mathcal{P}(N)$ instead of $\{p \in \mathcal{P}(N) \mid r \in p\}$. Following Definition 1.1, it is impossible to construct an FD in which r, the root feature, does not appear in every product.

Definition 4.20. *The size of a GenFTS fts of any type is $|fts| \triangleq |fts'| + |>|$ where fts' is the FTS obtained by removing the $>$ element from the tuple.*

With this generic definition, we can study which parts of the definition contribute to expressiveness or succinctness. FTS as defined in this thesis (Definition 4.1) and in [Classen et al., 2011c] correspond to GenFTS(FD, Exp), whereas those defined in [Classen et al., 2010b] correspond to GenFTS(FD, Prior).

Projection for the languages in GenFTS has to take priorities into account. It can be defined as follows.

Definition 4.21. *Given a GenFTS fts of any type, its projection to a product $p \in [\![d]\!]_{FD}$, noted $fts_{|p}$, is the transition system $(S, Act, trans', I, AP, L)$ where*

$$trans' = \big\{(s,\alpha,s') \mid s \xrightarrow{\alpha} s' \in trans \wedge p \in [\![\gamma(s \xrightarrow{\alpha} s')]\!] \\ \wedge \quad (\nexists s \xrightarrow{\alpha'} s'' \in trans \bullet p \in [\![\gamma(s \xrightarrow{\alpha'} s'')]\!] \\ \wedge\ s \xrightarrow{\alpha'} s'' > s \xrightarrow{\alpha} s')\big\}.$$

The definition states that a transition is removed if (i) its feature expression is not satisfied by the product, or (ii) there is a higher-priority transition of which the feature expression is also satisfied. Except for the different projection operation, the semantics of GenFTS is as for FTS (i.e., Definition 4.3 applies).

Let us start with the straightforward observation that for each $L, L' \subseteq \{FD, Exp, Prior\}$, $L \subseteq L'$ implies that a model of GenFTS(L) is also a model of GenFTS(L'). That is, models from languages with less constructs can always be translated into models of languages with more constructs. In terms of languages, GenFTS(L) is a *subset of* GenFTS(L').

A first interesting observation is that priorities do not increase expressiveness or succinctness when feature expressions are allowed.

Theorem 4.22. *Any GenFTS(FD, Exp, Prior) can be translated into a GenFTS(FD, Exp) by a translation of succinctness $O(x)$.*

Proof. Let $fts = (S, Act, trans, I, AP, L, d, \gamma, >)$ be a GenFTS(FD, Exp, Prior) and $fts' = (S, Act, trans, I, AP, L, d, \gamma', \emptyset)$ with γ' such that

$$\forall s \xrightarrow{\alpha} s' \in trans \bullet \gamma'(s \xrightarrow{\alpha} s') = \gamma(s \xrightarrow{\alpha} s') \wedge \\ \neg\big(\bigvee_{s \xrightarrow{\alpha'} s'' \in trans \bullet\ s \xrightarrow{\alpha'} s'' > s \xrightarrow{\alpha} s'} \gamma(s \xrightarrow{\alpha'} s'')\big)$$

is a GenFTS(FD, Exp). For all $p \in \mathcal{P}(N)$ and $s \xrightarrow{\alpha'} s' \in trans$, γ' is so that

$$p \in [\![\gamma'(s \xrightarrow{\alpha'} s')]\!] \implies \nexists s \xrightarrow{\alpha'} s'' \in trans \bullet p \in [\![\gamma(s \xrightarrow{\alpha'} s'')]\!] \wedge s \xrightarrow{\alpha'} s'' > s \xrightarrow{\alpha} s'$$

which is equivalent to the condition on priorities in Definition 4.21. The translation thus preserves the semantics: $[\![fts]\!]_{FTS} = [\![fts']\!]_{FTS}$. The translation does not affect the size because data is just moved from $>$ to γ. □

Due to the subset relation discussed before, this result implies that any GenFTS(FD, Prior) can be translated into a GenFTS(FD, Exp). Our decision to change the definition of FTS in [Classen et al., 2011c] was thus without loss of generality, expressiveness or succinctness. For example, the GenFTS(FD, Prior) in Figure 4.4 is equivalent to the GenFTS(FD, Exp) introduced at the beginning, in Figure 4.2.

The two following observations establish the mutual redundancy of FDs and features expression in GenFTS. The first observation is that any GenFTS(FD, Exp) can be translated into a GenFTS(Exp). That is, the FD can be omitted without loss of expressiveness.

Theorem 4.23. *Any GenFTS(FD, Exp) can be translated into a GenFTS(Exp) by a translation of succinctness $O(x+2^n|trans|)$, where trans is the transition relation of the first fts and n the number of features.*

Proof. Let $fts = (S, Act, trans, I, AP, L, d, \gamma, \emptyset)$ be a GenFTS(FD, Exp). An equivalent GenFTS(Exp) is given by $fts' = (S, Act, trans, I, AP, L, d', \gamma', \emptyset)$. Let d' be so that $[\![d']\!]_{FD} = \{p \in \mathcal{P}(N) \mid r \in p\}$. The feature expression of each transition is conjoined with $\mathbb{B}(d)$, the Boolean encoding of the FD: $\forall t \in trans$, $\gamma'(t) = \gamma(t) \wedge \mathbb{B}(d)$. Although fts' admits more products, the condition added to all transitions means that the projections of products invalid in d do not have transitions and thus a singleton state. Hence $[\![fts]\!]_{FTS} = [\![fts']\!]_{FTS}$. □

The second observation is that any GenFTS(FD, Exp) can be translated into a GenFTS(FD). In other words, restricting feature expressions to single features (i.e., consisting of a single positive literal) does also not affect expressiveness.

Theorem 4.24. *Any GenFTS(FD, Exp) can be translated into a GenFTS(FD) by a translation of succinctness $O(x + |trans|)$, where trans is the transition relation of the first fts.*

Proof. Let $fts = (S, Act, trans, I, AP, L, d, \gamma, \emptyset)$ be a GenFTS(FD, Exp). An equivalent GenFTS(FD) is given by $fts' = (S, Act, trans, I, AP, L, d', \gamma', \emptyset)$. Let $d' = d$. For each transition $t \in trans$, a new feature x_t is added to

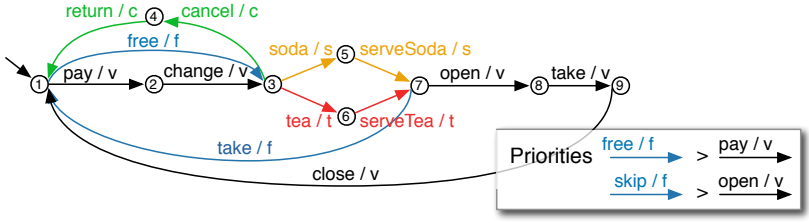


Figure 4.4: GenFTS(FD, Prior) of the vending machine SPL.

d' so that $[\![d]\!] = [\![d']\!]$, t is labeled by x_t: $\gamma'(t) = x_t$, and a constraint $x_t \Leftrightarrow \gamma(t)$ is added to d'. By construction, $[\![fts]\!]_{FTS} = [\![fts']\!]_{FTS}$. □

Note the regularity in these three proofs. The proof of Theorem 4.22 gives a method for eliminating priorities, by incorporating their information into the feature expressions. The proof of Theorem 4.23 gives a method for eliminating the FD, again by incorporating its information into the feature expressions. Finally, Theorem 4.24 gives a method for eliminating feature expressions by encoding them with new features in the FD.

Those steps can be reused and chained, leading to the following corollaries.

Corollary 4.25. *Any GenFTS(FD, Prior) can be translated into GenFTS(FD) by a translation of succinctness* $O(x + |trans|)$.

Proof. Transform to GenFTS(FD, Exp) following Theorem 4.22, and then into GenFTS(FD) following Theorem 4.24. □

Corollary 4.26. *Any GenFTS(FD, Exp) can be translated into a GenFTS(FD, Prior) by a translation of succinctness* $O(x + |trans|)$.

Proof. Transform to GenFTS(FD) following Theorem 4.24, which is a GenFTS(FD, Prior) by language inclusion. □

A question which remains open is whether it is possible to translate feature expressions to priorities, and how it can be accomplished. However, what can be shown is that the translation from GenFTS(Prior) to transition systems causes at least the same exponential blowup as for the other kind of GenFTS. More generally, the exponential blowup is incurred for any kind of GenFTS. The following result thus generalises Theorem 4.16.

Theorem 4.27. *There is no semantics-preserving translation from any GenFTS to transition systems with succinctness lower than* $O(2^n x)$*, where* n *is the number of features.*

Proof. For GenFTS(Exp) and all related variants, the proof of Theorem 4.16 applies immediately. For the other GenFTS(FD) and GenFTS(Prior), a class of models similar to the one used in the proof of Theorem 4.16 can be constructed.

For GenFTS(FD), consider a class of models in which the FD has $2n$ features, each feature f_i is paired with a feature f'_i and a constraint $f_i \Leftrightarrow \neg f'_i$. The FD thus has 2^n products despite having $2n$ features. The states and transitions of this class of models are like those in Figure 4.3(a). The $\neg f_i$ labels are replaced by f'_i, to conform to the restriction that transitions can only be labelled by positive literals. This class of models only contains GenFTS(FD) and is semantically equivalent to the one used in Figure 4.3(a). From here on, the proof of Theorem 4.16 can thus be followed.

For GenFTS(Prior), consider the class of models shown in Figure 4.5. The priorities between transitions are expressed graphically with a bold 'v' (i.e., $>$

in top-down direction). The r in this model refers to the root feature. The transition combination of $s \rightarrow s'$ labelled with f_1, $s \rightarrow s''$ labelled with r, and $s \rightarrow s' > s \rightarrow s''$ corresponds to the feature expressions $\gamma(s \rightarrow s') = f_1$ and $\gamma(s \rightarrow s'') = r \wedge \neg f_1$. Since the root feature is mandatory and thus always 1, the latter annotation is equivalent to $\neg f_1$. From here on, the proof of Theorem 4.16 can be followed, as the family of models used in this proof is semantically equivalent to the class of GenFTS(Prior) given in Figure 4.5. □

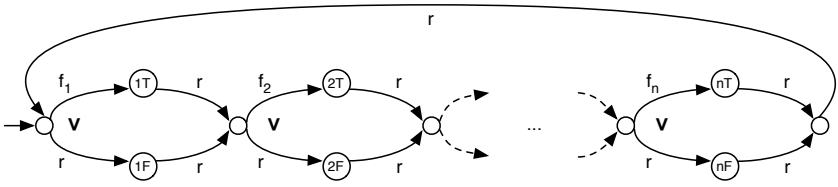


Figure 4.5: A class of GenFTS(Prior). The parameter n denotes the number of features; action labels were omitted; all transition labels are features and all state labels are atomic propositions; bold 'v's indicate priority.

4.4.5 Discussion and summary

Figure 4.6 gives an overview of the expressiveness and succinctness results obtained in this section.

An important result of this section is that FTS are exponentially more succinct than transition systems, and that this succinctness can be attributed to priorities, feature expressions and FDs alike. Furthermore, all three types of construct can be translated to each other (except for translations to priorities), which means that they can be considered as syntactic sugar for each other. However, as the overview in Figure 4.6 demonstrates, the most succinct way of specifying SPL behaviour is with GenFTS(FD, Exp), which corresponds to the definition of FTS used in the thesis.

While it is possible to argue about the utility of language constructs, it is often difficult to provide formal evidence. For example, in the past [Classen et al., 2011c], we argued for feature expressions because they allow for greater expressiveness, more intuitive models and simpler definitions. While the last two points might be correct, the first was just shown to be false. Feature expressions do not add expressiveness. They increase succinctness.

Similarly, in [Classen et al., 2011d] we remarked that both, priorities and feature expressions, can be encoded as part of the FD. We argued that this is not very intuitive and requires the introduction of a dummy feature for each feature expression. Moreover, it becomes hard to interpret the FTS without the FD. Again, these arguments rely more on intuition than evidence. In the previous section, we showed that feature expressions do increase succinctness.

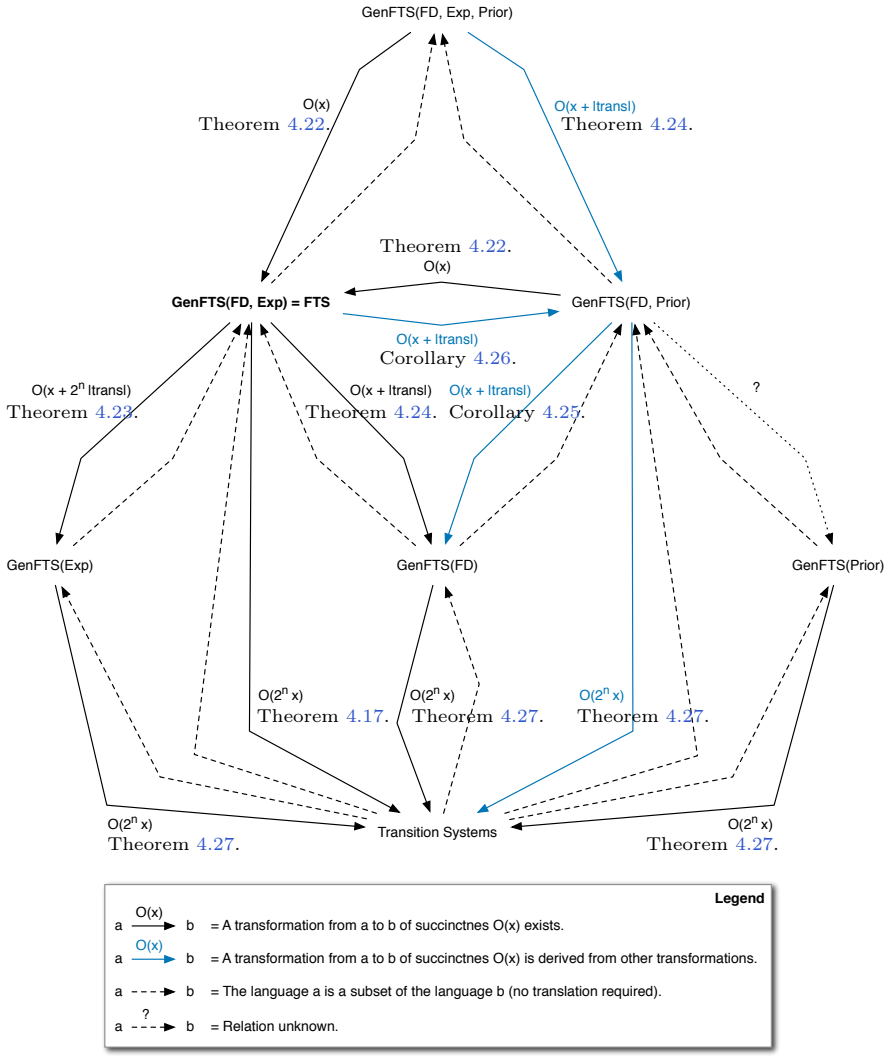


Figure 4.6: Overview of the expressiveness and succinctness results.

Priorities can be added to FTS without impacting succinctness. We can thus only argue that they might help to make models more intuitive and understandable. If a transition overrides another transition, this is more apparent when expressed with priorities than with feature expressions. A disadvantage of priorities, however, is that parallel composition as specified in Definition 4.7 requires conjunction of feature expressions. It cannot be easily defined for priorities, except for the case when the priority relation of one FTS is empty.

4.5 Examples

Before we conclude this chapter, we present two more illustrative examples.

4.5.1 The wiper system

The car wiper system example was proposed in [Gruler et al., 2008b]. It consists of two subsystems: a sensor unit, able to detect rain, and the wiper itself. Both the sensor and the wiper come in two qualities, high and low. A low quality rain sensor can only distinguish between *rain* and *no rain*, whereas the high quality sensor can also discriminate between *heavy* and *little rain*. Similarly, the high quality wipers can operate at two speeds, whereas the low quality wiper only operates at one speed. In addition, the low quality wipers can be set to wipe permanently. The FD in Figure 4.7 models this situation.

Gruler *et al.* propose the process algebra PL-CCS [Gruler et al., 2008b], a variant of Milner's *Calculus of Communicating Systems* (CCS) [Milner, 1980] to which a new operator $\oplus$ was added to represent alternative choice between two processes. The whole wiper system is modelled with the PL-CCS expression $WipFam$ in Figure 4.8. It is the parallel composition of the sensor and the wiper subsystem. The sensor subsystem is defined as being either the low or the high quality sensor subsystem. The wiper subsystem is defined similarly.

The PL-CCS definition of the two sensor subsystems is given in Figure 4.9. The low quality sensor will either sense no rain, or it will sense heavy/little rain in which case it sends the message *Rain*. As expected, the high quality sensor

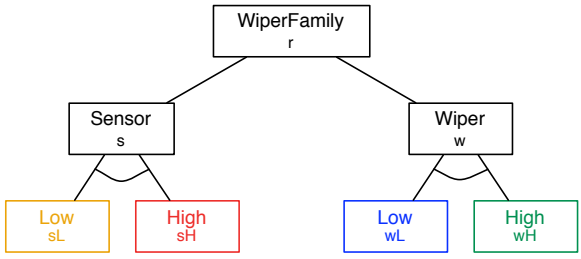


Figure 4.7: The FD for the wiper system.

$$
\begin{aligned}
WipFam &\stackrel{def}{=} Sensor \parallel Wiper \\
Sensor &\stackrel{def}{=} SensL \oplus_1 SensH \\
Wiper &\stackrel{def}{=} WipL \oplus_2 WipH
\end{aligned}
$$

Figure 4.8: The wiper system in PL-CCS, taken from [Gruler et al., 2008b].

Low quality.

$$
\begin{aligned}
SensL &\stackrel{def}{=} non.SensL + ltl.Raining + hvy.Raining + \overline{noRain}.SensL \\
Raining &\stackrel{def}{=} non.SensL + ltl.Raining + hvy.Raining + \overline{rain}.Raining
\end{aligned}
$$

High quality.

$$
\begin{aligned}
SensH &\stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{noRain}.SensH \\
Medium &\stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{rain}.Medium \\
Heavy &\stackrel{def}{=} non.SensH + ltl.Medium + hvy.Heavy + \overline{hvyRain}.Heavy
\end{aligned}
$$

Figure 4.9: The sensor subsystem in PL-CCS, taken from [Gruler et al., 2008b].

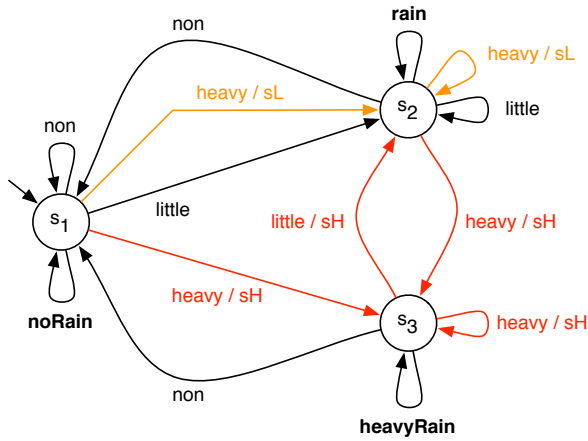


Figure 4.10: The FTS for the sensor subsystem.

Low quality.

$$
\begin{aligned}
WipL &\stackrel{def}{=} off.WipL + manualOn.Permanent + intvOn.Interval \\
Interval &\stackrel{def}{=} noRain.Interval + intvOff.WipL + intvOn.Interval \\
&\quad + rain.Wiping + hvyRain.Wiping \\
Wiping &\stackrel{def}{=} \overline{slowWip}.Interval + intvOn.Interval \\
Permanent &\stackrel{def}{=} \overline{permWip}.Permanent + off.WipL + intvOn.Interval
\end{aligned}
$$

High quality.

$$
\begin{aligned}
WipH &\stackrel{def}{=} off.WipH + intvOn.AutoIntv \\
AutoIntv &\stackrel{def}{=} noRain.AutoIntv + intvOn.AutoIntv + rain.Slow \\
&\quad + intvOff.WipH + hvyRain.Fast \\
Slow &\stackrel{def}{=} \overline{slowWip}.AutoIntv + intvOn.AutoIntv \\
Fast &\stackrel{def}{=} \overline{fastWip}.AutoIntv + intvOn.AutoIntv
\end{aligned}
$$

Figure 4.11: The wiper subsystem in PL-CCS, taken from [Gruler et al., 2008b].

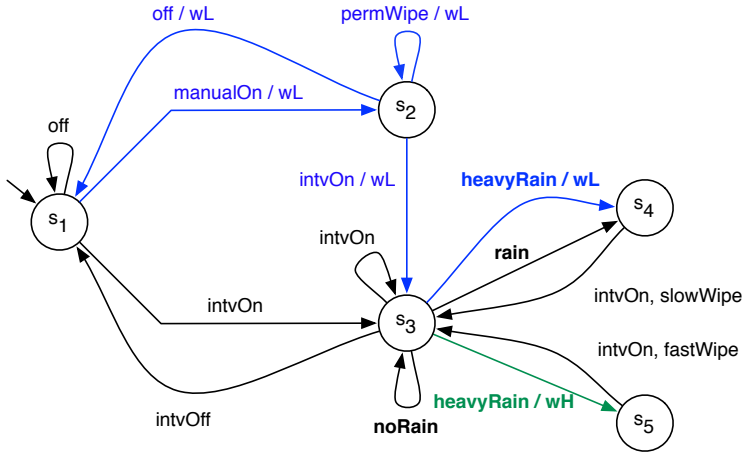


Figure 4.12: The FTS for the wiper subsystem.

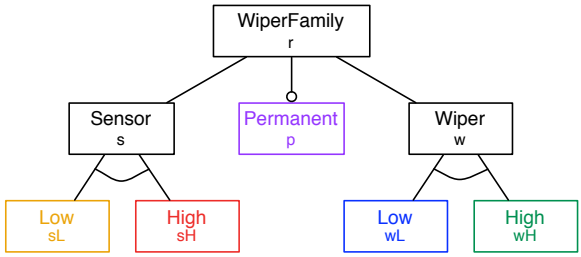


Figure 4.13: A FD for the wiper system in which permanent wiping is explicitly represented as a feature.

behaves differently: in case of heavy rain it sends the message $HvyRain$. An immediate observation is that both subsystems are quite similar and that the sending of the $Rain$ message is the same in both cases. Still, the corresponding part has to be duplicated inside both subsystems. An equivalent description in FTS is given in Figure 4.10. As before, the colour of a transition matches its feature label; if colours are unavailable on the medium being used, the feature label is always written behind the action label, separated by a slash. Transitions with no feature label are labelled implicitly by $true$. Since the part dealing with the detection of little or no rain is the same for both qualities, the corresponding actions in the FTS are part of the base system instead of being duplicated. Both features only differ visibly in the handling of the heavy rain condition. Note that in Figure 4.10 and subsequent figures, labels in bold font highlight transitions which are synchronised in a parallel composition.

As to the two wiper subsystems, their PL-CCS definition is given in Figure 4.11. Both subsystems have an interval switch, which will switch interval wiping on. During interval wiping, both subsystems wipe if the sensor subsystem reports rain; the high quality subsystem will wipe faster in case of heavy rain. In addition, the low quality wiper can be set to permanent wiping, which ignores the rain sensor. Here again, both subsystems are almost identical (except for the permanent wiping function), the sole difference being that the high quality variant reacts differently to a $HvyRain$ message. As a consequence, the definitions for both subsystems are almost duplicates. This duplication is not needed in FTS. Consider the FTS representation of the wiper subsystem shown in Figure 4.12. Note that two action labels on a transition, e.g., $intvOn, slowWipe$, is a shorthand notation for two transitions. The FTS in Figure 4.12 clearly shows that (except for the permanent wiping function) the high and low quality subsystems only differ in their handling of $HeavyRain$.

We conclude this example with an extension that is not part of the original paper [Gruler et al., 2008b]. Consider the case in which the permanent wiping feature can also be supported by high quality wipers (in fact, there is no reason why it should not). That is, permanent wiping will become an individual

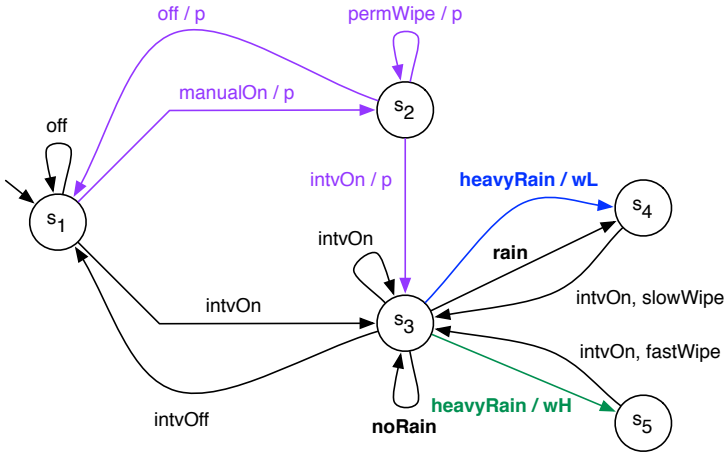


Figure 4.14: The modified FTS for the wiper system with permanent wiping as a separate feature.

feature which is optional. A revised FD that accommodates the new feature is presented in Figure 4.13. To make the corresponding change in the behavioural model, in FTS it is sufficient to relabel the transitions pertaining to the wiping feature, as shown in Figure 4.14. In PL-CCS, on the other hand, one will have to duplicate the definition of the permanent wiping mode in both subsystems.

All these models are distributed with the Haskell FTS library [Classen, 2010b], our first FTS implementation, briefly discussed in Part III.

4.5.2 The mine pump system

The mine pump system is a specification exemplar for distributed systems originally introduced in [Kramer et al., 1983]. The purpose of the system is to keep a mine shaft clear of water while avoiding the danger of a methane explosion. It consists of a water pump, a sensor measuring the water level and a sensor measuring the concentration of methane in the mine. The system should activate the pump once the water level reaches a preset threshold, but only if the methane is below a critical limit.

The system consists of three high-level features, shown in Figure 4.15: (i) a command interface c, which can be used to switch the water regulation function on or off; (ii) a methane alarm interface m, which can receive alarm messages from the methane sensor in case of critical methane, and (iii) the water regulation subsystem l. The system is distributed: the controller and sensors are individual subsystems which communicate by message passing. Although the system was not designed as an SPL, these components play the same roles as features in an SPL and can be modelled as such.

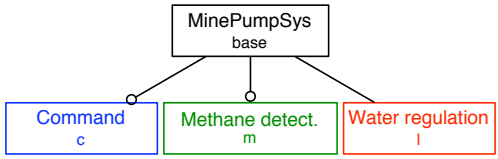


Figure 4.15: The FD for the mine pump controller.

Following the CONIC code included in [Kramer et al., 1983], we created an FTS of the mine pump system. The controller manipulates a variable representing the system state, and its reactions to events such as a methane alarm or high water depend on this state. In order to keep the model size reasonable, the controller is modelled by two FTS. The logic of the controller is modelled by the FTS in Figure 4.16 (the *system FTS*), whereas the system state is maintained by the FTS in Figure 4.17 (the *state FTS*). As before, when there are several transitions with different action labels between two states, only one transition is drawn, and the action (and feature) labels of the transitions are listed above or below the transition. In this FTS, we make use of action labels, which are specified next to a state in curly braces. In Figure 4.17 and subsequent figures, the transitions without feature labels are implicitly labelled by 1 (and coloured black). These transitions are synchronised during parallel composition and take the feature of the transition with which they are synchronised.

Basically, the system FTS describes the *actions on the system state*, but does not record it explicitly. The actual FTS of the controller is the parallel composition of the system FTS and the state FTS. Intuitively, each time the FTS executes a $set*$ action, e.g., *setReady*, it will be synchronised with the corresponding transition in the state FTS. The result is that the state in which the transition arrives receives the atomic propositions of the state FTS, e.g., *ready*. The state FTS will thus add an atomic proposition with the system state to each state of the system FTS. This causes a small blowup; the resulting FTS will have hundreds of states. There are five system states:

- *stopped* means that the water regulation function is off (controlled via the command interface). The system will avoid switching on the pump.
- *ready* means that the water regulation function is on (controlled via the command interface). The system will switch the pump on if there is no methane and the water level is high.
- *running* means that the pump is currently running.
- *lowstopped* means that the pump was stopped because the water level was low. The pump will resume in case the water rises again.
- *methanestopped* means that the pump was stopped because of a critical methane level. The pump will not resume until explicitly switched on via the command interface.

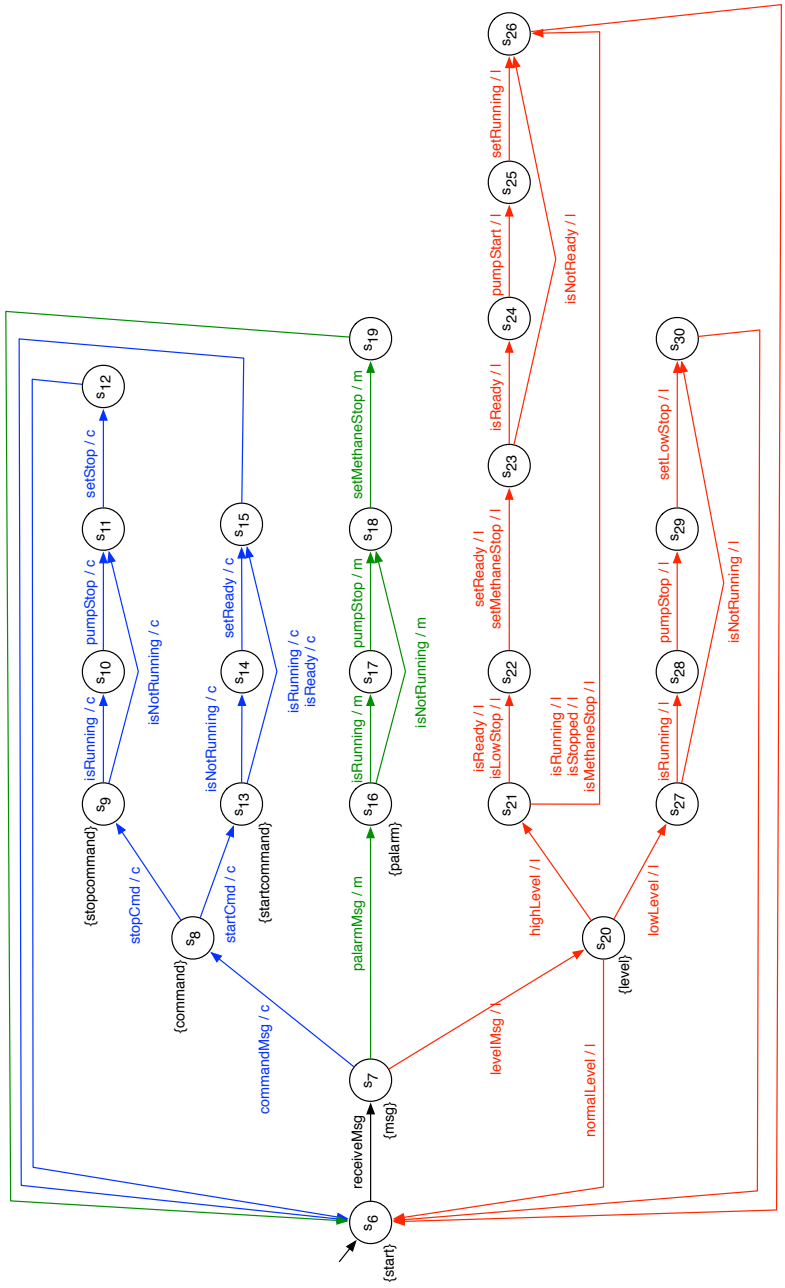


Figure 4.16: The *system* FTS of the mine pump controller.

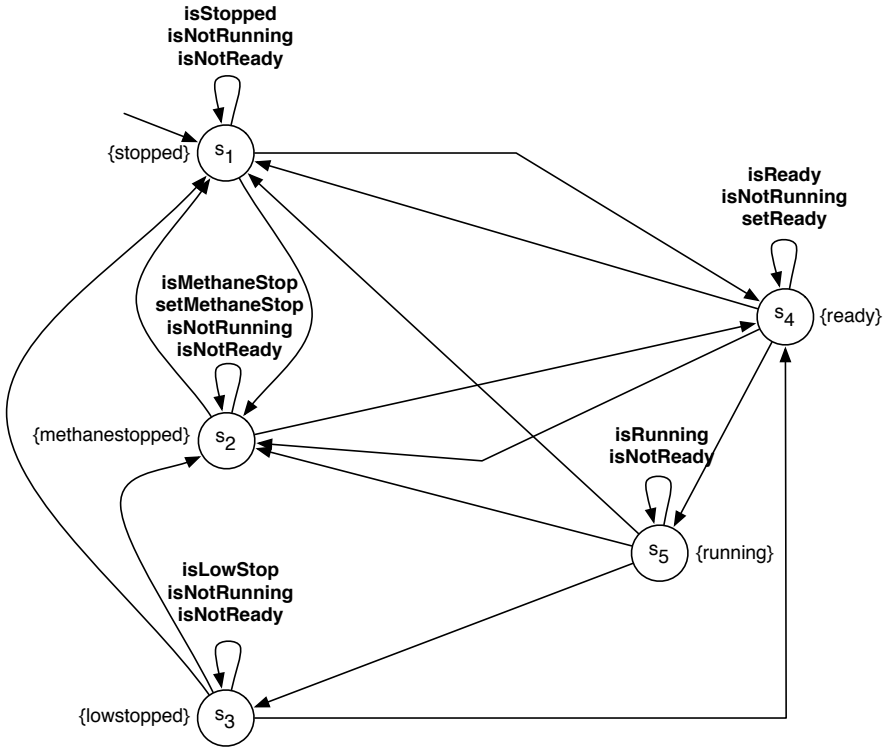


Figure 4.17: The *state* FTS of the mine pump controller. The action labels of transitions between states were omitted to avoid overload. They depend on the state in which the transition arrives. For instance, all transition arriving in s_4 have the action label *setReady*, those arriving in s_1 have *setStop*, and so on.

The system operates as follows. It will observe three types of events: commands, methane alarm messages and water readings. There are two types of command: stop and start. In case of a start command, the system state is changed and set to *ready*. In case of a stop command, the pump is stopped, and the system state set to *stopped*. In case of a methane alarm, the system stops the pump and sets the system state to *methanestopped*. The system can distinguish between three different water levels: in case of normal water, the system does nothing; if the water is high and the pump not yet running, the system will first check whether it is *ready* or whether it just stopped because of low water (*lowstopped*), if yes, it will check the methane level, and if there is no methane (that is, if after the check it is still *ready*), it will start the pump and set the state to *running*, otherwise it will do nothing. Once the water is low, the system switches off the pump and sets the system state to *lowstopped*.

The system interacts with its environment, which is modelled with three

other FTS that are put in parallel with the FTS of the system. The methane level is modelled with the FTS in Figure 4.18. Methane can rise and lower at will, represented by the *methaneRise* and *methaneLower* transitions. The $pAlarmMsg$ and $setMethaneStop$ transitions will synchronise with the system FTS, meaning that the system FTS will receive the alarm message only in case of high methane.

The water pump is represented by the FTS in Figure 4.19. The pump can be in two states, running or stopped. The actions *pumpStart* and *pumpStop*, synchronised with the system FTS, will cause this state to change. The action *pumpRunning* is used to model the interaction between pump and water. It is synchronised with the water FTS shown in Figure 4.20: a running pump will cause the water level to decrease. The level can rise at will. The *low*, *high* and *normalLevel* actions are synchronised with the main FTS, meaning the system will only observe low, high or normal water if this is indeed the case.

When the command interface and the methane alarm interface are considered optional, as in the first FD in Figure 4.15, there are four different products. We can add further variability by considering the start and stop message types as well as the three water level readings as individual features. A revised FD is shown in Figure 4.21. The product line now has 64 products. The revised system FTS is given in Figure 4.22. The other FTS do not change. These models are also distributed with the Haskell FTS library [Classen, 2010b]. We used these in [Classen et al., 2010b] to conduct the first experiments assessing the efficiency of the FTS algorithms.

4.6 Conclusion

FTS are a formalism designed to describe the combined behaviour of a whole system family. FTS are transition systems in which transitions are linked to the features of an SPL by the means of feature expressions (in addition to being labelled with actions). This allows us to model very detailed behavioural variations of the product line. In addition, features as treated as first-class abstractions, which allows both explicit variability management and separation of concerns, since a global view of the variability is available in an FD. FTS are the formal foundation for all further developments of this thesis.

FTS are exponentially more succinct than similar models, such as transition systems. However, they also require new model checking algorithms. Before we get to the model checking algorithms, we need to discuss what model checking in the context of SPLs means, and give a formal definition of the problem. We do this in the following chapter.

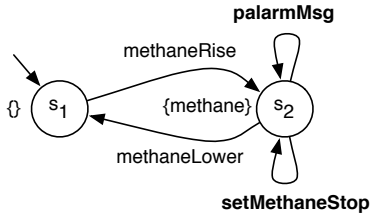


Figure 4.18: An FTS modelling the environment: the methane level.

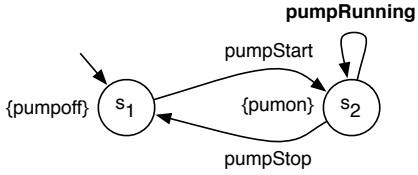


Figure 4.19: An FTS modelling the environment: the pump.

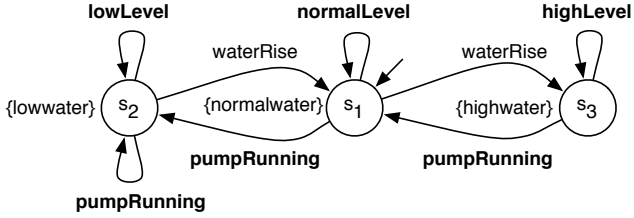


Figure 4.20: An FTS modelling the environment: the water level.

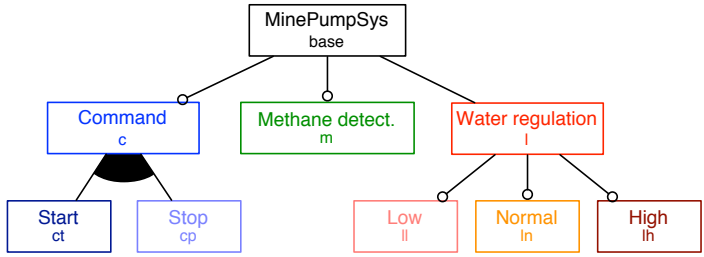


Figure 4.21: The refined FD for the mine pump system.

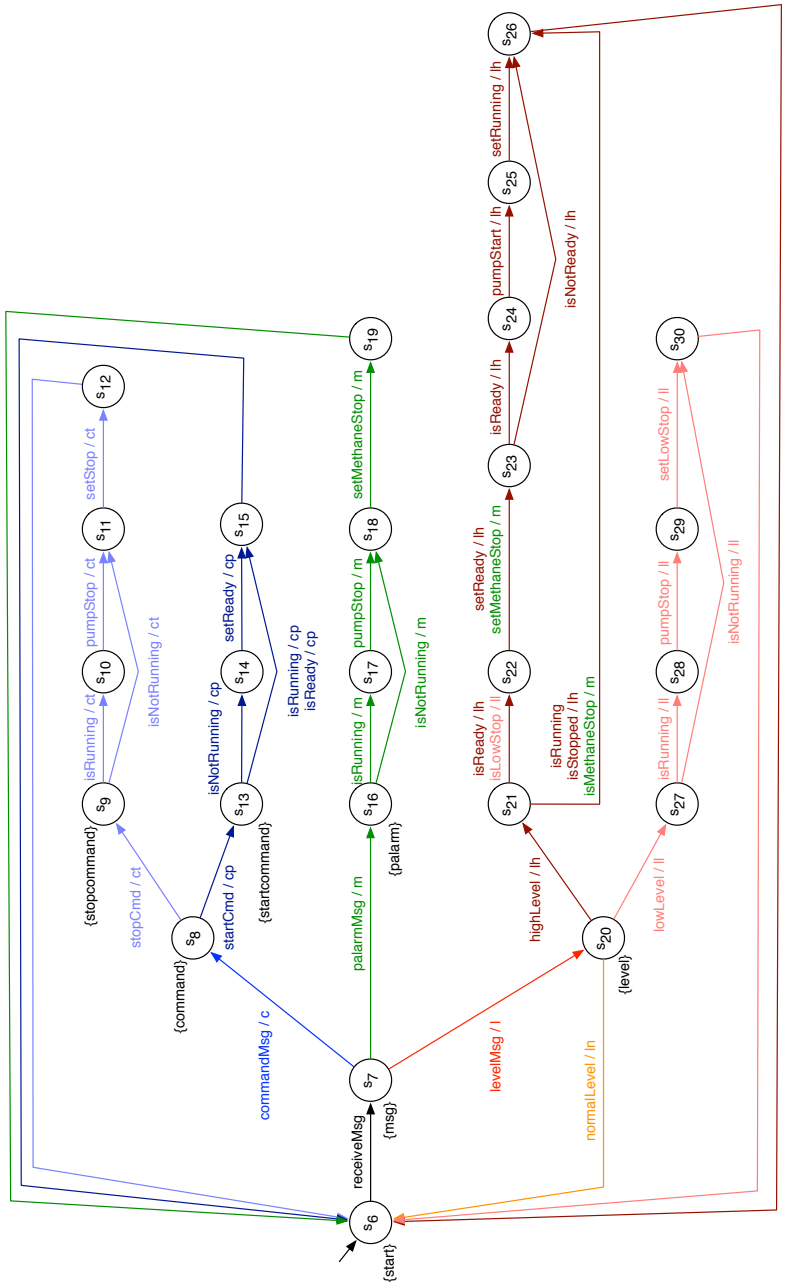


Figure 4.22: The refined FTS for the mine pump system.

Chapter 5

The Model Checking Problem

> *It is not from the benevolence of the butcher, the brewer, or the baker that we expect our dinner, but from their regard to their own interest.*
>
> Adam Smith, *The Wealth of Nations*, 1776

With FTS, we have a formalism for describing the behaviour of an SPL. Model checking is a technique for checking whether such a model exhibits certain desirable, or undesirable properties. Before we can introduce model checking techniques for FTS, however, we need to provide a formal description of the problem that these techniques are meant to solve. We also need to investigate what kinds of properties are of interest in SPLE.

We will start with a general introduction and contextualisation in Section 5.1. In Section 5.2, we then discuss what kind of temporal properties can be checked on an SPL. The model checking problem itself is formalised in Section 5.3. In Section 5.4, we discuss a number of common model checking techniques and their application in the context of SPL model checking, before concluding in Section 5.5.

5.1 Introduction

We have introduced the problem of model checking for single systems in Chapter 2. In a nutshell, the model checking problem consists in determining whether a behavioural model satisfies a given property. If the behavioural model describes just a single system, it is clear that model checking will provide information about this system.

In the case of SPLs, however, the behavioural model describes several systems (the products). The Boolean response of the model checking problem is thus insufficient. While a satisfied property (the positive answer) is, by Definition 4.3, satisfied by each product, a negative answer carries no information

about products. To be helpful, the answer should at least identify one violating product. But even this is not satisfactory. Reporting a single violating product allows no conclusion as to which features are responsible for the violation. Furthermore, it might be the case that all but a few products satisfy a property, which is hard to determine if a check just reports one violating product.

Ideally, model checking in SPLs should thus determine satisfaction or violation for all products in the SPL. Furthermore, it should present its result in terms of features. First, providing lists of violating products does not scale due the potentially huge number of products. Second, and more importantly, if features are treated as first-class concepts in the modelling language, they should be first-class concepts in decision procedures computed over the modelling language. If engineers think in terms of features when modelling a system, they most certainly think in terms of features when they verify it. Furthermore, reporting a violation in terms of features gives valuable clues as to the origin of the violation.

This leads us to the question whether and how features should be first-class concepts in the properties that are checked.

5.2 Expressing properties in fLTL and fCTL

Classical logics such as LTL and CTL can readily be used to express properties of SPLs. Such logics express properties about the existence and about the ordering of states in the executions of a system. From the previous chapter, it is clear that the choice of features may affect the existence and the ordering of states in an execution significantly. (In fact, the only effect of features.) However, features do not have to appear in these properties to make them relevant to SPL model checking. Features cannot even be used to characterise states. States are indirectly referred-to by assertions about atomic propositions.

Another way to look at this is that an SPL is a set of systems which are not fundamentally different from systems developed as single systems from the outset. The properties that one would like individual products to satisfy (deadlock freedom, respect of a critical section, request/answer patterns, and so on) are the same as those that have been subject to model checking in single systems for the last thirty years. We view model checking as orthogonal to other activities in SPLE. If model checking an SPL against a property corresponds to model checking all its products against this property, it would seem quite natural that the properties of interest to us are those that are used for single systems. And just as in single systems, we expect properties to be expressed with temporal logic formulae.

However, as a property might not be relevant to all products, a means to express the products for which a property holds should be added to temporal logics. This *quantifier* does not affect the semantics of the temporal property, but rather limits the range of products over which it holds. As an example, consider the property for the vending machine example, discussed in

Section 2.2.2: *"After selecting a beverage, the machine will always open the beverage compartment to allow the customer to collect her beverage."* The property is independent of all but the *FreeDrinks* feature, as products with this feature do not have a closing beverage compartment. That is, the property is irrelevant for products with the *FreeDrinks* feature; it should only hold for those without the feature.

To specify the quantifier, we chose to use feature expressions. They are already prevalent in all of FTS and fully expressive wrt. the set of products. This leads us to define *feature LTL* (fLTL) and *feature CTL* (fCTL) as follows.

Definition 5.1. *An fLTL (resp. fCTL) property ϕ is an expression $\phi := [\chi]\phi'$ where ϕ' is an LTL (resp. CTL) property and $\chi \in \mathbb{B}$ a feature expression. An FTS with FD d satisfies ϕ, noted $fts \models \phi$, iff*

$$\forall p \in [\![\chi]\!] \cap [\![d]\!] \bullet fts_{|p} \models \phi'.$$

That is, each product of the FD that is included in the quantification yields a transition system that satisfies the LTL (resp. CTL) property.

In fLTL, the example formula can be expressed as follows:

$$[\neg FreeDrinks]\ \Box\big(selected \Rightarrow \Diamond open\big).$$

Suppose that state ⑦ of the FTS in Figure 4.2 is labelled with the proposition *selected* and state ⑧ with the proposition *open*. It is clear that if the *freeDrinks* feature is selected, state ⑧ is never reached and the property violated. The quantifier eliminates such behaviours, and the quantified property is indeed satisfied by the FTS.

An LTL (resp. CTL) property (without quantifier) is interpreted over an FTS as the fLTL (resp. fCTL) property quantified over all products.

Definition 5.2. *For an LTL or CTL property ϕ, $fts \models \phi \triangleq fts \models [1]\phi$.*

Note that quantifiers in these logics cannot be nested. A formula can only have one quantifier at the root. The reason for this that in practice, there is no need for the nesting of such quantifiers. In this regard, our definition of fCTL differs from the one we gave in [Classen et al., 2011c] where nesting of quantifiers was allowed. Since we did not find any use for nesting, we decided to limit our logics to a single quantifier.

Recall that the logics LTL and CTL are subsets of CTL*. We do not consider CTL* here since its fragments LTL and CTL are generally treated separately. An algorithm for CTL* can be easily obtained from those for LTL and CTL [Emerson and Lei, 1987]. Nevertheless, one could define the logic *feature CTL** (fCTL*) similarly to the way fLTL and fCTL are defined in Definition 5.1. fLTL and fCTL would then be subsets of fCTL*, and a model checking algorithm for fCTL* over FTS could be obtained by combining those given for fLTL and fCTL in the following two chapters.

5.3 The model checking problem in SPLs

Following the above discussion, the SPL model checking problem is not a decision problem, where the answer would be Boolean, but a *function problem* [Papadimitriou, 1994]. It requires that a negative answer is accompanied by at least one example of a violating product.

Definition 5.3. *For a logic $L \in \{fLTL, fCTL\}$, given a property $[\chi]\phi$ in L and an FTS fts, $Mc_L(fts, [\chi]\phi)$ returns true iff $fts \models_L [\chi]\phi$. If $fts \not\models_L [\chi]\phi$, it returns $false$ and a non-empty set of products $px \subseteq [\![\chi]\!] \cap [\![d]\!]_{FD}$ such that $\forall p \in px \bullet fts_{|p} \not\models_L \phi$.*

This is analogous to classical model checking, which returns $false$ if it finds a single violating execution. There might be other executions that violate the property, but a single one is sufficient to prove the violation. In our case too, there might be other products that violate the property, but a single one is sufficient to prove that it does not hold for those specified by the quantifier.

However, because it does not say anything about the other products, such a model check is only of limited use. As said before, it does not reveal much about the features required for the violation to occur. Furthermore, if all other products satisfy the property, it might be easier to just exclude the violating product in the FD, rather than fix the problem. This leads us to propose an SPL-specific model checking problem: to determine for each product whether or not it satisfies the property.

Definition 5.4. *For a logic $L \in \{fLTL, fCTL\}$, given a property $[\chi]\phi$ and an FTS fts, $ExtMc_L(fts, [\chi]\phi)$ returns true iff $fts \models_L [\chi]\phi$. If $fts \not\models_L [\chi]\phi$, it returns $false$ and a non-empty set of products $px \subseteq [\![\chi]\!] \cap [\![d]\!]_{FD}$ such that $\forall p \in px \bullet fts_{|p} \not\models_L \phi$ and $\forall p \in ([\![\chi]\!] \cap [\![d]\!]_{FD}) \setminus px \implies fts_{|p} \models_L \phi$. To simplify the notation, we write $px_{\not\models\phi}$ (resp. $px_{\models\phi}$) to denote the set of products that violate (resp. satisfy) ϕ.*

In addition to the set of products, model checkers generally give an example of an execution that violates the property. This is crucial in practice, as it helps the engineer locate the error, or reproduce the problem in a simulation environment. For instance, the result of $Mc(fts, \Box(selected \Rightarrow \Diamond open))$, where fts denotes the FTS of Figure 4.4, would be $false$, the set $\{\{v, b, s, f, cur, eur\}, \{v, b, s, f, cur, usd\}\}$ and the counterexample:

$$① \xrightarrow{free} ③ \xrightarrow{soda} ⑤ \xrightarrow{serveSoda} ⑦ \xrightarrow{skip} ① \longrightarrow \ldots$$

Note that the decision problem consists in calculating the full set of violating products, even when these products have different counterexamples. We did not include the generation of such counterexamples in the decision problems, as this will make it easier to study their computational complexity. Especially in the case of CTL (hence fCTL), the generation of a counterexample is generally treated as a separate problem [Clarke et al., 1995].

Also note that in practice, the set of products is expected to be given as a feature expression. This is illustrated by the example we just gave, where the features *cur*, *eur* and *usd* are irrelevant, as they do not influence the behaviour. The feature expression $s \wedge f \wedge \neg t \wedge \neg c$ would expand into the same set of products, and only mentions features relevant to the property.

5.4 Practical aspects of SPL model checking

Having covered the fundamental model checking problems, we now look at more practical aspects, vacuity and deadlock detection, and how they apply to model checking of SPLs.

5.4.1 Vacuity detection

In practice, it is often necessary to check properties under certain assumptions. For example, if the system consists of several processes, it is reasonable to assume that each process gets scheduled fairly. Formally, an infinite execution should contain infinitely many steps of each process. When checking the property, only such executions should be taken into account, as others are deemed irrelevant. This is an example of a *fairness property*. Properties for the mine pump example discussed in Section 4.5.2 also have several assumptions. For instance, the system can receive different kinds of message. A reasonable assumption there is that it will read each kind of message infinitely often. Moreover, there is an assumption that the system actually does something, which weeds out executions in which only the environment changes. This is an example of a *progress assumption*.

Most of these assumptions can be specified as fairness properties. E.g., the progress assumption could be formulated as $\Box\Diamond progress$, where *progress* is an atomic proposition that holds in every state that is considered to be indicative of the system doing something. In LTL, a fairness property is itself an LTL property. This means that model checking an LTL property ϕ under the assumption ψ is equivalent to model checking the LTL property $\psi \Rightarrow \phi$. In CTL, the treatment of fairness properties is different, as they cannot be expressed in CTL. Instead, a preprocessing is done to identify states from which a fair path can leave, and the algorithm is adapted in order to restrict it to paths which only contain such states.

An important consideration when reasoning with assumptions is to avoid *vacuous satisfaction*. Assumptions have to be *discharged*; that is, there has to be at least one execution in which the assumption holds. Otherwise, any property under the assumption is trivially satisfied. Vacuous satisfaction not only applies to explicit assumptions, such as those given above. Other kinds of property can also be vacuously satisfied. A common example is *antecedent failure*, i.e., if the left-hand side of an implication does not become true in any execution, the right-hand side will never be checked and the property vacuously

satisfied [Beer et al., 2001]. A vacuously satisfied property is most likely an error in the property or the model. It is thus important to detect vacuous satisfaction and report it to the engineer.

As expected, vacuity detection in SPL model checking needs to take variability into account. In an FTS, vacuous satisfaction of a property may depend on the product. Discharging an assumption in an FTS should thus produce a set of products for which the assumption can be discharged. As for transition systems, this problem can be reduced to a model checking problem. Let us illustrate this for the case of an LTL property. In this case, the assumption ψ is discharged by model checking the system against its negation, $\neg\psi$. If this check fails, then there is at least one execution that satisfies the assumption, and reasoning under the assumption is sensible.

In the case of FTS model checking, the set of products for which an assumption ψ can be discharged is computed with an extended model check, $ExtMc(fts, \neg\psi)$. This set is then intersected with the set of products satisfying $\psi \Rightarrow \phi$, yielding those that are non-vacuously satisfied. $ExtMc(fts, \neg\psi)$ will yield all products $px_{\not\models\neg\psi}$ that violate $\neg\psi$, meaning that each of them has a behaviour that satisfies the assumption.

Theorem 5.5. *For a property ϕ and an assumption ψ, the set of products that non-vacuously satisfy ϕ under the assumption ψ is $px_{\not\models\neg\psi} \cap px_{\models\psi\Rightarrow\phi}$.*

This extends to more general vacuity detection methods, for both CTL and LTL, such as the one by Beer *et al.* [Beer et al., 2001]. The proposed method derives a property $witness(\phi)$, which should be violated for ϕ to be satisfied non-vacuously. This leads to a similar treatment as above.

5.4.2 Deadlock detection

Deadlocks are a common problem in systems consisting of several parallel processes. In a typical scenario, some process has acquired a lock on resource X and waits for resource Y, while another process holds the lock on resource Y and waits for the lock on X to be released. In this case, both processes mutually block each other. Coffman *et al.* give the necessary and sufficient conditions for a deadlock to occur [Coffman et al., 1971].

In a transition system, a deadlocked state is a reachable state with no outgoing transition. In terms of transition systems, deadlock checking thus boils down to checking whether such a state is reachable from an initial state. We thus define absence of deadlocks in an FTS as absence of deadlocks in the transition system of every product. This yields the following decision problem.

Definition 5.6. *Given an FTS fts,* CheckDeadlock(fts) *returns true iff $\forall p \in [\![d]\!]_{FD}$, $fts_{|p}$ is free from deadlocks. Otherwise it returns false and px a non-empty set of products which contain deadlocks. Analogous to $ExtMc$, $ExtCheckDeadlock$ returns the full set of products with deadlocks.*

In classical transition systems, deadlock checking is mainly used when the transition system was obtained through parallel composition, as by definition, deadlocks need several processes competing for resources [Coffman et al., 1971]. It is interesting to observe that in FTS, deadlock checking is also sensible when there is no parallel composition involved. This is because in an FTS, a deadlock can also stem from an erroneous feature expression on a transition.

Consider the vending machine example from Figure 4.2. State ③ has three outgoing transitions, labeled with features *Soda*, *Tea* and *Cancel*. If there were a product without these features in which state ③ were reachable, it would be a state without outgoing transitions, hence a deadlock state. This problem does not occur since *Soda* or *Tea* are included in every product.

Deadlock detection algorithms for FTS thus have to account for the fact that each transition is labelled with an arbitrary feature expression. Clearly, states with no outgoing transition in the FTS will have none when projected to a product. However, since projection can remove transitions, there might also be states that lack outgoing transitions in certain products only. For a state s, the set of products for which it has an outgoing transition is given by

$$out(s) \triangleq [\![\bigvee_{s \xrightarrow{\alpha} s' \in trans} \gamma(s \xrightarrow{\alpha} s')]\!].$$

A state s thus lacks an outgoing transition in products $[\![d]\!]_{FD} \setminus out(s)$. This is not a problem as long as s is not reachable in these products. Let $in(s)$ be the products in which s is reachable. Deadlocked states are thus those for which $in(s) \not\subseteq out(s)$. This generalises the classical notion of deadlock, in which $out(s) = \emptyset$.

5.5 Conclusion

We discussed the problem of model checking in the context of SPLs. We found that model checking can be seen as orthogonal to other concerns in SPLE, and that temporal logics can be nearly reused as-is. We proposed the logics fLTL and fCTL, which extend their almost-namesakes with the ability to specify the set of products over which a property holds.

We defined two model checking problems. One problem is to find at least one violating product and report it. The other is to find all violating products. In the next chapter, we provide algorithms for solving these problems.

Chapter 6

Explicit Algorithms for FTS Model Checking

> *What worries me about religion is that it teaches people to be satisfied with not understanding.*
>
> Richard Dawkins, *BBC interview*, 1996

After having established the modelling language in Chapter 4 and its decision problems in Chapter 5, we can now provide algorithms for solving these problems. As a first step, we will introduce basic algorithmic principles for FTS model checking, and derive semi-symbolic fLTL model checking algorithms based on a depth-first search. In the following chapter, we will proceed to fully symbolic fixed-point based algorithms for fCTL model checking.

The algorithms will be introduced in several steps. After giving a straightforward (but potentially inefficient) algorithm in Section 6.1, we describe how to compute reachability in an FTS more efficiently in Section 6.2. We then study the skeleton of the FTS model checking algorithm in Section 6.3. Following this, we present algorithms that check reachability-based properties (such as absence of deadlocks, or safety) and fLTL in Section 6.4. Finally, we discuss optimisations in Section 6.5 and the computational complexity of these algorithms Section 6.6, before concluding in Section 6.7.

6.1 Introduction

As we have seen in Theorem 4.4, the classical model checking algorithm cannot be used immediately for FTS. It would find false positives, i.e., executions which do not exist in the FTS, and thus be incomplete. To use it for FTS model checking, the FTS has to be projected to a product first.

A straightforward algorithm for FTS model checking would thus be to iterate through the set of products (which can be computed from the FD), and to

model check each product separately using the model checking algorithm for single systems. We call this the *naïve algorithm.*

Algorithm 6.1 ($Mc([\chi]\phi, fts)$)**.** *Iterate through* $p \in [\![\chi]\!] \cap [\![d]\!]_{FD}$ *and compute* $SMc(\phi, fts_{|p})$*; if it is* 0*, halt and return* 0 *and* p*. After the last* p*, return* 1*.*

Algorithm 6.2 ($ExtMc([\chi]\phi, fts)$)**.** *The Boolean variable* r *is initialised to* 1*. Iterate through* $p \in [\![\chi]\!] \cap [\![d]\!]_{FD}$ *and compute* $SMc(\phi, fts_{|p})$*; if it is* 0*, add* p *to the output and set* r *to* 0*. After the last* p*, return* r*.*

This approach appears to be rather inefficient. Indeed, exponentially many products will be explored in spite of their great similarity.

6.2 Reachability in FTS

Model checking algorithms perform a search in the state space and produce information about states. As shown by Definition 4.2, each product of the FTS can have a different state space. Therefore, the model checking algorithm has to keep track of the states as well as the products in which they exist.

6.2.1 Introduction

The reachability relation R is the structure computed by the algorithm as the FTS is explored. It is not merely a set of states, but a set of couples. A couple (s, px) means that state s is reachable in the products in px.

Definition 6.3. *A reachability relation of an FTS is a function,* $R : S \to \mathcal{PP}(N)$*, so that* $\forall s \in S, p \in R(s)$*,* s *is* reachable *in* $fts_{|p}$*:* $\exists \pi \in [\![fts_{|p}]\!]_{TS}, i \in \mathbb{N} \bullet head(\pi_i) = s$ *(remember that* π_i *denotes the state at position* i *of the execution* π*). A reachability relation* R *is* full *when there is no state* $s \in S$ *reachable in a product* $p \in [\![d]\!]_{FD}$ *and* $p \notin R(s)$*. We assume that* $R(s) = \emptyset$ *if* $s \notin dom(R)$*.*

Computing R efficiently is the key of our algorithms. As shown before, the naïve way would be to explore the transition system of each product separately. This will yield a set of reachable states for each product, from which R can easily be obtained. Instead, we propose an algorithm that computes R by exploring the FTS. The starting point of this algorithm is the observation that the initial states of the FTS are part of all products.

Definition 6.4. Initially reachable *states of an FTS are*

$$Init \triangleq \big\{(s, [\![d]\!]_{FD}) \mid s \in I\big\}.$$

A transition t is part of the products that satisfy its feature expression, $[\![\gamma(t)]\!]$. Assuming that its source state is reachable by products in px, the transition can be fired by products in $px \cap [\![\gamma(t)]\!]$. This yields a set of products for which the target state is reachable.

Definition 6.5. *The* successors *of a state $s \in S$ reachable by products in $px \in \mathcal{PP}(N)$ are* $Post(s, px) \triangleq \{(s', px') \mid s \xrightarrow{\alpha} s' \in trans \ \wedge px' = px \cap [\![\gamma(s \xrightarrow{\alpha} s')]\!]\}$.

Let us illustrate this with the vending machine FTS of Figure 4.2. State ① is an initial state, and thus reachable by all products, $[\![d]\!]_{FD}$. From there, the transition ① $\xrightarrow{pay}$ ② can only be fired by products in $[\![\neg f]\!]$. In consequence, state ② is reachable by products in $[\![d]\!]_{FD} \cap [\![\neg f]\!]$. Transition ② $\xrightarrow{change}$ ③ can be fired for the same products. Transition ③ $\xrightarrow{soda}$ ⑤ can be fired for all products in $[\![s]\!]$, and so state ⑤ is reachable by products in $[\![d]\!]_{FD} \cap [\![s \wedge \neg f]\!]$, and so on.

Post is defined for single states, but can easily be extended to reachability relations, where $Post(R)$ stands for $\bigcup_{s \in S} Post(s, R(s))$. Since a state s can be a successor to several states in R, $Post(R)$ is not necessarily a function. To this end, the result of *Post* is always *compacted* as follows.

$$compact(x) \triangleq \{(s, px) \mid s \in S \wedge px = \textstyle\bigcup_{(s,px') \in x} px'\}.$$

The full reachability relation of an FTS, R_{full} is then given by a fixed point.

Theorem 6.6. $R_{full} = \mu X.Init \cup Post(X)$.

Proof. ($\subseteq$) Assume that $\exists s \in S, p \in [\![d]\!]_{FD} \bullet p \notin R_{full}(s)$ with s reachable in $fts_{|p}$. That is, an execution $\pi \in [\![fts_{|p}]\!]_{TS}$ visits s. Let $i \in \mathbb{N}$ be the smallest index such that $head(\pi_i) = s$. Since R_{full} is a fixed point of $\lambda X.Init \cup Post(X)$, we have $p \notin R_{full}(head(\pi_{i-1}))$ for otherwise $p \in R_{full}(s)$ since $(s, \{p\}) \in Post(head(\pi_{i-1}), \{p\})$ by our hypothesis. This recursive argument leads to $p \notin R_{full}(head(\pi))$, which is impossible as $head(\pi) \in I$ and $\forall s_i \in I \bullet R_{full}(s_i) = [\![d]\!]_{FD}$ since R_{full} is a fixed point of $\lambda X.Init \cup Post(X)$. Hence, no such state s exists.

($\supseteq$) Assume that $\exists s \in S, p \in [\![d]\!]_{FD} \bullet p \in R_{full}(s)$ with s not reachable in $fts_{|p}$. By definition of R_{full}, it contains some predecessor of s, say s' with $p \in R_{full}(s')$. If s is not reachable in p, then neither is s'. This observation recursively applies to all its predecessors, *pre*, none of which can be in *Init*. The relation $R' = \{(s', R_{full}(s')) \mid s' \notin pre\} \cup \{(s', R_{full}(s') \setminus \{p\}) \mid s' \in pre\}$ is thus a smaller fixed point of $\mu X.Init \cup Post(X)$, which is impossible by our hypothesis. □

As an illustration, the full reachability relation of the vending machine example is shown in Figure 6.1. Each box with a dashed border describes the set of products in which its corresponding state is reachable.

Reachability relations can be ordered.

Definition 6.7. $R_1 \preceq R_2 \Leftrightarrow \forall s \in S \bullet R_1(s) \subseteq R_2(s)$.

It is immediate that *Post* preserves this order. Following [Tarski, 1955], the fixed point from Theorem 6.6 can thus be computed by repeated application of *Post*. The full reachability relation of the FTS is also the maximal reachability relation computed from the initial states of the FTS.

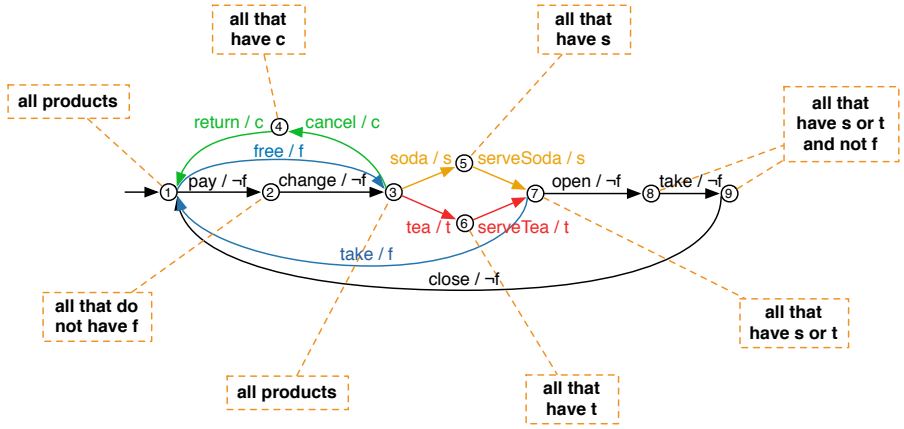


Figure 6.1: Full reachability relation of the vending machine.

6.2.2 Encoding sets of products

The main distinction between FTS and classical transition system model checking is that in FTS model checking, information about products has to be kept along with the states.

In practice, of course, it would be rather inefficient to record sets of products explicitly as suggested by Definition 6.3, that is, by enumerating them (e.g., using an array of bits). Here, we study two *symbolic* encodings for sets of products, which reduce the overhead caused by having to store sets of products.

The rf/ef encoding, $\mathbb{C}(N)$

One such encoding consists in representing sets of products by recording which features they must have (required features, rf) and which they cannot have (excluded features, ef). This symbolic data structure is defined as follows.

Definition 6.8. *Let* $\mathbb{C}(N) \triangleq \mathcal{P}(\mathcal{P}(N) \times \mathcal{P}(N))$. *The* rf/ef *encoding of a set of products is a set* $c \in \mathbb{C}(N)$ *with* $[\![c]\!] \triangleq \{p \in \mathcal{PP}(N) \mid (rf, ef) \in c \wedge rf \subseteq p \wedge ef \cap p = \emptyset\}$. *We sometimes write* $[\![rf, ef]\!]$ *as a shorthand for* $[\![\{(rf, ef)\}]\!]$.

For example, let $N = \{a, b, c\}$, then $[\![\{a\}, \{b\}]\!] = \big\{\{a\}, \{a, c\}\big\}$.

A couple (rf, ef) can be seen as a conjunction of literals; positive literals for the elements in rf and negative for those in ef, e.g., $a \wedge \neg b$ for the previous example. A set of such couples is thus isomorphic to a Boolean expression in disjunctive normal form. Since any expression in propositional logic can be translated into disjunctive normal form, the FD and any feature expression can be represented by a set of (rf, ef) couples. Let $\mathbb{C}(d)$ denote the rf/ef encoding of an FD d. Given two sets of couples c_1 and c_2, the union (resp.

intersection) of the two sets they represent can be calculated with an operation of disjunction (resp. conjunction) defined as follows.

Definition 6.9. *For $c_1, c_2 \in \mathbb{C}(N)$, disjunction is $c_1 \vee c_2 \triangleq c_1 \cup c_2$ and conjunction $c_1 \wedge c_2 \triangleq \{(rf_1 \cup rf_2, ef_1 \cup ef_2) \mid (rf_1, ef_1) \in c_1 \wedge (rf_2, ef_2) \in c_2\}$.*

Theorem 6.10. *For $c_1, c_2 \in \mathbb{C}(N)$, $c_1 \vee c_2 = [\![c_1]\!] \cup [\![c_2]\!]$ and $c_1 \wedge c_2 = [\![c_1]\!] \cap [\![c_2]\!]$.*

With this encoding, the initially reachable states $Init_c$ and the successors $Post_c$ (the 'c' subscript stands for the rf/ef encoding) are defined as follows.

Definition 6.11. *In the rf/ef encoding, R_c is a function $S \to \mathbb{C}(N)$. The initially reachable states are $Init_c \triangleq \{(s, \mathbb{C}(d)) \mid s \in I\}$; the successors of a state $s \in S$ reachable by products symbolically given by $c \in \mathbb{C}(N)$ are*

$$Post_c(s, c) \triangleq \{(s', c') \mid s \xrightarrow{\alpha} s' \in trans \wedge \ c' = c \wedge \gamma(s \xrightarrow{\alpha} s')\}.$$

Where the conjunction of two symbolic sets is calculated as in Definition 6.9.

As before, $Post_c$ is defined for single states but extends to reachability relations, where $Post_c(R)$ stands for $\bigvee_{s \in S} Post_c(s, R(s))$. Furthermore, $Post_c$ does not necessarily produce a function, since a state s' can be a successor to s via several transitions. To this end, application of $Post_c$ is always followed by:

$$compact(x) \triangleq \{(s, c) \mid s \in S \wedge c = \bigvee_{(s,c') \in x} c'\}.$$

The symbolic successor function is equivalent to its explicit counterpart:

Theorem 6.12. *For any $(s, c) \in S \to \mathbb{C}(N)$,*

$$Post(s, [\![c]\!]) = \{s', [\![c']\!] \mid (s', c') \in Post_c(s, c)\}$$

Proof. This follows from the observations that any set of products can be rf/ef-encoded and that conjunction of two such sets corresponds to the intersection of their semantics, or $\forall c_1, c_2 \in \mathbb{C}(N) \bullet [\![c_1 \wedge c_2]\!] = [\![c_1]\!] \cap [\![c_2]\!]$. □

Let us illustrate this with the vending machine FTS of Figure 4.2 (and repeated in Figure 6.2). State ① is an initial state, and thus reachable by all products, $\mathbb{C}(d)$. From there, the transition ① $\xrightarrow{pay}$ ② can only be fired by products not containing the feature f (the label of ① $\xrightarrow{pay}$ ②). State ② is thus reachable by these products only, that is, the feature f is added to the excluded features: $\{(\emptyset, \{f\})\} \wedge \mathbb{C}(d)$. From state ②, transition ② $\xrightarrow{change}$ ③ can be fired for the same products. From state ③, transition ③ $\xrightarrow{soda}$ ⑤ can be fired for the products containing s. State ⑤ is thus reachable by all products containing s and not containing f, that is, s is added to the required transitions: $\{(\{v\}, \{f\})\} \wedge \mathbb{C}(d)$. Figure 6.2 shows the full reachability relation of the vending machine in rf/ef encoding, except for the $\wedge\, \mathbb{C}(d)$.

The rf/ef encoding can be optimised in several ways. The most important optimisation is based on anti-chains [De Wulf et al., 2006]. A couple (rf, ef) in

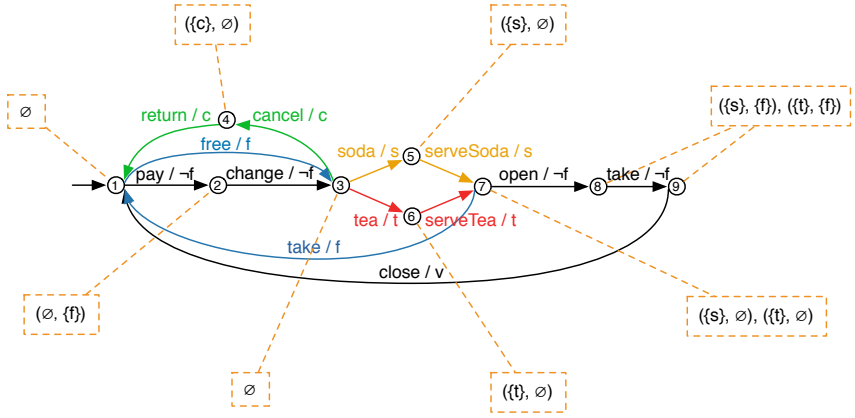


Figure 6.2: Full reachability relation of the vending machine in rf/ef encoding.

a symbolic encoding c represents a set of products $[\![rf, ef]\!]$. If another couple $(rf', ef') \in c$ is so that $[\![rf', ef']\!] \subseteq [\![rf, ef]\!]$, then (rf', ef') can be removed from c without loss of information. More generally, it is sufficient to keep the maximal elements (an anti-chain) of the partial order induced by the subset relation $\subseteq$ over the sets of products represented by each couple. Fortunately, the partial order has a correspondence in the symbolic encoding:

Definition 6.13. *For singletons in $\mathbb{C}(N)$, the relation $\sqsubseteq$ is defined as a partial order: $(rf, ef) \sqsubseteq (rf', ef') \triangleq (rf \supseteq rf') \wedge (ef \supseteq ef')$. For two sets $c, c' \subseteq \mathbb{C}(N)$, $c \sqsubseteq c' \triangleq \forall (rf, ef) \in c, \exists (rf', ef') \in c' \bullet (rf, ef) \sqsubseteq (rf', ef')$.*

Theorem 6.14. *For any $c, c' \subseteq \mathbb{C}(N)$, $c \sqsubseteq c' \Leftrightarrow [\![c]\!] \subseteq [\![c']\!]$.*

It can be shown that the partially ordered set of products and the one of symbolic couples are related by a Galois connection.

Theorem 6.15. *The semantic function, $[\![]\!]$, and the following* abstraction function*: $symb(px) = \left(\bigcap px, N \setminus \bigcup px\right)$, are a Galois connection of the explicit and the symbolic encoding with their partial orders; symb is the lower adjoint of $[\![]\!]$.*

Proof. Let $L_1 = \mathcal{PP}(N)$ and $L_2 = \mathbb{C}(N)$, then $symb : L_1 \to L_2$ and $[\![]\!] : L_2 \to L_1$ are so that $\forall px \in L_1$

$$
\begin{aligned}
& px \subseteq [\![symb(px)]\!] \\
& px \subseteq [\![\textstyle\bigcap px, N \setminus \bigcup px]\!] \\
& px \subseteq \{p \in L_1 \mid \textstyle\bigcap px \subseteq p \wedge (N \setminus \bigcup px) \cap p = \emptyset\} \\
\forall p' \in px \bullet\ & \textstyle\bigcap px \subseteq p' \wedge (N \setminus \bigcup px) \cap p' = \emptyset
\end{aligned}
$$

and $\forall (rf, ef) \in L_2$

$$\begin{aligned} symb([\![rf, ef]\!]) &\sqsubseteq (rf, ef) \\ (\bigcap [\![rf, ef]\!], N \setminus \bigcup [\![rf, ef]\!]) &\sqsubseteq (rf, ef) \\ (rf, N \setminus (N \setminus ef)) &\sqsubseteq (rf, ef) \\ (rf, ef) &= (rf, ef) \end{aligned}$$

□

Given this partial order, the optimisation consists in filtering out couples that are smaller than other couples wrt. $\sqsubseteq$. More importantly, testing whether a state s is reachable by products in $[\![rf, ef]\!]$ should not be done by just checking whether $R_c(s) = (rf, ef)$. Rather, one has to check whether a larger or equal couple exists: $\exists (rf', ef') \in R_c(s) \bullet (rf, ef) \sqsubseteq (rf', ef')$.

Further, for each couple (rf, ef), $rf \cap ef \neq \emptyset \Rightarrow [\![rf, ef]\!] = \emptyset$, which means that such couples can be removed. These optimisations can be incorporated into the *compact* function.

The Boolean function encoding, $\mathbb{B}(N)$

An alternative encoding for sets of products are Boolean functions, such as those used already for feature expressions. Such functions can be represented by BDDs, a symbolic representation on which propositional operators can be computed efficiently [Bryant, 1992]. Boolean functions have the advantage of being applicable in all cases and being close to the definitions we already have.

Recall that for a set of features N, $\mathbb{B}(N)$ denotes the set of all Boolean functions over the variables N, and given an FD d, $\mathbb{B}(d)$ denote its Boolean function encoding (see Section 1.2). In the case of the Boolean function encoding, the elements of the reachability calculation are defined as follows.

Definition 6.16. *R_b is a function $S \to \mathbb{B}(N)$. The initially reachable states are $Init_b \triangleq \{(s, \mathbb{B}(d)) \mid s \in I\}$; the successors of $s \in S$ reachable by products $\chi \in \mathbb{B}(N)$ are*

$$Post_b(s, \chi) \triangleq \{(s', \chi') \mid s \xrightarrow{\alpha} s' \in trans \wedge \ \chi' = \chi \wedge \gamma(s \xrightarrow{\alpha} s')\}.$$

Given $Init_b$ and $Post_b$, the fixed point computation for R_{bfull} is the same as in Theorem 6.6. The *compact* function uses disjunction instead of set union:

$$compact_b(x) \triangleq \{(s, \chi) \mid s \in S \wedge c = \bigvee_{(s,\chi') \in x} \chi'\}.$$

Again, let us illustrate this with the vending machine FTS of Figure 4.2 (and repeated in Figure 6.3). State ① is an initial state, hence $R_b(①) = \mathbb{B}(d)$. Firing transition ① $\xrightarrow{pay}$ ② yields $R(②) = \mathbb{B}(d) \wedge \neg f$, of ② $\xrightarrow{change}$ ③ yields $R(③) = \mathbb{B}(d) \wedge \neg f$, of ③ $\xrightarrow{soda}$ ⑤ yields $R(⑤) = \mathbb{B}(d) \wedge \neg f \wedge s$, and so on. This is very similar to the calculation of the explicit reachability relation. The difference from the explicit reachability relation is that the function $\mathbb{B}(d) \wedge \neg f \wedge s$

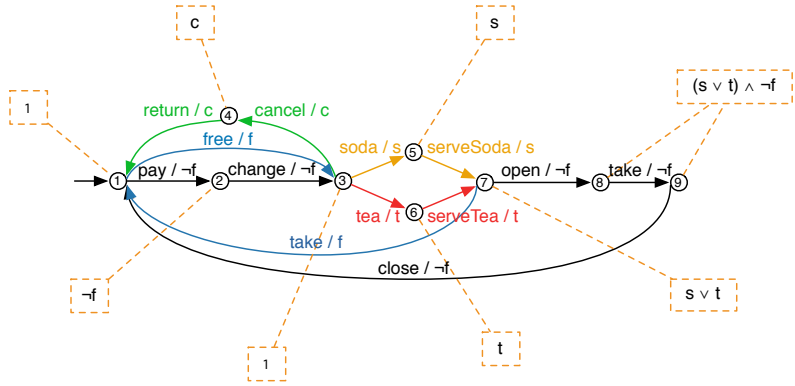


Figure 6.3: Full reachability relation of the vending machine in the Boolean function encoding.

is stored instead of the set of products that it represents. Figure 6.3 shows the full reachability relation of the vending machine in the Boolean function encoding, except for the $\wedge\, \mathbb{B}(d)$.

The subset relation $\subseteq$ is a partial order over sets of products. It has a correspondence in $\mathbb{B}(N)$.

Definition 6.17. *For* $\chi, \chi' \in \mathbb{B}(N)$, $\chi \sqsubseteq \chi' \triangleq \chi \Rightarrow \chi'$.

Theorem 6.18. *For* $\chi, \chi' \in \mathbb{B}(N)$, $\chi \sqsubseteq \chi' \Leftrightarrow [\![\chi]\!] \subseteq [\![\chi']\!]$.

This relation is used to test whether a state s is already known to be reachable in products $[\![\chi]\!]$, i.e., $\chi \sqsubseteq R_b(s)$. In practice, it can be implemented with a satisfiability check: $\chi \sqsubseteq R_b(s)$ is equivalent to UNSAT $\chi \wedge \neg R_b(s)$. All computations involving sets of products can thus be performed on symbolic sets. The anti-chain optimisations required for the rf/ef encoding are immediate here.

An alternative to using BDDs is to store Boolean functions directly as parse trees. This has some disadvantages, such as the difficulty of keeping the formula concise and the necessity of transforming it into a different format for analysis. Formulas could also be stored directly in *Conjunctive Normal Form* (CNF). This allows for some degree of minimisation, but suffers from blowup when using disjunction, a rather common operation in our algorithms.

Comparison and usage

The Boolean function encoding is polyvalent, in that it can ensure a rather concise representation for most sets of products. Furthermore, the abundance of freely available BDD implementations allows for reuse when implementing an algorithm using this encoding. Our tool SNIP (see Chapter 9) thus uses the

Boolean function encoding, implemented with both BDDs and CNFs. Experiments showed that the use of CNFs is rather inefficient when no minimisation techniques are used. An avenue which has not yet been explored is to store Boolean functions as parse trees. This can be efficient when the Boolean functions can be minimised so that most functions that come up are identical. This would allow the use of caching, e.g., for SAT results.

The rf/ef encoding is less polyvalent. A close look at Definition 6.9 shows that the size of the conjunction of two symbolic sets c_1 and c_2 is $|c_1|.|c_2|$. Although the optimisations discussed above can somewhat mitigate this, the repeated application of conjunction will quickly lead to large sets of couples. Eventually, this will bring any algorithm using this encoding to a crawl. However, whenever conjunction is used in the algorithm, one member is a transition label (see Definition 6.11). If the rf/ef encoding of all feature labels were singleton sets, one factor in $|c_1|.|c_2|$ could be reduced to one each time, thereby avoiding the blowup. In terms of feature labels, singleton rf/ef couples correspond to conjunctions of literals. This means that the rf/ef encoding can actually be used for FTS restricted to such feature labels. In fact, GenFTS(FD, prior) is such an FTS language—hence our use of the rf/ef encoding in the first FTS implementation, the Haskell FTS library [Classen et al., 2010b].

The rf/ef encoding has another more serious limitation: it is hard to derive a compact rf/ef encoding for the FD. The Boolean function encoding of an FD naturally leads to a CNF, whose size will increase exponentially when transformed into a DNF (which would correspond to the rf/ef encoding). Moreover, $|\mathbb{C}(d)| = O(2^n)$. This means that $Init_c$ would potentially be huge, and since R_c is derived from $Init_c$, it would be huge too. Strictly speaking, the worst-case size is the same for the Boolean function encoding, $|\mathbb{B}(d)| = O(2^n)$. However, BDDs tend to be small in practice, and have been used successfully in the context of FDs [Mendonca et al., 2009, Mendonca, 2009]. A solution to this problem is to use over-approximation, which we discuss in the next section.

6.2.3 The role of the FD

As $Init_b$ is initialised (henceforth called *seeded*) with the symbolic encoding of the FD, all subsequent computations will only consider valid products. A potential drawback of this (in addition to the one pointed out just before) is that the symbolic encoding of the FD will eventually be repeated for every state in R_c or R_b. Since it does not change over the course of the verification and is part of any state, it could as well be factored out. This would lead to smaller symbolic encodings for the products of each state.

An alternative is thus to initialise $Init_c$ or $Init_b$ with the set of all *possible* products, i.e., $\emptyset$ for the rf/ef encoding and 1 for the Boolean function encoding. This way, the reachability relation captures the products that can reach a state, without the guarantee that these products are indeed valid. An additional check is thus required to prevent the computation from considering invalid products, as this might lead to false positives (i.e., an error state found to be reachable,

but in an invalid product) and unnecessary calculations (i.e., calculating the reachable states of invalid products).

However, this cannot be done by explicitly excluding invalid products from the symbolic sets. This would lead back to the problem we are trying to solve. Instead, it is sufficient to make sure that each symbolic set of products contains at least one valid product. This way, no computation over symbolic sets will lead to false positives. In practice, the symbolic sets whose intersection with the set of valid products is empty can be removed as part of the *compact* function. The non-emptiness of the intersection of these and the FD can easily be checked with SAT (for the rf/ef encoding), or BDD manipulations.

The FD not only influences the reachable states. It also plays a role when testing subset relations between sets of products. E.g., for the Boolean function encoding, when testing whether a state s is reachable in products $[\![\chi]\!]$, we test whether $\chi \sqsubseteq R_b(s)$. The result of this can depend of the FD. For instance, let $\chi = a$ and $R_b(s) = b$, clearly $\chi \not\sqsubseteq R_b(s)$, meaning that s is not reachable. However, if the FD had a constraint saying that b requires a (i.e., $b \implies a$), then s would in fact be reachable.

Theorem 6.19. *For two symbolic sets of products $\chi, \chi' \in \mathbb{B}(N)$, $\chi \sqsubseteq \chi' \implies [\![\chi]\!] \cap [\![d]\!]_{FD} \subseteq [\![\chi']\!] \cap [\![d]\!]_{FD}$. The reverse implication does not always hold.*

Proof. Follows from Theorem 6.18 and the example. □

Not using the FD during this calculation is thus an over-approximation and has to be accounted for in the algorithms. For instance, it is fine to use it for the reachability test described above. While it might happen that a state is visited even though it is already known to be reachable, Theorem 6.19 guarantees that an unexplored state will be recognised as such. The algorithm will still be sound. The inefficiency due to unnecessary state visits is hoped to be offset by the gain in efficiency when performing the subset test.

Note also that the use of this approximation might result in an unsound or incomplete algorithm. This is the case for the deadlock test, $in(s) \not\subseteq out(s)$, discussed in Section 5.4.2. A corollary of Theorem 6.19 is that whether or not $in(s) \not\subseteq out(s)$ cannot be determined correctly when $in(s)$ and $out(s)$ are approximations. To correct for the approximation, both symbolic sets have to be intersected with the FD first. Note that in the case of the algorithm, $in(s)$ corresponds to $R_b(s)$ whereas $out(s)$ corresponds to a Boolean function χ, calculated locally for a state. For the Boolean function encoding, the test whether $[\![\chi]\!] \cap [\![d]\!]_{FD} \not\subseteq [\![R_b(s)]\!] \cap [\![d]\!]_{FD}$ is equivalent to SAT $\mathbb{B}(d) \wedge \chi \wedge \neg R_b(s)$. Similarly, the test whether $[\![\chi]\!] \cap [\![d]\!]_{FD} \subseteq [\![R_b(s)]\!] \cap [\![d]\!]_{FD}$ is equivalent to UNSAT of the same expression.

For the rf/ef encoding, tests like this are most efficiently implemented by SAT checks. This leads to a hybrid approach where the rf/ef encoding is used for sets attached to states, while the FD is represented in a format that can be readily analysed, e.g., as CNF. For non-emptiness checking, an rf/ef set can then be transformed into the format of the FD for the purpose of the analysis.

6.2.4 Product quantification

Before proceeding to the algorithms, let us discuss how product quantification can be handled. Recall, from Section 5.2, that the set of products for which a property should hold can be expressed by a product quantifier, that is, a Boolean function.

The net effect of a quantifier is to restrict the set of products. It can be seen as an additional constraint over the FD.

Theorem 6.20. *Verification of a quantified property $[\chi]\phi$ over an FTS with FD d is equivalent to verifying the non-quantified property ϕ over the FTS with a changed FD d', obtained by adding χ as a constraint to d.*

Proof. Follows from Definition 5.1. □

Verification of quantified properties can thus be reduced to verification of non-quantified properties, e.g., fLTL to LTL and fCTL to CTL. We therefore only need to give algorithms for properties *without* product quantification.

6.3 Computing R with a depth-first search

Our algorithms are based on computing R. While Theorem 6.6 could be implemented right away with set operations [Clarke et al., 1999], this is inefficient unless data structures such as BDDs are used to represent all sets. This would result in a *fully symbolic* algorithm, which we cover in the following chapter. Here, we are are interested in *semi-symbolic* algorithms, where products are represented symbolically but states are explored explicitly, one by one.

Explicit state space exploration is similar to graph exploration, and the two types of algorithm generally used are *Depth-First Search* (DFS) and *Breadth-First Search* (BFS). Procedure `Reachables` given below computes R with a DFS. Its output is equivalent to the result of Theorem 6.6. This procedure serves as the basis for all subsequent algorithms. It is therefore kept simple, and abstracts away from the symbolic encodings discussed previously.

Our algorithm generalises the standard DFS algorithm for transition systems, by marking states with sets of products, rather than Boolean *visited* flags. In contrast to the DFS algorithm for transition systems, where no state is visited twice, our algorithm can visit states multiple times. This follows directly from Theorem 6.6 and is due to the fact that reachability is defined wrt. a set of products. When $R(s) = p_x$ and the DFS arrives at s for the second time with $p_{x'} \not\subseteq p_x$, then s, although already visited, has to be *re-explored*. This is because transitions that were disallowed for p_x might be allowed in $p_{x'}$.

The algorithm maintains R and a stack of states, the *execution stack*. Line 1 mimics *Init* from Definition 6.4, modulo the discussion of Section 6.2.3: the initial states are reachable in all feature combinations (not only the valid products), and a DFS is started for each of them (line 6). At each iteration, the DFS calculates the set *new* of unvisited successors of the current state (line 9).

Input: $fts = (S, Act, trans, I, AP, L, d, \gamma)$.
Output: The full reachability relation of fts.

```
1  R ← {(s0, PP(N)) | s0 ∈ I};
2  Stack ← [];
3  while I ≠ ∅ do
4  |  Take s0 from I;
5  |  I ← I \ {s0};
6  |  push((s0, PP(N)), Stack);
7  |  while Stack ≠ [] do
8  |  |  (s, px) ← top(Stack);
   |  |  new ← { (s', px'') | (s', px') ∈ Post(s, px) ∧
   |  |                       px'' = px' \ R(s') ∧
   |  |                       px'' ∩ [[d]]_FD ≠ ∅ ∧
9  |  |                       px'' ≠ ∅ };
10 |  |  if new = ∅ then
11 |  |  |  pop(Stack)
12 |  |  else
13 |  |  |  Take (s', px') ∈ new;
14 |  |  |  R(s') ← R(s') ∪ px';
15 |  |  |  push((s', px'), Stack)
16 |  |  end
17 |  end
18 end
19 return R
```

Procedure `Reachables`(fts)

It uses the *Post* operator from Definition 6.5 (first condition), and only considers products that are not yet visited (second condition) and that are valid (third condition). It also makes sure that at least one valid product is among the remaining products. If all successors were visited, the procedure backtracks (line 11). Otherwise it proceeds with one of the successor states, which is added to R (line 15).

An illustration of the `Reachables` procedure based on the vending machine example is shown in Figure 6.4. As in Figure 6.3, we use the Boolean function encoding. The algorithm completes the illustration started in Section 6.2, except that it follows the `Reachables` procedure and does not seed the initial states with $\mathbb{B}(d)$. State ① is thus labelled with 1. Firing transition ① $\xrightarrow{pay}$ ② yields $\neg f$ for state ②, and so on.[1] Each Boolean function in Figure 6.4 is numbered, which represents the succession of Boolean functions uncovered along the DFS executed by `Reachables`. When execution reaches state ④ in step 4,

[1] The order chosen for transitions in the DFS is to start with the transition leading to the lower indexed state, e.g., first ③ $\xrightarrow{cancel}$ ④, then ③ $\xrightarrow{soda}$ ⑤, and finally ③ $\xrightarrow{tea}$ ⑥.

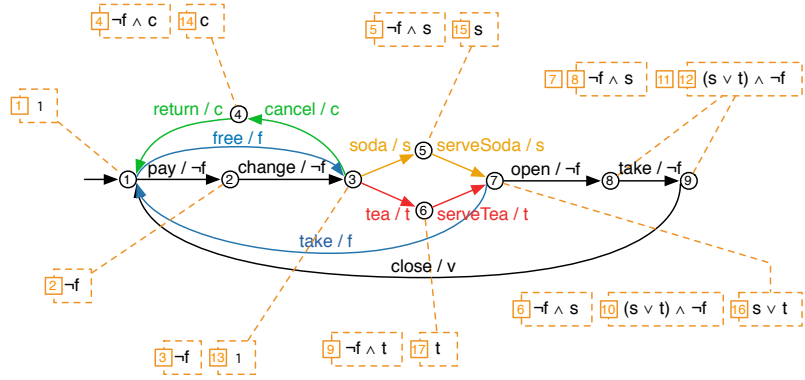


Figure 6.4: Computing the full reachability relation with a DFS.

the set *new* at line 9 is empty. This is because the only outgoing transition, ④ $\xrightarrow{return}$ ①, would lead to feature expression $\neg f \wedge c$ for state ①, with $R(①) = 1$. The procedure thus backtracks to state ③, where transition ③ $\xrightarrow{soda}$ ⑤ is the next transition that can be fired (step 5 in Figure 6.4). At some point (step 12), execution will backtrack to state ①, from which transition ① $\xrightarrow{free}$ ③ leads to state ③ for the second time, with feature expression f. Since $R(③) = \neg f$ and $[\![f]\!] \not\subseteq [\![\neg f]\!]$, it has to be re-explored. Hence $R(③)$ becomes 1 (step 13), and states ⑤, ⑥ and ⑦ are re-explored as well.

Let us study some properties of the algorithm. A starting observation is that the sets of products decrease monotonically along the execution stack, and that the sets of products on an execution stack are always subsets of those in R.

Theorem 6.21. *Given an execution stack* $[(s_0, px_0), \ldots, (s_k, px_k)]$*, always* $\forall i \in [1, k] \bullet px_i \subseteq px_{i-1}$.

Proof. This follows from the observation that *Post* never expands the set of products passed to it. □

Theorem 6.22. *For any state* s *on an execution stack* $[\ldots, (s, px), \ldots]$ *with reachability relation* R*:* $px \subseteq R(s)$.

Proof. For the initial states, this follows immediately from lines 1 and 6 and otherwise from lines 14 and 15. That not necessarily $px = R(s)$ follows from the observation that s might be re-explored. □

A consequence of these theorems is that even with re-explorations, a state cannot appear twice on an execution stack. Intuitively, a re-exploration happens when reaching a state with new products. Because any state that is already on the stack will be reached with *less* products than before, the re-explored state cannot be on the stack already.

Theorem 6.23. *Given an execution stack* $[(s_0, px_0), \ldots, (s_k, px_k)]$, *always* $\forall i \in [0..k-1] \bullet s_i \neq s_k$.

Proof. Let us assume that $\exists i \in [0..k-1] \bullet s_i = s_k$. Theorem 6.21 implies that $px_k \subseteq px_i$; furthermore, $px_i \subseteq R(s_i)$. Hence, $px_k \subseteq R(s_i)$ meaning that (s_k, px_k) cannot have been in *new*, contradicting the hypothesis. □

These considerations are particularly helpful when implementing the algorithm. They allow to make certain assumptions (e.g., a state is only once on the stack) and write more efficient code.

Another interesting observation is that the execution stack does not need to contain the sets of features for which each state was visited. In fact, it is sufficient to use those in R.

Theorem 6.24. *Let* (s, px) *be the top of the execution stack and* R *the reachability relation at this point,*

$$\begin{aligned} &\{(s', px') \in Post(s, px) \quad \mid px' \not\subseteq R(s')\} \\ = &\{(s', px') \in Post(s, R(s)) \mid px' \not\subseteq R(s')\} \end{aligned}$$

Proof. By Theorem 6.23, s only appears once on the stack. Hence, the DFS for products in $R(s) \setminus px$ is finished and $\forall (s', px') \in Post(s, R(s) \setminus px) \bullet px' \subseteq R(s')$. □

While this approach might reduce the memory needed for the stack, it slows down the calculation of *new*, since more states are generated and checked than necessary. This is a memory/speed trade-off.

A DFS can also be used to check whether a certain state is on a cycle, i.e., whether it can be reached from itself. This DFS is generally nested inside a DFS that identifies the states for which this check is to be performed. Intuitively, it corresponds to calculating a reachability relation R' starting with an arbitrary state (s, px) instead of an initial state. Recall that a reachability relation is *maximal* when it cannot be expanded anymore. The maximal reachability relations that can be computed from different initial states can be different. More precisely, if the computation of R' is seeded with a subset of R, the maximal R' will be smaller than the maximal R.

Theorem 6.25. *For two maximal reachability relations* R *and* R', *when* R' *is computed from* $Init' \subseteq R$, *then* $R' \preceq R$.

Proof. Because R and R' are both maximal, the computation that started form $Init'$ was also part of the computation of R. Its result, R', is thus part of R. □

Note that the above theorem is independent from the algorithm used to implement the calculation. Nevertheless, it is important for the case of nested DFSs. Suppose that a DFS is started from a state deep in the stack, and the algorithm then backtracks and starts a DFS from a state not as deep. By Theorem 6.25, the second DFS does not have to start from scratch, but can

extend the results that were found earlier. Whatever the first DFS found would be found by the second as well.

In the context of nested DFSs, an important corollary of Theorem 6.21 is that any set of products that comes up along an execution has a non-empty intersection with those before it. When checking whether a state s is on a cycle, it is sufficient to check whether any state on the execution stack up to s can be reached. The set of products for which it can be reached is necessarily a subset of the set of products for which it could be reached the first time.

6.4 Model checking algorithms

With the basic techniques covered, we can now turn to the algorithms. We proceed in two steps. We start with reachability properties, i.e., rather easy properties which can be expressed without a temporal logic. The algorithm for reachability is based on a single DFS. We then proceed to the algorithm for LTL, which is based on a nested DFS. By Theorem 6.20, the algorithm for fLTL is an immediate derivative of this algorithm.

6.4.1 Model checking reachability properties

Assertions, safety properties or absence of deadlocks are *reachability properties*. They can be expressed as properties that should hold for each state. To verify them, it is thus sufficient to check whether violating states are reachable. Let us first specify what *violating states*, denoted *bad* hereafter, are in each case.

Assertions. An assertion specifies a condition ϕ on the atomic propositions that should hold in all states, i.e., *bad* is given by $\{s \in S \mid L(s) \not\models \phi\}$.

Safety properties. A safety property expresses a prefix of behaviour that a system should not have. They can be expressed in LTL, or as automata. In automata-based model checking [Vardi and Wolper, 1986], safety properties are known as *regular* properties. All violations of a safety property ϕ can be expressed by an automaton $A_{\neg\phi}$, which accepts exactly the bad behavioural prefixes. To check an FTS against such a property, their *synchronous product* has to be calculated. In terms of languages, the synchronous product represents the intersection of the language of the FTS and the language of the automaton. Model checking a regular property amounts to checking whether this intersection is empty. If it is not, there are executions that violate the property. The synchronous product of the automaton and the FTS yields an FTS with readily identifiable bad states. The synchronous product of an FTS and an automaton is similar to that of a transition system and an automaton [Baier and Katoen, 2008]. A transition $s \xrightarrow{\alpha} t$ of the FTS synchronises with a transition $q \xrightarrow{apx} p$ when the transition is labeled with the atomic propositions of the target state, $apx = L(t)$. The only difference from the standard definition is that it has to preserve the feature expressions of the original FTS.

Definition 6.26. *For an FTS* $fts = (S, Act, trans, I, AP, L, d, \gamma)$ *and an automaton* $a = (Q, \mathcal{P}(AP), \delta, Q_0, F)$*, the* synchronous product *is an FTS* $fts \otimes a = (S \times Q, Act, trans', I', AP', L', d, \gamma')$*, where*

- $AP' = Q$ *and* $L'(s,q) = q$*, i.e. the new FTS is labeled with the states of the automaton,*
- $(s,q) \xrightarrow{\alpha}{}' (t,p)$ *iff* $s \xrightarrow{\alpha} t \ \wedge \ q \xrightarrow{L(t)} p$*,*
- $I' = \{(s_0,q) \mid s_0 \in I \ \wedge \ \exists q_0 \in Q_0 \bullet (q_0, L(s_0), q) \in \delta\}$*, i.e. the initial states are those that can be reached from an initial state of the automaton,*
- $\gamma'\big((s,q) \xrightarrow{\alpha}{}' (t,p)\big) = \gamma(s \xrightarrow{\alpha} t)$*,*

The violating states in the resulting FTS are those that are labelled with an accepting state of the automaton, i.e. $bad = \{s \in S \times Q \mid L'(s) \in F\}$. The relation between safety properties and automata is well known [Baier and Katoen, 2008] and will not be further explored here.

Deadlocks. Absence of deadlocks in an FTS is defined as absence of deadlocks in the transition system of each product, cfr. Definition 5.6. As noted before, projection can remove transitions, which means that states might lack outgoing transitions in certain products only. A deadlocked state is thus one in which $R(s) \not\subseteq out(s)$, where $out(s) \triangleq [\![\bigvee_{s \xrightarrow{\alpha} s' \in trans} \gamma(s \xrightarrow{\alpha} s')]\!]$. Hence, $bad = \{s \in S \mid R(s) \not\subseteq out(s)\}$. $R(s)$ is calculated when computing reachability, and $out(s)$ can be determined locally for each state. This is thus a reachability property. When s is a deadlock state, the set of deadlocked products is given by $R(s) \setminus out(s)$. This is different from the previous cases, in which the bad products are always $R(s)$, i.e., those in which the state is reachable.

Algorithms. Once the set of bad states is known, the algorithm is the same in all three cases: compute the reachability relation R as discussed in Section 6.3, and each time a state s is added or has its set of products updated, test whether $s \in bad$. This leads to the following algorithms.

Algorithm 6.27 ($Mc_{reach}(\phi, fts)$ or $CheckDeadlock(fts)$)**.** *Compute* R *(using the* `Reachables` *procedure) until the first bad state* s *is found. At this point, return* 0*, the current execution stack (which is the counterexample), and the set of products* px *for which the state is bad. For asserts and safety properties,* $px = R(s)$ *whereas for deadlocks,* $px = R(s) \setminus out(s)$*. When the maximal* R *is computed and no bad state is found, return* 1*.*

Algorithm 6.28 ($ExtMc_{reach}(\phi, fts)$ or $ExtCheckDeadlock(fts)$)**.** *Maintain a set of counterexamples* c *while computing the maximal* R*. Each time a bad state is encountered, the couple* (e, px) *with the current execution stack* e *and the set of bad products* px *is added to* c*. When the algorithm finishes and* c *is empty, return* 1*. Otherwise, return* 0 *and* c*. In this case, the set of violating products is given by* $[\![d]\!]_{FD} \cap \bigcup_{(e,px) \in c} px$*.*

Again, these algorithms abstract away from the symbolic encoding used to implement the sets of products; they can be implemented with either one. The

sets px will then be in the form of a symbolic set. To present a symbolic set to the user, it is generally formatted as a Boolean expression. Note that this is only user-friendly because we follow the approach discussed in Section 6.2.3, that is, because $Init_c$ or $Init_b$ are not initialised with the encoding of the FD. Otherwise, the expression shown to the user would contain the entire encoding of the FD, which would look rather obscure. Instead, the output just mentions the features that were essential for the particular violation.

One could further simplify the returned expression based on information contained in the FD. Assume that the bad products are $a \wedge b$ and that the feature a appears in all products. In this case, the expression could be simplified to just b. However, such a simplification would require additional computation and might mislead the user into thinking that a is not involved in the problem at hand. Therefore, we do not implement such simplifications in our tools.

For the ExtMc$_{reach}$ and ExtCheckDeadlock algorithms, another consideration for user-friendliness is to present a summary of the results at the end of the execution. For instance, the set of all violating products is easy to calculate during the algorithm, and should be included as part of the summary.

It is important to note that these algorithms produce a counterexample for each product reported in the violation. Although there is a counterexample for each product, the number of counterexamples is generally much lower, as the algorithm associates counterexamples to sets of products (i.e., given a set of violating products, the given counterexample is an execution in all of these products). However, as presented above, Algorithm 6.28 might return multiple counterexamples for the same product. In Section 6.5 we describe an optimisation of the algorithm that prevents this.

Furthermore, note that unlike Algorithm 6.1, the naïve algorithm, Algorithm 6.27 can provide a *set* of violating products, not just a single product. This is an important advantage, as it gives hints about which features cause a problem. For example, if a certain feature does not come up on any of the transitions in the execution, the algorithm provides this information to the user. Furthermore, the algorithm will automatically ignore features that do not appear in the FTS, such as the currency related features in the vending machine example. The naïve algorithm will require some sort of preprocessing to determine those features, in order to avoid checking products with identical transition systems.

6.4.2 Model checking LTL properties

LTL properties are part of the class of ω-regular properties for which we use the technique of automata-based model checking given in [Vardi and Wolper, 1986]. The treatment is similar to that of safety properties. Given an LTL property ϕ, it consists in constructing a Büchi automaton, $a_{\neg\phi}$, that accepts all the executions that violate ϕ. Büchi automata accept infinite behaviours. The synchronous product (Definition 6.26) of this automaton and the FTS yields the FTS that is explored.

The resulting FTS, $fts \otimes a_{\neg\phi}$, has labeled accepting states: $accept \triangleq \{s \in S \times Q \mid L'(s) \in F\}$. If one of them can be visited infinitely often (the Büchi acceptance condition), a violating execution is found. Otherwise the property is satisfied. Converting LTL properties to Büchi automata is well studied [Gastin and Oddoux, 2001] and all existing results extend to FTS model checking.

The algorithm proceeds in a similar fashion as the one for transition systems: find an accepting state and when one is found, find a path back to itself. The identification of accepting states is done by computing R. For each accepting state s, a new reachability relation R'_s is computed, with $Init'_s \triangleq \{(s, R(s))\}$ and $Post'_s \triangleq Post$. The violating products (or an empty set, if s could not be reached from itself) are then given by $R'_s(s)$.

This computation can be implemented in several ways. Of course, one could compute the fixed points directly as described above. This yields an algorithm that is quadratic in the number of states. A more efficient approach, proposed in [Courcoubetis et al., 1992], is to perform a *nested DFS*: the outer DFS identifies all accepting states. For each one, in postorder, an *inner* DFS is launched that tries to find a path back to itself. Because the states are explored starting with the last, each inner DFS can ignore the states that were already visited by previous inner DFSs [Courcoubetis et al., 1992]. This results in an algorithm that is *linear* in the number of states.

This optimisation can be used for FTS model checking as well. Basically, it corresponds to using the reachability relation R' for all inner DFSs, instead of computing it from scratch for every state. This yields the following algorithms.

Algorithm 6.29 ($Mc_{LTL}(\phi, fts)$)**.** *First, compute* $fts \otimes a_{\neg\phi}$*. Compute* R *with a DFS, the* outer *DFS, and each time an accepting state* s *is popped from the execution stack, compute* R' *with a DFS starting with* $(s, R(s))$*, the* inner *DFS. As soon as it reaches a state* s' *that is on the execution stack of the outer DFS, return* 0*, a counterexample and the set of violating products* $R'(s')$*. When the maximal* R' *is computed and no such state was found, continue with the outer DFS. When the maximal* R *is computed, return* 1*.*

Algorithm 6.30 ($ExtMc_{LTL}(\phi, fts)$)**.** *First, compute* $fts \otimes a_{\neg\phi}$*. Maintain a set of counterexamples* c *while computing the maximal* R*. Follow the algorithm for* $Mc(\phi, fts)$ *and each time a violation is found,* $(e, R'(s'))$ *with the counterexample* e *and the set of bad products* $R'(s')$ *is added to* c*. When the algorithm finishes and* c *is empty, return* 1*. Otherwise, return* 0 *and* c*.*

The optimisation relies on the assumption that states found during the earlier inner searches cannot lead back to the source state of the current inner search. The following theorem establishes that this is also a safe assumption in our case.

Theorem 6.31. *When a reachable accepting state is on a cycle for some non-empty set of products, then the algorithm will return* 0*.*

Proof. This is analogous to [Courcoubetis et al., 1992]. Let s (with products px) be the deepest reachable accepting state that is on a cycle in products px_{cycle}.

Let π be an execution from s back to itself that exists in products px_{cycle}. The central observation is that no state of π can be reached from a deeper accepting state, s' (with products px'). If this were the case, then s' could also reach s in products px_{cycle}, meaning that s would be deeper than s', which is not the case. Hence, for every state s_{cycle} in π, $px_{cycle} \cap R'(s_{cycle}) = \emptyset$ at the time the inner DFS starts, and thus the path π back to s will be found. □

6.5 Optimisations

An important optimisation concerns the algorithms for $ExtMc$, which compute maximal reachability relations. After having found a violating state, those algorithms will continue exploration until a maximal reachability relation is computed. Since the purpose of the algorithm is just to identify all violating products, it can ignore products that are already known to violate. Formally, given the set of counterexamples c maintained by the algorithm, the set of violating products is $px_{bad} = \bigcup_{(px,e)\in c} px$. Any state s with products $px \subseteq px_{bad}$ can be ignored, for all violations that would be found through s would be for products that are already known to violate (by Theorems 6.21 and 6.25).

In a DFS procedure, such as `Reachables`, this can be achieved by filtering out such states as part of the calculation of new. However, this can only eliminate newly discovered states, not those that are already on the stack. E.g., when a violation is found for products px, and the top i states on the stack are reachable only in products px, then they can all be popped immediately, since each attempt at exploring them would yield an empty set new. Filtering out states on the stack can be done efficiently by popping off elements (s, px) until $px \not\subseteq px_{bad}$. Following Theorem 6.21, the remaining states have sets of products that are all greater than px_{bad}.

A natural extension of this is to stop the search once all valid products are found to be violating. Also note that checking whether a new state is already known to be bad, and checking whether it is reachable in a valid product can be combined into a single check. In SNIP, this optimisation is implemented by adding a constraint to the formula representing the FD.

While these optimisations have clear benefits in all cases, we can also envision some optimisations that are more speculative (which we wave not yet implemented in any of our tools). They are based on the observation, from Theorem 6.21, that the sets of products shrink along an execution. Assume that two disjoint executions π_1 and π_2 lead to the same bad state. Further, assume that π_1 is longer than π_2. It is likely that the set of products in which π_1 exists is smaller than the set of products in which π_2 exists. Intuitively, the longer the path, the higher the likelihood to encounter new features, and thus shrink the set of products for which the path exists. If the algorithm explores π_1 first, it is likely to find a larger set of violating products, thus excluding a larger set of products from future searches, being more efficient.

A BFS will reach each state in the shortest possible path. Given the previous

observation, a BFS should thus have an advantage over a DFS when computing a maximal reachability relation for $ExtMc$. On the other hand, with a DFS, the algorithm for detecting cycles can be realised in a runtime that is linear in the number of states. A disadvantage of a BFS is the necessity to explore many states if the violating states are deep in the state space. It might outweigh the advantage when computing Mc. The question of the better exploration method is thus still left open. However, there are indications that BFS might be useful in FTS model checking, at least for reachability properties.

Another optimisation based on the same premise, but which could also work for a DFS based search, would be to fire the transitions leaving a state by starting with the least discriminating one. For a state s with products px and outgoing transitions $t_1, \ldots, t_k$, the target states would be reachable in $px \cap [\![\gamma(t_1)]\!], \ldots, px \cap [\![\gamma(t_k)]\!]$. These sets can be partially ordered by inclusion, and the maximal elements correspond to the least discriminating transitions. Assuming that all transitions lead to a bad state, firing the least discriminating ones first is likely to result in a larger set of violating products than first firing the other transitions. A practical obstacle to this optimisation method is the calculation of the order. So far, it has not been implemented.

6.6 Algorithmic complexity

An overview of the complexity results is given in Table 6.1. In the following, fts denotes the FTS under verification, n its number of features, and ϕ the LTL property. The size of an FTS is defined in Definition 4.14.

We first discuss the algorithmic complexity of the naïve Algorithms 6.1 and 6.2. The time complexity of both algorithms is identical: iterating through the set of products takes $O(2^n)$, projection can be computed in $O(|expr|)$, and model checking a single transition system takes $O(|fts|)$ for a reachability property and $O(2^{|\phi|}|fts|)$ for an LTL property (see, e.g., [Baier and Katoen, 2008]). Similarly for the space complexity. An important observation is that the naïve algorithms never have to maintain a structure of size $O(2^n)$. The iteration through the set of products can be achieved by generating a potential product, testing whether it is valid, and then performing model checking, all in $O(|fts|)$ space for reachability and in $O(2^{|\phi|}|fts|)$ for LTL properties.

The time complexity of `Reachables` is $O(4^n|fts|)$. In the worst case, the algorithm will visit every state for every product, i.e., the loop at line 7 will be executed $O(2^n|fts|)$ times. Calculating new at line 9 is then $O(2^n)$. Observe that the calculation at line 9 is $O(4^n)$ if taken in isolation. It requires set intersection and subset relation tests, which are $O(4^n)$ in all three encodings (explicit, rf/ef and Boolean function). It can be shown, however, that one member taking part in these operations is inversely proportional in size to the number of iterations in the loop at line 7. If the loop visits $O(2^{n-k})$ products with $k \in [0..n]$, then line 9 is $O(2^{n+k})$; e.g., when $|[\![\gamma(s \xrightarrow{\alpha} s')]\!]| = O(2^n)$ then $|px| = O(2^k)$. Note that the procedure requires a preprocessing step that

Table 6.1: Algorithmic complexity of reachability and of fLTL model checking.

Reachability		Time	Space
Naïve algorithm	Mc	$O(2^n\|fts\|)$	$O(\|fts\|)$
	$ExtMc$	$O(2^n\|fts\|)$	$O(\|fts\|)$
FTS algorithm	Mc	$O(4^n\|fts\|)$	$O(2^n\|S\| + \|fts\|))$
	$ExtMc$	$O(4^n\|fts\|)$	$O(2^n\|S\| + \|fts\|))$
fLTL		Time	Space
Naïve algorithm	Mc	$O(2^n 2^{\|\phi\|}\|fts\|))$	$O(2^{\|\phi\|}\|fts\|))$
	$ExtMc$	$O(2^n 2^{\|\phi\|}\|fts\|))$	$O(2^{\|\phi\|}\|fts\|))$
FTS algorithm	Mc	$O(4^n 2^{\|\phi\|}\|fts\|)$	$O(2^n 2^{\|\phi\|}\|S\| + 2^{\|\phi\|}\|fts\|))$
	$ExtMc$	$O(4^n 2^{\|\phi\|}\|fts\|)$	$O(2^n 2^{\|\phi\|}\|S\| + 2^{\|\phi\|}\|fts\|))$

converts the FD as well as all feature expressions into the chosen encoding. This step is $O(2^n|d| + 2^n|expr|)$, i.e., less than $O(4^n|fts|)$. Our computation assumes that insertion and lookup in the reachability relation is implemented in linear time, e.g., using hash tables as in SNIP. In [Classen et al., 2010b], we reported a complexity of $O(9^n|fts|)$, based on the theoretical maximum size of the rf/ef encoding being $O(3^n)$. Any efficient implementation of this encoding, however, will have at most $O(2^n)$ couples. The space complexity of `Reachables` is linear in the size of the state space and exponential in the number of features.

The algorithmic complexity of FTS LTL model checking, Algorithms 6.29 and 6.30, can be derived from this. The Büchi automaton corresponding to an LTL property ϕ is of size $O(2^{|\phi|})$. Its synchronous product with the FTS yields an FTS of size $O(2^{|\phi|}|fts|)$. The double DFS, as shown in Theorem 6.31, is of the same complexity as `Reachables`.

With regards to computational complexity, the naïve algorithms seem to be superior to the FTS algorithms. The higher exponential factor in the FTS algorithms reflects the costs of the symbolic encodings. However, the symbolic encodings were chosen explicitly in order to yield *concise* representations for sets of products. Therefore, we believe that this factor is not an indicator for performance in practice. In contrast, the naïve $ExtMc$ algorithm cannot avoid the exponential factor. It has to iterate through all products. This also holds for the naïve Mc algorithm, in case a property is satisfied by all products. The naïve Mc algorithm only works well when the relative number of violating products is high, as this raises the probability of quickly finding a violating product. Furthermore, an important difference between the naïve Mc algorithm and the FTS algorithm is that the former only returns a single violating product. The FTS Mc algorithm, in contrast, can return a set of

products (not necessarily complete) characterised by the interacting features.

Another measure of efficiency, independent from algorithmic complexity, is the reduction in state space achieved by our algorithms. It is clear that the naïve algorithms, when not using other optimisation techniques such as partial order reduction, will achieve no reduction of the state space. State space reduction is the purpose of our FTS algorithm, and thus an important measure of success. In this regard, we know that the FTS algorithm for $ExtMc$ will explore *at most* the number of states explored by the naïve algorithm. The experiments in Chapter 9 will shed more light on this.

We conclude by studying the complexity of the decision problems.

Definition 6.32 (Reachability)**.** *Given an FTS and one of its states, is there a product in which it is reachable?*

Theorem 6.33. Reachability *for FTS is NP-Complete.*

Proof. Reduce SAT to Reachability: each variable of SAT is a feature. The FTS has two states s_1 and s_2 with $I = \{s_1\}$. A transition labelled with the SAT expression leads from s_1 to s_2. The FD is so that it allows any product. SAT now corresponds to Reachability of s_2. This reduction is in constant time and space. The reverse reduction is also possible; it is well known that reachability and FDs can be encoded with SAT. □

By comparison, Reachability in transition systems is NL-Complete [Papadimitriou, 1994]. This shows that the succinctness from allowing Boolean expressions (or priorities, or just FDs, by Theorem 4.27) comes at a cost.

For both logics $L \in \{LTL, CTL\}$, the problems Mc_L and ExtMc_L are function problems [Papadimitriou, 1994]. To compare them to the LTL model checking problem for transition systems, we define Mc_L (D) as the decision problem corresponding to Mc_L and ExtMc_L.

Definition 6.34 (Mc_L (D))**.** *Given an FTS and a property in the logic L ($\in \{LTL, CTL\}$), is there a product whose transition system violates the property?*

Theorem 6.35. Mc_{LTL} (D) *is PSPACE-Complete.*

Proof. LTL model checking can be reduced to Mc_{LTL} (D); it is thus PSPACE-Hard. Mc_{LTL} (D) can be computed in NPSPACE (hence PSPACE) by guessing a product, computing its transition system, and then following [Sistla and Clarke, 1985]. □

In contrast to Reachability, the FTS model checking problem is not inherently harder than the one for transition systems. The complexity of the function problems is derived from this. Interestingly, both are of equal complexity.

Theorem 6.36. Mc_{LTL} *and* ExtMc_{LTL} *are FPSPACE-Complete.*

Proof. A procedure for MC_{LTL} and EXTMC_{LTL} in polynomial space is given by the naïve algorithm, when $SingleMc$ is computed following [Sistla and Clarke, 1985]. They are FPSPACE-Hard as MC_{LTL} (D) can be reduced to both MC_{LTL} and EXTMC_{LTL}. □

6.7 Conclusion

We just presented and studied a series of semi-symbolic model checking algorithms for FTS. The algorithms solve the decision problems defined in Chapter 5 for fLTL, and for deadlock checking. We also gave two naïve solutions to the same problems, which will serve as benchmark baselines in all the experiments we conduct.

The algorithms are semi-symbolic, because states are explored explicitly one by one, but products are represented symbolically. We presented two symbolic data structures: the rf/ef encoding, and the Boolean function encoding. In the following chapter, we will give a fully symbolic model checking algorithm, this time for fCTL. This will allow us to clarify the relation between FTS model checking and classical model checking.

Chapter 7

Symbolic Algorithms for FTS Model Checking

> *Society in every state is a blessing, but government even in its best state is but a necessary evil.*
>
> Thomas Paine, *Common Sense*, 1776

So far, we have introduced FTS, and model checking algorithms that search the state space with a depth-first search. A different kind of algorithm commonly used in symbolic model checking, relies on a fixed-point computation implemented on symbolic data structures, which roughly corresponds to a breadth-first search. In this chapter, we show that the fundamental principles for computing reachability in FTS also apply to this kind of algorithm, and use it for fCTL model checking. The main motivation for this that symbolic algorithms can, to some extent, address the state explosion problem and allow to verify large state spaces [McMillan, 1993].

The chapter will proceed as follows. In Section 7.1, we introduce a semi-symbolic fCTL model checking algorithm, using the principles established in the previous chapter. We then convert this algorithm into a fully symbolic algorithm in Section 7.2. In Section 7.3, we show how this algorithm can, in part, be reduced to classical symbolic model checking. In Section 7.4 we study its computational complexity, before concluding in Section 7.5.

7.1 Introduction

The algorithms presented in the previous section enumerate and visit system states *one by one*. Their aim is to mitigate the additional complexity that is due to the use of features in FTS. They still face the state explosion problem as they do visit all states of the system one by one. An existing solution to this problem in single system model checking is symbolic model checking, that

is, the use of symbolic representations of the state space (see Section 2.4.1). In this chapter, we combine FTS and symbolic model checking to tackle both the aforementioned sources of complexity at once.

Symbolic model checking algorithms are based on fixed-point computations, making them rather different from the depth-first search approach of the previous chapter. As CTL (and hence fCTL) lends itself well to this kind of algorithm, we focus here on fCTL. As an introduction, let us briefly give an overview of fCTL model checking for FTS, independent of the encoding used.

Following Theorem 6.20, fCTL model checking can be reduced to CTL model checking of an FTS with a modified FD. In the following, we thus just consider CTL. The model checking algorithm for CTL is based on the recursive computation of *satisfaction sets* along the parse tree of the formula (see also Section 2.3). A satisfaction set is a set of states that satisfy a particular subformula. A full algorithm for CTL model checking of FTS is given by the parse-tree computation and a recursive definition of the satisfaction sets.

The principles established so far for FTS model checking can be applied to this procedure, too: a satisfaction set has to keep track of states as well as the products in which they satisfy the formula. Its structure is thus the same as that of the reachability relations of Section 6.2.

Definition 7.1. *For an FTS fts and a CTL formula ϕ, a satisfaction set is a total function, $Sat(\phi) : S \to \mathcal{PP}(N)$, so that $\forall s \in S, p \in Sat(\phi)(s) \bullet fts_{|p}, s \models \phi$.*

Another point in which the CTL algorithm differs from the automata-based LTL algorithm is that it uses backward instead of forward searches. However, the execution method from Section 6.2 works both ways. Instead of the *Post* function, calculating successors, we use *Pre*, calculating predecessors.

Definition 7.2. *The* predecessors *of a state $s \in S$ reachable by products in $px \in \mathcal{PP}(N)$ are*

$$Pre(s, px) \triangleq \left\{(s', px') \mid s' \xrightarrow{\alpha} s \in trans \ \wedge px' = px \cap [\![\gamma(s' \xrightarrow{\alpha} s)]\!]\right\}$$

Similar to Post, Pre can easily be extended to reachability relations, where $Pre(R)$ stands for $\bigcup_{s \in S} Pre(s, R(s))$ followed by compact.

For instance, consider state ① of the vending machine FTS in Figure 4.2. Its predecessors in the product with all but the *FreeDrinks* feature, i.e., $Pre(③, \{v, b\ s, t\ c,\ cur, eur\})$, are states ④ and ⑨. Its predecessors in a product *with* the *FreeDrinks* feature, e.g., $Pre(③, \{v, f, s, cur, eur\})$, are states ⑦ and ⑨. This might seem incorrect, as state ⑨ is not supposed to be reachable in these products. However, the meaning of $(s, px) \in Pre(s', px')$ is that s' (or state ①) can be reached *from* s (or from state ⑨) in products px. Whether state ⑨ is reachable in products px can only be determined by continuing the backwards computation to the initial state, state ①. During this computation, the predecessors of state ⑧ are calculated, at which point the set of products is shrunken to those in $[\![\neg f]\!]$, the label on ⑦ $\xrightarrow{open}$ ⑧. The result will thus be that state ⑨ is not reachable in products with the *FreeDrinks* feature; as expected.

To complete the model checking algorithm for fCTL, it is sufficient to give a recursive definition of $Sat(\phi)$. The satisfaction sets of state formulae are recursively defined as follows.

Definition 7.3. *CTL state formulae satisfaction sets, $s \in S$:*

$$\begin{aligned} Sat(true)(s) &= [\![d]\!]_{FD} \\ Sat(a)(s) &= [\![d]\!]_{FD} \ if \ a \in L(s), \quad \emptyset \ otherwise \\ Sat(\phi_1 \wedge \phi_2)(s) &= Sat(\phi_1)(s) \cap Sat(\phi_2)(s) \\ Sat(\neg\phi)(s) &= [\![d]\!]_{FD} \setminus Sat(\phi)(s) \end{aligned}$$

The two first rules mimic the initial states from Definition 6.4: seeding the algorithm with the set of valid products. The rules for conjunction and negation should be clear. Note that this algorithm does not consider the symbolic set encodings. They were discussed at length before and can be applied here as well. Depending on the type of symbolic set used, it might be more efficient to use approximated product sets as discussed in Section 6.2.3, i.e., seeding the algorithm with $\mathcal{PP}(N)$ instead of $[\![d]\!]_{FD}$.

With the definition of Pre, computing $E\bigcirc$ is rather straightforward:

Definition 7.4. $Sat(E \bigcirc \phi) = Pre(Sat(\phi))$.

That is, if a state s satisfies ϕ for a set of products px, then all its predecessors s' with $s' \xrightarrow{\alpha} s$ satisfy $E \bigcirc \phi$ in products $px \cap \gamma(s' \xrightarrow{\alpha} s)$.

The computations of EU and $E\Box$ are based on fixed point algorithms. In standard CTL model checking, $E(\phi_1 U \phi_2)$ is characterised by the least fixed point: $\mu T \bullet \phi_2 \vee (\phi_1 \wedge E \bigcirc \ T)$. Basically, any state that satisfies ϕ_2 satisfies $E(\phi_1 U \phi_2)$, and so do all its predecessors if they also satisfy ϕ_1. The corresponding algorithm therefore starts with the states satisfying ϕ_2 and then searches backwards for all predecessors satisfying ϕ_1. Of course, the predecessors only exist in certain products, say px_1, and ϕ_1 is only satisfied in certain products, say px_2, which have to be intersected.

Definition 7.5. $Sat(E(\phi_1 U \phi_2)) = T_i \bullet T_i = T_{i+1}$,

$$\begin{aligned} where \quad T_0 &= Sat(\phi_2) \\ T_{i+1} &= T_i \cup \{(s, px_1 \cap px_2) \mid (s, px_1) \in Pre(T_i) \\ &\qquad \wedge \ (s, px_2) \in Sat(\phi_1) \\ &\qquad \wedge \ (\nexists (s', px') \in T_i \\ &\qquad\quad \bullet \ s = s' \wedge (px_1 \cap px_2 \subseteq px'))\} \end{aligned}$$

Another way to look at this procedure is as a backwards computation of a maximal reachability relation, or satisfaction set, from $Sat(\phi_2)$ that only considers states in $Sat(\phi_1)$. Of course, an efficient implementation of such an algorithm does not compute $Pre(T_i)$ for the whole set of T_i, but rather for the elements that were added in the previous iteration. The smaller this delta, the faster

the computation. This is why the last condition in the calculation makes sure that the algorithm does not reconsider states that were already explored.

The algorithm for $E\Box$ is similar to the one for EU, except that a greatest fixed point has to be computed. $E\Box\phi$ is characterised by: $\nu T \bullet \phi \wedge E \bigcirc T$. So basically, any state satisfies T if one of its successors also satisfies T, and the same for its successors, and so on. The algorithm thus starts with $Sat(\phi)$, and progressively shrinks it by removing states that are not in $Pre(Sat(\phi))$.

Definition 7.6. $Sat(E\Box\phi)) = T_i \bullet T_i = T_{i+1}$,

$$\begin{aligned} where \quad T_0 &= Sat(\phi) \\ T_{i+1} &= \{(s, px_1 \cap px_2) \mid (s, px_1) \in T_i \\ &\qquad \wedge\ \exists (s, px_2) \in Pre(T_i)\} \end{aligned}$$

Given these definitions, an algorithm for model checking an FTS fts against an fCTL property $[\chi]\phi$ is simply to transform the FTS into fts', which contains the additional FD constraint χ (Theorem 6.20), and then compute $Sat(\phi)$ recursively. This yields the set of products that satisfy the property. If $Sat(\phi) = \emptyset$, the property is violated.

An interesting difference between this fixed-point based algorithm and the DFS-based algorithm of Section 6.4.2 is that the fixed-point algorithm always computes maximal reachability relations. This means that it will always produce an answer to both decision problems Mc and $ExtMc$.

7.2 Symbolic model checking of fCTL properties

Let us now turn the semi-symbolic algorithm given in Section 7.1 into a fully symbolic algorithm. In the symbolic setting, sets of states and the transition relation are encoded directly with their characteristic functions. As we already said, characteristic functions can be represented by BDDs.

We proceed in two steps. First, we describe a symbolic encoding for FTS, before we proceed to the algorithms.

7.2.1 Encoding FTS symbolically

We use the same notation as the one defined in Section 2.4.1. Recall that we assume the existence of a binary encoding of states, that is, a function $enc : S \to \{0,1\}^k$, where k is chosen large enough to encode all states. Given a product p, we also use the notation $enc(p)$ to denote the encoded product. With this encoding, $\{0,1\}^k$ implicitly denotes the sets of all (encoded) states and $\{0,1\}^n$ the set of all (encoded) products.

With the notation in place, we now show how an FTS can be encoded symbolically. The set of states is represented by a Boolean function χ_S and the set of initial states by χ_I. As usual, the labelling of states with atomic

propositions L is represented by recording for each atomic proposition $a \in AP$ the set of states χ_a that are labeled by the proposition:

$$\chi_a(\overline{s}) : \{0,1\}^k \to \{0,1\} \bullet \chi_a(enc(s)) = 1 \iff a \in L(s).$$

The transition relation is represented by a function that takes two encoded states (start and end) and an encoded product, and returns 1 iff some transition $s \xrightarrow{\alpha} s'$ exists in the product. The feature expression of a transition is implicitly embedded in the encoding.[1] Formally,

$$\chi_{trans}(\overline{s}, \overline{s}', \overline{p}) : \{0,1\}^k \times \{0,1\}^k \times \{0,1\}^n \to \{0,1\},$$

such that $\chi_{trans}(enc(s), enc(s'), enc(p)) = 1$ iff some $s \xrightarrow{\alpha} s'$ in $fts_{|p}$. The feature expression on the transition is the cofactor for the encoding of both states:

$$\chi_{\bigvee_\alpha \gamma(s \xrightarrow{\alpha} s')}(\overline{p}) : \{0,1\}^n \to \{0,1\} \triangleq \chi_{trans[\overline{s} \leftarrow enc(s), \overline{s}' \leftarrow enc(s')]}(\overline{p}).$$

Transitions with the same start and end states are implicitly merged (with a disjunction of their Boolean function labels).

This yields a symbolic encoding for FTS covering all of Definition 4.1.

Satisfaction sets are also encoded by their characteristic function,

$$\chi_{Sat(\phi)}(\overline{s}, \overline{p}) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\},$$

so that $\chi_{Sat(\phi)}(enc(s), enc(p)) = 1$ iff $fts_{|p}, s \models \phi$.

The heart of our fixed point algorithms is the predecessor calculation. All the information it needs is contained in the transition relation, and calculating the predecessors of a single sate in a single product amounts to instantiating two arguments of the characteristic function of the transition relation:

$$\chi_{Pre(s,p)}(\overline{x}) : \{0,1\}^k \to \{0,1\} \triangleq \chi_{trans[\overline{s}' \leftarrow enc(s), \overline{p} \leftarrow enc(p)]}(\overline{x}),$$

i.e., the cofactor of the transition relation χ_{trans} for the product p and state s.

Of course, this calculation has to be very efficient since it is executed at each step of the algorithm. Therefore, the computation cannot rely on single state/product predecessor computations to accomplish this. We rather need to compute it on *a set* of such couples, generally a satisfaction set of some property ϕ. This leads us to define the operator $SetPre$ as follows.

Definition 7.7.
$$\chi_{SetPre(Sat(\phi))}(\overline{s}, \overline{p}) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\}$$
$$\triangleq \exists \overline{s}' \bullet \chi_{Sat(\phi)}(\overline{s}', \overline{p}) \wedge \chi_{trans}(\overline{s}, \overline{s}', \overline{p}).$$

Intuitively, $SetPre(Sat(\phi))$ is the set of couples (s, p) such that there exists a state s' that satisfies ϕ in product p and to which s has a transition in product p. Since the operation is computed on the symbolic encoding of the sets, it does not consider states or products individually.

[1] Which is natural as both the transitions and the feature expression are Boolean functions.

7.2.2 Symbolic algorithms

Having the fundamentals covered, we can proceed to the model checking algorithms. As before, we rely on Theorem 6.20 to reduce fCTL to CTL by transforming the FD, and define the satisfaction set calculation for CTL.

The satisfaction sets for state formulae are again rather straightforward:

Definition 7.8. *CTL state formulae satisfaction sets:*

$$\begin{aligned}
\chi_{Sat(true)}(\overline{s},\overline{p}) &= 1\\
\chi_{Sat(a)}(\overline{s},\overline{p}) &= \chi_a(\overline{s})\\
\chi_{Sat(\phi_1\wedge\phi_2)}(\overline{s},\overline{p}) &= \chi_{Sat(\phi_1)}(\overline{s},\overline{p}) \wedge \chi_{Sat(\phi_2)}(\overline{s},\overline{p})\\
\chi_{Sat(\neg\phi)}(\overline{s},\overline{p}) &= \neg\chi_{Sat(\phi_1)}(\overline{s},\overline{p})
\end{aligned}$$

Note that this is not equivalent to the way we defined the satisfaction sets in Definition 7.3. Here, we do not seed the initial states with the valid products of the FD. We follow the approach discussed in Section 6.2.3. In consequence, we need another way to make sure that only valid products are considered. We do this as part of the last step of the algorithm.

Let us now define the satisfaction sets for CTL path formulae. These can be obtained almost immediately from the definitions in Section 7.1. To obtain $Sat(E \bigcirc \phi)$ it is sufficient to calculate the predecessors of $Sat(\phi)$, that is, to apply the $SetPre$ operator from Definition 7.7 to $Sat(\phi)$.

Definition 7.9. $\chi_{Sat(E\bigcirc\phi)}(\overline{s},\overline{p}) \triangleq SetPre(Sat(\phi))(\overline{s},\overline{p})$

The algorithm for $Sat(E\phi_1 U \phi_2)$ proceeds in the same way as the semi-symbolic algorithm of Definition 7.5. It starts with the states and products satisfying ϕ_2 and works backwards, searching for predecessors which satisfy ϕ_1.

Definition 7.10. $\chi_{Sat(E(\phi_1 U \phi_2)} = \chi_{T_i} \bullet \chi_{T_i} = \chi_{T_{i+1}}$,

$$\begin{aligned}
\textit{where}\quad \chi_{T_0}(\overline{s},\overline{p}) &= \chi_{Sat(\phi_2)}(\overline{s},\overline{p})\\
\chi_{T_{i+1}}(\overline{s},\overline{p}) &= \chi_{T_i}(\overline{s},\overline{p}) \vee \big(\chi_{Sat(\phi_1)}(\overline{s},\overline{p})\\
&\qquad \wedge \chi_{SetPre(T_i)}(\overline{s},\overline{p})\\
&\qquad \wedge \neg\chi_{T_i}(\overline{s},\overline{p})\big)
\end{aligned}$$

In each iteration, we add the states $(\overline{s},\overline{p})$ that satisfy ϕ_1, i.e. $\chi_{Sat(\phi_1)}(\overline{s},\overline{p})$, and are predecessors of a state in T_i, i.e. $\chi_{SetPre(T_i)}(\overline{s},\overline{p})$. An optimisation known in current CTL algorithms, and crucial here, is to only add states that were not already in T_i, i.e. $\neg\chi_{T_i}(\overline{s},\overline{p})$. Otherwise, previously visited states would be re-visited, which would be inefficient due to the added feature variables.

The algorithm for $E\Box\phi$ starts off with all states and products satisfying ϕ and progressively shrinks this set by removing states and products whose successors do not satisfy ϕ.

Definition 7.11. $\chi_{Sat(E\Box\phi)} = \chi_{T_i} \bullet \chi_{T_i} = \chi_{T_{i+1}}$,

$$\begin{aligned} \textit{where} \quad \chi_{T_0}(\overline{s},\overline{p}) &= \chi_{Sat(\phi)}(\overline{s},\overline{p}) \\ \chi_{T_{i+1}}(\overline{s},\overline{p}) &= \chi_{T_i}(\overline{s},\overline{p}) \wedge \chi_{SetPre(T_i)}(\overline{s},\overline{p}) \end{aligned}$$

The final step of the model checking algorithm is to check whether all initial states satisfy ϕ, and for which products they do. Given $\chi_{Sat(\phi)}(\overline{s},\overline{p})$, the set of products that violate ϕ is obtained by intersecting the complement of $Sat(\phi)$ with the set of initial states, and then projecting on the state variables. This leaves a Boolean function over the feature variables characterising the set of violating products. This set has to be intersected with the set of valid products, unless the calculation was seeded with the valid products.

Definition 7.12. *The set of products $\chi_{px_{bad}}$ violating a CTL property ϕ is* $\chi_{px_{bad}}(\overline{p}) = \exists\overline{s} \bullet \chi_I(\overline{s}) \wedge \neg\chi_{Sat(\phi)}(\overline{s},\overline{p}) \wedge \mathbb{B}(d)(\overline{p})$.

If $\chi_{px_{bad}} = 0$, the property is satisfied by all products.

The algorithms for calculating satisfaction sets combined with the parse tree computation lead to a complete model checking algorithm for CTL, and hence fCTL, over FTS.

Algorithm 7.13 ($Mc_{CTL}(\phi, fts)$, $ExtMc_{CTL}(\phi, fts)$)**.** *Compute $Sat(\phi)$ recursively along the parse tree of ϕ following Definitions 7.8, 7.9, 7.10 and 7.11. Calculate $\chi_{px_{bad}}$ following Definition 7.12. If $\chi_{px_{bad}} = 0$, return 1. Otherwise, return 0 and $\chi_{px_{bad}}$*

7.3 Reducing fCTL model checking to classical model checking

A closer look at the algorithms for calculating satisfaction sets in the previous section reveals that they do not treat feature and state variables differently. This means that it might be relatively easy to reduce fCTL model checking to classical symbolic model checking. However, in classical symbolic model checking, satisfaction sets only refer to states, not to features. A way to achieve this is to change our symbolic encoding of FTS slightly, by moving the features from the transitions to the states:

$$\chi_S(\overline{s},\overline{p}) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\} \bullet \chi_S(enc(s), enc(p)) = 1 \iff s \in S.$$

That is, the features are parameters of the characteristic function of the set of states, but their value does not matter. The initial states and the sets of states that capture the action labelling are defined similarly:

$$\chi_I(\overline{s},\overline{p}) : \{0,1\}^k \times \{0,1\}^n \to \{0,1\} \bullet \chi_S(enc(s), enc(p)) = 1 \iff s \in I.$$

$$\chi_a(\overline{s},\overline{p}) : \{0,1\}^k \to \{0,1\} \bullet \chi_a(enc(s)) = 1 \iff a \in L(s).$$

As features are now part of the states, and the symbolic transition relation is a function over two copies of the states, it now has two copies of the features instead of just one:

$$\chi_{trans}(\overline{s}, \overline{p}, \overline{s}', \overline{p}') : \{0,1\}^k \times \{0,1\}^n \times \{0,1\}^k \times \{0,1\}^n \rightarrow \{0,1\},$$

such that $\chi_{trans}(enc(s), enc(p), enc(s'), enc(p')) = 1$ iff some $s \xrightarrow{\alpha} s'$ in $fts_{|p}$ and $p = p'$. This version of the symbolic transition relation is only equivalent to the one of Section 7.2.1 if the feature variables are left unchanged by the transition relation. This is ensured by the last condition, $p = p'$.

Given that features are now part of the states, the characteristic function of the satisfaction sets keeps the same signature as before and the predecessor calculation becomes:

Definition 7.14. $\chi_{SetPre(Sat(\phi))}(\overline{s}, \overline{p}) \triangleq \exists \overline{s}' \bullet \chi_{Sat(\phi)}(\overline{s}', \overline{p}) \wedge \chi_{trans}(\overline{s}, \overline{p}, \overline{s}', \overline{p})$.

This definition coincides with the definition of the predecessors in standard symbolic CTL model checking algorithms, under the condition that the feature variables in a transition do not change. Furthermore, satisfaction sets in our algorithm now coincide with those in standard symbolic CTL model checking algorithms, and even their calculation is the same. By combining these observations we can, at least for the calculation of $Sat(\phi)$, reduce symbolic FTS model checking to symbolic model checking of specially crafted transition systems.

It is in the final step of the algorithm, i.e., checking whether all initial states satisfy the property, where feature variables are treated differently from the states. This step needs to be adapted as described in Definition 7.12, by quantifying away the state variables in order to obtain the set of violating products. If this is not done, the algorithm will just yield $false$ if there are violating products, without indicating which products are to blame.

The modified encoding has another advantage: it allows to express fCTL properties in CTL. This is due to the fact that feature variables are state variables that can be referenced in a property. A set of products χ_{px} can thus be expressed in the specification language, which means that the fCTL formula $[\chi_{px}]\phi$ can be translated to the CTL formula $(\chi_{px}) \implies \phi$.

7.4 Algorithmic complexity

An overview of the complexity results is given in Table 7.1. In this table, and in the following paragraphs, complexities are given wrt. an FTS fts (with S being the set of states and n the number of features) and an fCTL property $[\chi]\phi$, i.e., ϕ is a CTL property.

For reference, single-system CTL model checking of transition systems has a computational complexity of $O(|S|.|\phi|)$. Let us start with the algorithmic complexity of the naïve fCTL model checking algorithms, Algorithms 6.1 and 6.2 from the previous chapter. As for fLTL, the time complexity of both algorithms is the same: iterate through the set of products, $O(2^n)$, calculate projection,

Table 7.1: Algorithmic complexity of fCTL model checking.

fCTL		Time	Space
Naïve algorithm	Mc	$O(2^n\|\phi\|\|fts\|)$	$O(\|\phi\|\|fts\|)$
	$ExtMc$	$O(2^n\|\phi\|\|fts\|)$	$O(\|\phi\|\|fts\|)$
FTS algorithm	Mc	$O(2^n\|\phi\|\|fts\|)$	$O(\|\phi\|\|fts\|)$
	$ExtMc$	$O(2^n\|\phi\|\|fts\|)$	$O(\|\phi\|\|fts\|)$

$O(|expr|)$, and model check the transition system, $O(|\phi||fts|)$. This yields a total time complexity of $O(2^n|\phi||fts|)$ and space complexity of $O(|\phi||fts|)$.

The algorithmic complexity of Algorithm 7.13 for FTS CTL model checking is $O(|fts|.|\phi|.2^n)$. Basically, a satisfaction set is calculated for each node in the formula giving the factor $|\phi|$. This calculation is linear in the size of the state space for $Sat(1)$, $Sat(a)$, $Sat(\neg\phi)$, $Sat(\phi_1 \wedge \phi_2)$ and $Sat(E \bigcirc \phi)$. The fixed points of $Sat(E(\phi_1 U \phi_2))$ and $Sat(E\Box\phi)$ both take $O(|S|.2^n)$ since, in the worst case, they proceed monotonically through 2^n products for each state.

With regards to computational complexity, the naïve algorithm is equal to ours. This is consistent with the fact that FTS CTL model checking can be reduced to the classical symbolic CTL model checking algorithm, used by the naïve algorithm. The difference between both is that the naïve algorithm performs $O(2^n)$ model checks of models of size $O(|fts|)$, whereas our algorithm performs a single model check, of a model of size $O(2^n|fts|)$. Our hope is that the similarities between the $O(2^n)$ models will cause the BDDs of the single model to be smaller than the sum of the size of the smaller BDDs in the naïve algorithm. Furthermore, as variable orderings play a crucial role in the efficiency of BDD operations, our algorithm has the advantage of requiring only a single ordering (which can then be tuned).

Note that the additional exponential factor of our algorithm cannot be avoided unless model checking is restricted to models less powerful than FTS, as done in [Li et al., 2002b]. Also note that our algorithm remains linear in the size of the state space and is more efficient than the one presented in [Lauenroth et al., 2009]. More precisely, the latter is $O(|\phi|.|S|!) = O(|\phi|.|S|^{|S|})$, and the models it treats can be transformed to FTS in constant time. The algorithm of [Lauenroth et al., 2009] is thus in EXPTIME whereas ours is in E, i.e. DTIME($2^{O(x)}$), a class that *"captures a more benign aspect of exponential time"* [Papadimitriou, 1994]. Furthermore, it is important to note that in practice, the size of the state space is much larger than the number of features.

Le us conclude with a study of the complexity of the decision problems. Reachability and LTL model checking were studied in Section 6.6. Recall Definition 6.34, where we defined the decision problem MC_{CTL} (D), as the counterpart to the two function problems MC_{CTL} and ExtMC_{CTL}.

Theorem 7.15. MC_{CTL} (D) *is NP-Complete.*

Proof. This can be proven with the same method in which Theorem 6.33 was proven: reduce SAT to MC_{CTL} (D) and the other way around. □

By comparison, CTL model checking for transition systems is P-Complete [Schnoebelen, 2002]. This result is thus similar to the one for REACHABILITY.

Just as the decision problem MC (D) can be reduced to and from SAT, the function problem MC_{CTL} function problem can be reduced to and from FSAT. FSAT is the function problem that consists in finding a satisfying assignment for the variables of a SAT problem (rather than merely deciding satisfaction).

Theorem 7.16. MC_{CTL} *is FNP-Complete.*

Proof. It is easy to see that MC_{CTL} can be reduced to FSAT, by following the proof of Theorem 7.15, and reusing the variable assignment to produce the violating product. The reduction in the other direction is trivial. □

FNP is the class of function problems that can be solved by a non-deterministic Turing machine in polynomial time [Papadimitriou, 1994]. According to this result, the FTS model checking problem is inherently harder than the one for transition systems. This seems to contradict our earlier result, that fCTL model checking of FTS can be reduced to CTL model checking of transition systems. However, these transition systems are larger by a factor of $O(2^n)$.

Unlike EXTMC_{LTL} which was part of the same computational complexity class as MC_{LTL}, the EXTMC_{CTL} decision problem is not part of FNP. Since it needs to determine each violation, and moreover each valid product, it is part of the class of counting problems #P. It can be reduced to #SAT, the problem of identifying and counting all satisfying assignments for a SAT problem.

Theorem 7.17. EXTMC_{CTL} *is* #P*-Complete.*

Proof. The proof is similar to the preceding proofs. EXTMC_{CTL} can be reduced to #SAT, by following the proof of Theorem 7.15. It thus produces a list of violating products. The reduction in the other direction is trivial. □

These complexity results are somewhat misleading, because the state space is given as a set of states directly. In practice, the state space is defined by a set of variables, which means that its size is exponential in the number of variables. Since the number of variables is likely to be larger than the number of features, the size of the state space is likely to be much larger than 2^n, which would mean that $O(|S|) = O(2^n|S|)$. In practice, the CTL model checking problem for FTS is thus not inherently harder than the one for transition systems.

7.5 Conclusion

We presented algorithms for fCTL model checking of FTS. The algorithms were first given in a semi-symbolic form (comparable to the fLTL algorithms of the previous chapter), which was then transformed into a fully symbolic algorithm.

A change in the symbolic encoding of FTS allowed us to reduce the core of this algorithm to classical symbolic CTL model checking over transition systems.

This progression illustrates the relation of our semi-symbolic algorithm to existing symbolic model checking algorithms. Conceptually, features are a particular kind of variable in a symbolic model checking problem. The symbolic model checking of FTS is thus much closer to the classical symbolic model checking than the semi-symbolic algorithms of the previous section are to classical explicit model checking algorithms. Indeed, the algorithms of Chapter 6 are half-way between explicit and symbolic.

This concludes Part II of the thesis, the largely theoretical development of models and algorithms. In Part III, we show how this can be put into practice.

Part III

Implementation and Evaluation

Putting FTS into practice

> *Hofstadter's Law: It always takes longer than you expect, even when you take into account Hofstadter's Law.*
>
> Douglas Hofstadter, *Gödel, Escher, Bach*, 1979

The results presented so far are all foundational and theoretical. In the following two chapters, we present our efforts to put these results into practice, by implementing them as part of model checking tools. While implementing the algorithms lead by itself to new insights, these tools also allow us to evaluate the efficiency of our algorithms through experiments. In addition, with the development of the tools, we can assess the feasibility of our theories, and we can examine them from a different perspective.

Over the past years, a number of model checkers have been implemented. All of them are available (open source) at the FTS website [Classen, 2010b]. In this brief introduction to Part III, we provide an overview, and discuss the motivation and the various implementation choices.

On the use case of SPL model checking

Let us quickly recall to what extent the use case of SPL model checking differs from the one in model checking of single systems. This is one of the motivations for us to implement new model checking tools, or change (rather than reuse) existing model checkers.

The principal difference between SPL model checking and single systems model checking is the presence of *variability*. Variability in SPLs is typically expressed in terms of *features*. Since features serve as the central unit of difference in SPLs, it is imperative that model checking tools recognise them as a first-class concept, that is, inputs and outputs should be expressed in terms of features. There are, however, currently no model checkers that do this, except for those developed during the course of this thesis. Furthermore, FDs are commonly used to capture known dependency or exclusiveness relations of features. They have to be integrated into the model checking approach. Otherwise, the model checker might identify problems in products that are not valid in the first place, or discover information that is already known.

Having features as a first-class concept means that results of a model checker have to be provided in terms of features. This is not the case if just a single product is verified, or a list of products one by one (as done by the naïve algorithm). Either case yields information about specific *products*, which is limiting as problematic features cannot readily be inferred from violating products. This is not only limited as a verification result, but also inappropriate for the engineer who thinks in terms of features when specifying the model. Without knowing which features are responsible, it is much more difficult to locate an error, especially if it involves several interacting features. Extending classical model checkers to SPLs almost inevitably leads to this situation. The model checkers developed as part of the thesis are currently the only tools that present their results in terms of features.

In addition to these user interface considerations, the SPL model checking problems are slightly different from those addressed by classical model checkers. First, in SPL model checking, we distinguish two model checking problems, MC and EXTMC. While the first of these is rather similar to what is done in model checking of single systems, the second is a use case which is not supported by model checkers for single systems. Secondly, the logics fLTL and fCTL offer the ability to specify properties for a relevant subset of the valid products. Naturally, this is also not supported by existing tools.

Modelling language

Our first proof-of-concept implementation of the FTS algorithms was written in the functional programming language Haskell. The tool comes in the form of a library that can be loaded into a Haskell interpreter to be accessed through a command line interface, or compiled to perform verifications in batch mode. Using Haskell has some advantages, such as its pervasive use of lazy evaluation and the natural translation of mathematical formulae into program code. The tool implements the explicit LTL model checking algorithm of Chapter 6 using the rf/ef encoding. It interfaces with ltl2ba,[2] to automate the translation from LTL to Büchi automata, and uses Graphviz[3] to render FTS graphically. We conducted benchmarks that show that our algorithm is up to seven times, and in average three times, faster than the naïve algorithm.

The biggest inconvenient of this tool is that its models have to be specified directly in FTS. While it is not impossible to specify models in FTS, as the examples in Chapter 4 show, as soon as the complexity of the system grows it becomes very hard. This can be seen in the mine pump example of Section 4.5.2, where we use parallel composition even for the non-parallel parts of the system, in order to keep the size of the models reasonable. This is because FTS are not at the appropriate level of abstraction to be used for modelling. This is not a failure in the design of FTS. As we and others pointed out [Classen et al.,

[2] www.lsv.ens-cachan.fr/~gastin/ltl2ba
[3] www.graphviz.org

2010b, Kim et al., 2010], FTS are a foundational formalism. They describe semantics and are not meant to be used as a modelling language directly.

While the Haskell FTS library served its purpose as a proof-of-concept implementation of our algorithms, it can barely be used for larger benchmarks. Furthermore, its interface being essentially a set of Haskell functions, it is hard to learn or use by a third party. We thus decided to implement a new tool, with a focus on ease of use. A first step towards this goal is to define a high-level modelling languages on top of FTS.

As we have seen before (Section 1.3), there are two types of language in FOSD: those in which features are expressed in the form of annotations, and those in which features are independent modules that are composed with a base system. Both types have their benefits, and so we decided to use both of them. We thus propose two different languages. The first one, fSMV [Plath and Ryan, 2001], is a feature-oriented extension of the (Nu)SMV input language [McMillan, 1993, Cimatti et al., 2000, Cavada et al., 2004]. The other is fPromela, an extension of the Promela language from SPIN [Holzmann, 2004]. fSMV is based on superimposition: features are specified modularly as changes to be done to a base system. A product is constructed by *composition* of features. fPromela, in contrast, follows an annotative approach in which statements can be *guarded* by features. There, products are constructed by *pruning* non-selected features. This corresponds to the common way of implementing features in industrial SPLs (e.g., with `#ifdef`s [Kästner et al., 2008, Liebig et al., 2010], or other annotations [Boucher et al., 2010b, Kästner et al., 2008]).

Other implementation choices

The modelling language is one of many choices that have to be made when implementing model checking algorithms. Another important choice is the kind of algorithm to be used in the tool, i.e., either the semi-symbolic algorithms of Chapter 6 or the fully symbolic algorithms of Chapter 7. Both choices are closely linked, as models have to be translated into the format required by the algorithm. Finally, there is the choice of the logic, fLTL or fCTL.

As shown in Chapter 7, the symbolic FTS algorithm can be reduced to the classical symbolic model checking algorithm, requiring only a well isolated step to be changed. It therefore lends itself well to being implemented as part of an existing model checker. To change an existing explicit model checking algorithm into our semi-symbolic algorithm, in contrast, changes to almost every step of the algorithm would be required.

In consequence, we decided to implement fSMV as part of the NuSMV model checker [Cimatti et al., 2000, Cavada et al., 2004] with a fully symbolic algorithm for fCTL. In contrast, the model checker for fPromela, SNIP, uses the semi-symbolic fLTL algorithms and was implemented from scratch. An overview of the developed tools, and their key characteristics is given in Table 7.2. With this combination of tools, we cover all of the theory discussed so

far. In Chapter 8, we present fSMV and fNuSMV and in Chapter 9, fPromela and SNIP.

Table 7.2: FTS model checkers developed as part of the thesis.

	Haskell FTS Lib	**fNuSMV**	**SNIP**
Release date	September 2009	January 2010	November 2010
Language	FTS	fSMV	fPromela
– Style	Automata, expressed as Haskell data structures	Declarative, boolean transition relation is specified directly	Procedural, very intuitive
– Features	Annotation	Composition	Annotation
– Supports FDs	No	No[4]	Yes, TVL [Classen et al., 2011a]
Logic	LTL	fCTL	fLTL
Algorithm	Semi-symbolic	Symbolic	Semi-symbolic, on-the-fly
– Products	rf/ef	BDDs	BBDs
– Variables	Explicit	BDDs	Explicit
Platform	Haskell (Hugs or GHC)	NuSMV (C++)	C

[4]Note that it would be rather easy to implement this. The reason why this was not implemented is that, at the time, we did not have a suitable FD language.

Chapter 8

fNuSMV and fSMV

> *The hopes which inspire communism are, in the main, as admirable as those instilled by the Sermon on the Mount, but they are held as fanatically and are as likely to do as much harm.*
>
> Betrand Russel, *The Practice and Theory of Bolshevism*, 1920

In this chapter, we present fNuSMV, a model checker that implements the fully symbolic algorithms of Chapter 7. fNuSMV was the first model checker we developed after the experimental Haskell FTS library. It gave us access to existing models and the ability to model more substantial systems. As a result, we used it to conduct an experiment to assess the efficiency of the symbolic FTS algorithms. The modelling language used by our toolset is fSMV, a feature-oriented extension of the SMV language proposed in [Plath and Ryan, 2001].

After an introduction and overview in Section 8.1, we present fSMV, the specification language, in Section 8.2. In Section 8.3, we describe the toolset and discuss implementation details. In Section 8.4, we report on the experiment conducted with the toolset. We conclude in Section 8.5.

8.1 Introduction

As we have shown in Chapter 7, the fully symbolic model checking algorithm for FTS can largely be reduced to the classical algorithm for transition systems. To implement it, we decided to extend the state-of-the-art symbolic model checker NuSMV [Cimatti et al., 2000].[1] Thereby, we can take advantage of its existing infrastructure. We refer to the extended NuSMV as 'fNuSMV'.

The input language of NuSMV can be used as-is to create models that correspond to the symbolic encoding required for FTS model checking. However, to make modelling more intuitive, we reuse a language by Plath and Ryan [Plath and Ryan, 2001], which is specifically designed for specifying features. The

[1] http://nusmv.irst.itc.it

language, which we call 'fSMV', is a feature-oriented extension of the input language of NuSMV.[2] It is based on the compositional FOSD paradigm: features are specified independently, and a product is created by composition.

In addition to fNuSMV, the toolset comprises a script that implements the feature composition mechanism of fSMV [Plath and Ryan, 2001]. The common use case of the toolset takes as input a list of features specified in fSMV, a NuSMV model that represents the *base system* (i.e., the common core of all products), and one or more fCTL properties. First, the composition tool is used to compose the base system with a number of features. The result of this composition is then passed on to fNuSMV. To activate FTS model checking, the new command line parameter `-fbdd` has to be set, otherwise, fNuSMV will behave as NuSMV. For each violated property, it prints a Boolean expression characterising the products that violate the property. The modifications made to NuSMV are available as a patch for NuSMV 2.5.0 [Classen, 2010b].

8.2 The fSMV modelling language

In this section, we first provide a brief overview of the fSMV syntax. We then show that its models are in fact FTS and that it can hence serve as a high-level language for FTS. Finally, we discuss its expressiveness.

8.2.1 Syntax

Essentially, a NuSMV model consists of a set of variable declarations and a set of assignments. The variable declarations define the state space and the assignments define the transition relation. In each assignment, the value of a variable in the next state is defined in function of the variable values in the present state. For each variable, there can also be an assignment that defines its initial value. Alternatively, the value of a variable can be defined directly in function of the other variables. Modules can be used to encapsulate and factor out recurring elements. Henceforth, we will refer to this language as 'SMV'.

The typical example of an SMV model (taken from [Cavada et al., 2004]) is the following.

Listing 8.1: Controller base system.

```
1 MODULE main
2 VAR
3   request: boolean;
4   state:   {idle, busy};
5 ASSIGN
6   init(state) := idle;
7   next(state) := case state = idle & request: busy;
```

[2] More precisely, they used the earlier SMV model checker. The input language of NuSMV is almost identical.

```
8                    1: {idle, busy};
9                esac;
```

The above model describes a controller that is either idle or busy treating a request. The `VAR` section defines variables, and the `ASSIGN` section defines their values (and thereby the transition relation). Requests are modelled by the variable `request`. The absence of any assignments for this variable means that its value is chosen non-deterministically in each state, which models the fact that requests are controlled by the environment. The `state` variable represents the state of the controller. It is of an enumerated type. The `init` assignment defines its initial value (initially, the system is idle). The `next` assignment, defines the transition relation: when the controller is idle and there is a request, it will treat the request and be busy (line 7), otherwise, it may continue to be busy for a while and return to idle once the request is treated (line 8). A `case` statement is a conditional expression where each line is of the form `condition: value;`. The conditions are evaluated in the order in which they are specified, and the value of the first true condition is taken. The `1` at line 8 means *true*, i.e., it acts like an 'else' in programming languages such as C or Java. The `{idle, busy};` at line 8 is the non-deterministic choice between those values.

The semantics of such a model is a transition system. Its state space is the product of the domains of the variables. The state space can be narrowed with the keyword `INVAR`, which is used to define a constraint which has to hold in all states. The initial states of the transition system are those whose variable values satisfy the `init` statements. Similarly, there is a transition between a pair of states when their variable values satisfy the `next` statements. SMV has other ways to define the transition relation. In this work, we restrict ourselves to the `ASSIGN` syntax. In addition to the transition system, an SMV model also contains CTL properties. These are called 'specifications' and follow the `SPEC` keyword.

Of course, NuSMV never constructs this transition system explicitly. Instead, it constructs a symbolic transition relation, i.e., a Boolean function with two copies of each variable, one for the start state and one for the end state. This Boolean function can be derived almost immediately from the assignment statements: when `var` refers to a variable in the start state, `next(var)` refers to the variable in the end state. The Boolean function is thus the conjunction of all assignment statements.

An fSMV model consists of a base system, such as the one shown above, a list of features and an FD in TVL [Classen et al., 2011a] syntax.[3] A base system is specified in SMV, whereas features are specified in a dialect of SMV. The whole language is called 'fSMV'. Features in fSMV are based on *superimposition* [Francez and Forman, 1990]: a feature describes the changes to be made to the base system. A feature declaration consists of three parts [Plath

[3]Note, however, that FDs are not part of the original definition in [Plath and Ryan, 2001] and are currently not implemented by our toolset.

and Ryan, 2001]:

(1) `REQUIRE` defines variables that the feature needs. These have to be defined in the base system, or by other features. The `REQUIRE` clauses define constraints on the order in which features can be composed.

(2) `INTRODUCE` defines new variables or specifications that the feature adds to the system.

(3) `CHANGE` defines changes made to existing variables. Types of change are:

  (3.1) `IMPOSE` a new definition of an existing variable. This means that the feature replaces the `init` or `next` state definition of the variable. An `IMPOSE` clause can be guarded with an `IF` clause, meaning that it only has an effect if a certain condition holds.

  (3.2) `TREAT` existing variables differently. When the value of the variable is read inside the definition of some other variable, the read value is modified. `TREAT` clauses can also be guarded, but this is syntactic sugar [Plath and Ryan, 2001].

As an example, consider a feature *Sleep* which adds a switch to the system that causes it to discard any further request. The switch is modelled with a new non-deterministic variable `sleep` (using `INTRODUCE`). The system is changed in such a way that if the system is sleeping and finished treating requests, then it will stay idle, not accepting any new requests (using `IMPOSE`).

Listing 8.2: The *Sleep* feature of the controller system.

```
1  FEATURE sleep
2
3  REQUIRE
4    MODULE main
5    VAR state: {idle, busy};
6
7  INTRODUCE
8    MODULE main
9    VAR sleep: boolean;
10
11 CHANGE
12   MODULE main
13   IF sleep & state = idle THEN
14      IMPOSE next(state) := idle;
```

Given a base system and a feature whose `REQUIRE` constraints are satisfied by the base system, the *feature composition* operation creates a new base system. Feature composition is syntactic, and consists in replacing existing `assign` or `init` statements, and adding new variables. It is performed in three

steps: first, `TREAT` assignments are applied, then `IMPOSE` and then `INTRODUCE` assignments.

The composition of the base system and the preceding *Sleep* feature yields the following system.

```
1  MODULE main
2  VAR
3    request: boolean;
4    state:   {idle, busy};
5    sleep:   boolean;
6  ASSIGN
7    init(state) := ready;
8    next(state) :=
9      case sleep & state = idle: idle;
10          1: case state = idle & request: busy;
11                  1: {idle, busy};
12             esac;
13          esac;
```

To make it possible to structure large models and to reuse model fragments, SMV and fSMV have the `MODULE` syntax. A module encapsulates variables and assignments and can be used as a type inside other modules. The `main` module defines the behaviour of the system. Any other module must be used in the `main` module (or in a module used in the `main` module). Modules can be parameterised. A parameter is a reference to a variable to which the module would otherwise not have access. Modules can be considered syntactical sugar and can easily be eliminated by a syntactic procedure. When we give formal definitions, we thus abstract away from modules.

NuSMV also allows to use parallel composition by declaring a variable (whose type is a module) as a `process`. The executions of processes are interleaved. In fSMV, parallel composition can be used at the level of the base system. Regarding the discussion of Section 4.3, parallel composition in fSMV is thus inherently Intra-SPL composition.

8.2.2 Semantics

As described above, the semantics of a normal SMV model (or an fSMV base system) is a transition system. Furthermore, as the composition of a base system and a feature yields another base system, the semantics of a product (i.e., a base system composed with several features) is a transition system as well. This is consistent with FTS, where the behaviour of a product is also a transition system. However, if fSMV is to serve as a high-level language for FTS, we need to express its semantics in terms of FTS. We do this by giving a translation from fSMV to symbolic FTS as defined in Chapter 7. To be able to do this in a precise manner, we need to formalise the description of fSMV.

Let us start by defining an fSMV base system.

Definition 8.1. *Let V be a set of variables, D a set of (finite) domains or types, and $E(V)$ the set of all SMV expressions over V. Let $A(V_1, V_2)$ be the set of assignments where variables in V_1 can be on the left-hand side and expressions over variables in V_2 can be the left-hand side. Formally, $A(V_1, V_2) \subseteq \{\texttt{-}, \texttt{init}, \texttt{next}\} \times V_1 \times E(V_2)$, that is, a set of triples (s, d, e) where s distinguishes between* `d` *(the -),* `init(d)` *or* `next(d)` *for $d \in V_1$, and $e \in E(V_2)$ is an expression.*[4] *A base system m is a tuple $m = (v, \tau, a, p)$, where*

- *$v \subseteq V$ is a set of variables,*
- *$\tau : V \to D$ a function assigning a domain to each variable,*
- *$a \subseteq A(v, v)$ is a set of assignments, and*
- *$p \subseteq \mathcal{P}(v) \times \mathcal{P}(v)$ is a (possibly empty) set of processes. A process is a couple (v_p, w_p) where v_p denotes the set of variables read by the process and $w_p \subseteq v_p$ denotes the set of variables written by the process. Furthermore, SMV requires that the sets of written variables do not overlap.*

For a model without parallel composition, the set p is empty. The semantics of a base system is a transition system [McMillan, 1993].

As said before, for the purpose of this discussion, we abstract away from modules without loss of generality. An fSMV model is defined as follows.

Definition 8.2. *An fSMV model is a pair (b, d, G), where b is a base model as defined in Definition 8.1, d is an FD as defined in Definition 1.1 and G an (ordered) list of features. Let N be the set of features in the FD, we assume there to be a bijective function $impl : N \to G$ with codomain G, that associates features from the FD and their* implementations *in the model. When it is clear from the context, we write f instead of $impl(f)$ or $impl^{-1}(f)$. Each feature $f \in G$ is a tuple consisting of*

- *$v_f \subseteq V$, a set of new variables;*
- *$\tau_f : v_f \to D$, a type function;*
- *$p_f : p \to \mathcal{P}(v_f) \times \mathcal{P}(v_f)$, a function that tells for each process whether the new variables belong to it (read and write respectively); sets of written variables cannot overlap;*
- *$a_f \subseteq A(v_f, v \cup v_f)$, a set of* `INTRODUCE` *assignments;*
- *$m_f \subseteq E(v \cup v_f) \times A(v, v \cup v_f)$, a set of guarded* `IMPOSE` *assignments. The first element is the guard, the second is an assignment where the left-hand side is the variable that is affected, and the right-hand side the value it takes if the guard is true;*

[4] An expression alone is not an assignment, it just defines a value.

- $t_f \subseteq A(v, v \cup v_f)$, *a set of* `TREAT` *assignments. The left-hand side is the variable that is affected, and the right-hand side the value substituted by the feature.*

This definition does not formalise the `REQUIRES` constraint which has no effect on the behaviour of the products.

Since feature composition is a syntactical operation, composing features in a different order might lead to a different result, i.e., two features do not necessarily commute. Intuitively, features composed later can override changes made by earlier features. Strictly speaking, this means that a product in fSMV is a *list* of features, not a set. This is incompatible with the way products are defined in FTS and FDs. A way around this mismatch would be to assume that condition IV of [Plath and Ryan, 2001] is met: that the order of features is irrelevant, i.e., all features commute: $f_i \otimes f_j = f_j \otimes f_i$. This assumption would allow us to consider a product as a set of features. However, it would also exclude any model in which two features change the same variable. We thus opted for a different solution, which is to assume that a total ordering of the features is given as part of the model. With this assumption, a product can be also be given by a set of features. Note that this drastically reduces the number of products, from $O(\sum_{i=0}^{n} \frac{n!}{(n-i)!})$ to $O(2^n)$.

Feature composition can be formally defined as follows.

Definition 8.3. *Composition of a base system* $b = (v, \tau, a, p)$ *and a feature* $f = (v_f, \tau_f, p_f, a_f, m_f, t_f)$ *is noted* $b \otimes f$ *and produces a new base system* $b' = (v', \tau', a', p')$*, where*

- $v' = v \cup v_f$ *and* $\tau' = \tau \cup \tau_f$
- a' *is obtained by first applying* t_f *to* a*, then* m_f*, and finally adding* a_f*, formally:*

$$
\begin{aligned}
a' &= a_m \cup a_f \\
a_m &= \big\{(s,d,e') \mid (s,d,e) \in a_t \wedge \\
&\qquad \text{if } \exists (g,(s',d',e'')) \in m_f \bullet s = s' \wedge d = d' \\
&\qquad \text{then } e' = \texttt{case}\ g : e''; \ 1 : e\ \texttt{esac;} \\
&\qquad \text{else } e' = e\big\} \\
a_t &= \big\{(s,d,e') \mid (s,d,e) \in a \wedge e' = treat(e, t_f)\big\}
\end{aligned}
$$

  where $treat(e, t_f)$ *transforms* e *so that for all* $(s,d,e'') \in t_f$*, the occurrences of* $s(d)$ *are replaced by* e''*.*

- $p' = \{(v \cup v_f, w \cup w_f) \mid (v,w) \in p \wedge p_f(v,w) = (v_f, w_f)\}$

The definition of a', the set of assignments of the composed system, is somewhat cryptic. It is defined with three intermediate results. The first is a_t, i.e., after the `TREAT` assignments were applied. A `TREAT` assignment changes the right-hand side of all assignments by replacing the occurrences of a variable by an

expression. The second intermediate result is a_m, i.e., after `TREAT` and `IMPOSE` were applied. An `IMPOSE` assignment replaces the right-hand side of a single assignment (the variable it concerns) by a case statement: if the guard is true, the replacement expression is used, otherwise, the previous expression.

Note that we intentionally keep these definitions at a high level of abstraction. They are sufficiently detailed to make the following discussion precise and abstract enough to make it intuitive. In particular, we do not detail the syntax or semantics of expressions and types. There are a number of rules on what constitutes a valid model (wrt. types, variable names, etc.) which we also omit. The interested reader is referred to [McMillan, 1993, Cavada et al., 2004, Plath and Ryan, 2000] for a detailed formal definition of SMV, NuSMV and fSMV.

As said before, given a base system b and a list of features, $b \otimes f_1 \otimes \ldots \otimes f_n$ denotes a symbolic transition system. To produce a symbolic FTS, we can use the *lifting* technique of [Post and Sinz, 2008]. The idea is to introduce a new Boolean variable for each feature. Furthermore, all changes made by a feature are guarded by its feature variable. This leads us to define *lifted* feature composition as follows (the changes wrt. Definition 8.3 are shown in colour).

Definition 8.4. *Lifted composition of a base system* $b = (v, \tau, a, p)$ *and a feature* $f = (v_f, \tau_f, p_f, a_f, m_f, t_f)$ *is noted* $b \odot f$ *and produces a new base system* $b' = (v', \tau', a', p')$*, where*

- $v' = v \cup v_f \cup \{var(f)\}$ *and* $\tau' = \tau \cup \tau_f \cup \{var(f), \{0,1\}\}$*, where* $var(f)$ *denotes the feature variable associated to* f*,*
- a' *is obtained by first applying* t_f *to* a*, then* m_f*, adding* a_f*, and finally an assignment that requires the feature variable to remain constant formally:*

$$
\begin{aligned}
a' &= a_m \cup a_f \cup \{(\texttt{next}, var(f), var(f))\} \\
a_m &= \{(s,d,e') \mid (s,d,e) \in a_t \wedge \\
&\qquad \text{if } \exists (g,(s',d',e'')) \in m_f \bullet s = s' \wedge d = d' \\
&\qquad \text{then } e' = \texttt{case}\ var(f)\ \&\ g : e''; \ 1 : e\ \texttt{esac;} \\
&\qquad \text{else } e' = e\} \\
a_t &= \{(s,d,e') \mid (s,d,e) \in a \wedge e' = treat(e, t_f)\}
\end{aligned}
$$

  where $treat(e, t_f)$ *transforms* e *so that for all* $(s,d,e') \in t_f$*, the occurrences of* $s(d)$ *are replaced by* `case` $var(f) : e'; \ 1 : e$ `esac;`*.*
- $p' = \{(v \cup v_f, w \cup w_f) \mid (v,w) \in p \wedge p_f(v,w) = (v_f, w_f)\}$

Guarding a change made by a feature with the corresponding feature variable means that the behaviour *with* (resp. *without*) the feature can be obtained by setting the feature variable to 1 (resp. 0). For example, the lifted composition of the controller base system and the *Sleep* feature yields the following.

```
1 MODULE main
```

```
2  VAR
3    sleepFeature: boolean;      -- added feature variable
4    request: boolean;
5    state:   {idle, busy};
6    sleep:   boolean;
7  ASSIGN
8    next(sleepFeature) := boolean;
9    init(state) := idle;
10   next(state) :=
11     case sleepFeature & sleep & state = idle: idle;
12          1: case state = idle & request: busy;
13                  1: {idle, busy};
14             esac;
15     esac;
```

The lifted composition of a base system and a list of features, $b \odot f_1 \odot \ldots \odot f_n$ denotes a symbolic FTS. It corresponds to the symbolic encoding given in Section 7.3: the features are part of the states because each has a Boolean feature variable, the feature variables are initialised non-deterministically, and they do not change their value as part of a transition. Since this lifted composition is also a valid SMV model, and given that symbolic FTS model checking can be reduced to model checking of symbolic transition systems, we can feed it as-is into NuSMV and reuse the result of the satisfaction set computation.

Recall that an fSMV model is a base system and a *list* of features. The lifted composition of the base system and all features in the given order is thus always well-defined. To simplify the notation, we therefore write $fts(m) \triangleq b \odot f_1 \odot \ldots \odot f_n$, for an fSMV model $m = (b, \{f_1, \ldots, f_n\})$, to denote the corresponding symbolic FTS. A symbolic FTS can be projected to a product by fixing the values of the feature variables according to the product:

Definition 8.5. *Given an fSMV model m with features G, the projection of $fts(m) = (v, \tau, a, p)$ to a product $p \in [\![d]\!]_{FD}$ is the model $fts(m)_{|p} \triangleq (v, \tau, a', p)$ where $a' = a \cup \{(\mathit{init}, var(f), f \in p) \mid f \in G\}$.*

We have shown that lifted feature composition indeed yields a symbolic FTS. What is left to be shown is that it preserves the semantics of the normal feature composition as it was defined by [Plath and Ryan, 2001]. This is indeed the case. The following theorem establishes that lifted composition (Definition 8.4) is *trace equivalent* [Baier and Katoen, 2008] to normal feature composition (Definition 8.3) modulo projection, if restricted to variables shared by both. Trace equivalence was already discussed in Section 4.4.2. It means that both have the same executions, where two executions are equivalent if the variables shared by both are identical. The last bit is important since the projection of a lifted composition can have more variables than the normal composition: those added by non-selected features. When comparing both systems, these variables are of course irrelevant.

Theorem 8.6. *For any fSMV* $m = (b, d, \{f_1, \ldots, f_n\})$ *and product* $p = \{f_i, \ldots, f_j\}$*, where the indexing from* i *to* j *corresponds to the feature order given in* m*,*

$$[\![b \otimes f_i \otimes \ldots \otimes f_j]\!] \equiv [\![(b \odot f_1 \odot \ldots \odot f_n)_{|p}]\!]$$

where $[\![smv]\!]$ *denotes the set of executions of an SMV model, and* $\equiv$ *denotes trace equivalence.*

Proof sketch. First, observe that lifted feature composition does not remove any of the code in the base system; rather, it wraps changed code inside `case` statements. In a projection, the values of the features are all fixed. This means that references to features can be replaced everywhere by their value. Any changes by non-selected features will be of the kind `case` 0 & ... : exp_1; 1 : exp_2 `esac;`, where exp_2 is the expression that was there before the feature was added. Such `case` statements can be simplified to exp_2`;`. For selected features, the simplification is similar, except that the change made by the feature is preserved. Intuitively, these semantics-preserving transformations correspond to unwrapping of `case` statements introduced by lifted composition. The result is syntactically equivalent to $b \otimes f_i \otimes \ldots \otimes f_j$, with the exception of added variables (and their assignments), which are irrelevant anyway. □

We define the semantics of fSMV in terms of SMV, i.e., as a symbolic FTS. In this case, the parallel composition is defined as it is for SMV. An alternative would be to define the semantics in terms of explicit FTS (Definition 4.1). This would make it possible to define the semantics of fSMV compositionally, where each process translates to an FTS. We have done this in [Classen, 2010a].

8.2.3 Expressiveness

We evaluate the expressiveness of fSMV to assess the extent to which it covers the FTS language. This is done wrt. explicit FTS, as defined in Chapter 6.

The results of the previous section already established that any fSMV can be translated to an FTS, i.e., that the fSMV language is a subset of the FTS language. The following theorem establishes that the converse also holds, i.e., that both languages are expressively equivalent.

Theorem 8.7. *Any FTS can be translated into fSMV.*

Proof. Given an FTS $(S, Act, trans, I, AP, L, d, \gamma)$, construct an fSMV model (b, d, F) with $b = (v, \tau, a, \emptyset)$, where

(1) One variable of the fSMV, *state*, is used to encode all the states of the FTS: $\tau(state) = S$. For every feature $f \in N$, the fSMV will have a variable f with $\tau(f) = \{0, 1\}$. Hence, $v = \{state\} \cup N$;

(2) In the base system, the feature variables are always 0. The assignments related to the feature variables are thus $a_F = \{(\text{-}, f, 0) \mid f \in N\}$. The initial value of the *state* variable is the non-deterministic choice between the

initial state of the FTS, and its `next` value is derived from the transition relation of the FTS. The assignments for the *state* variable are

$$a_s = \{(\texttt{init}, state, I), (\texttt{next}, state, \texttt{case}\ case_1 \ldots case_k\ \texttt{esac;})\}$$

where the $case_i$ are given by $\{state = s\ \texttt{\&}\ \gamma(s \xrightarrow{\alpha} s')\texttt{:}\ s'\texttt{;} \mid s \xrightarrow{\alpha} s' \in trans\}$. The set of assignments is then $a = a_F \cup a_s$;

(3) Each feature imposes that its associated variable (which is part of the base system) takes the value 1, i.e.,

$$F = \{(\emptyset, \emptyset, \emptyset, \emptyset, \{(1, (\text{-}, f, 1))\}, \emptyset) \mid f \in N\}.$$

In consequence, feature composition of a base system with a set of features yields a transition system of which all transitions whose $\gamma(s \xrightarrow{\alpha} s')$ evaluates to 0 for the feature variables have been removed. This corresponds exactly to projection as defined in Definition 4.2. □

Note that Theorem 8.7 does not have to take parallel composition into account directly. Any parallel composition of two FTS is an FTS itself, and can hence be translated into an fSMV model.

8.3 fNuSMV

We now give an overview of our model checking toolset. First, we present it from the user perspective. We then discuss more technical details of the implementation.

8.3.1 User interface and illustration

Like NuSMV, our toolset is command-line based. This is rather natural, especially since the composition scripts lend themselves well to pipe-based chaining of commands.

The input to the tool is a model expressed in fSMV. Concretely, this means at least one file containing the base system and one file for each feature. The feature order is not explicitly part of the syntax, it is specified when using the composition script. The composition script, `compose.php`, is indeed the first tool to be used in a normal use case. Let us illustrate this with the running example of the chapter, the controller. Assume that there are two files `base.smv` corresponding to Listing 8.1 and `sleep.feat` corresponding to Listing 8.2. The feature composition (following Definition 8.3) of the base system and the *Sleep* feature is given by:

```
$ php compose.php sleep.feat < base.smv > ts.smv
```

The composition script is written in PHP, which is why the command starts with `php`. The result, `ts.smv`, is the behaviour of the product consisting of the base system and the sleep feature. It can be analysed using NuSMV. To produce the lifted feature composition (Definition 8.4) instead of the normal feature composition, the command-line parameter `-l` has to be set, i.e.:

```
$ php compose.php -l sleep.feat < base.smv > fts.smv
```

The script thus has one parameter, the file name of the feature to be composed, and reads the base system from `stdin` (hence the input redirection `<` in the previous two examples). This means that when there are several features to compose, the calls can be chained with pipes, e.g.:

```
$   php compose.php -l sleep.feat < base.smv
  | php compose.php -l other.feat > fts.smv
```

When chaining multiple calls, the `-l` parameter has to be set for all of them or for none of them.

The heart of our toolset is fNuSMV, which is a modified version of NuSMV. It is distributed as a patch (created using the standard Unix `diff` tool) which has to be applied to the NuSMV codebase before NuSMV is compiled. When this is done, a new command-line parameter `-fbdd` is available in NuSMV.

Let us illustrate the use of fNuSMV with the controller example. A property for such a system would be that *"every request eventually results in the controller being busy"*. In CTL, this becomes $A\Box(request \Rightarrow A\Diamond\ state = busy)$, or in NuSMV syntax `SPEC AG (request -> AF state=busy)`. In NuSMV, the temporal operators are written with letters: $\bigcirc$ becomes `X`, $\Box$ becomes `G` and $\Diamond$ becomes `F` (the `U` for *until* remains). Checking this property using fNuSMV would look as follows.

```
$ ./NuSMV -fbdd fts.smv
*** This is NuSMV 2.5.0 [...]
-- Computing fbdd init.. done.
-- specification AG (request -> AF state = busy)  is false
-- (fbdd) specification is false for products satisfying:
   f.fSleep
-- (fbdd) specification is true for products satisfying:
   !f.fSleep
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  f.fSleep = TRUE
  request = FALSE
  state = idle
  sleep = FALSE
-- Loop starts here
-> State: 1.2 <-
  request = TRUE
  sleep = TRUE
```

```
-> State: 1.3 <-
```

The global result, printed at line 4, is that the property is violated. At line 6, fNuSMV prints a feature expression characterising the products that violate the property, in this case all those containing the *Sleep* feature. At line 8, the negation of this feature expression is shown (this helps in the case of larger expressions). This is followed by a counterexample at lines 10 to 21. As we said before, the generation of counterexamples in the case of CTL was not studied, and so fNuSMV will not print counterexamples for every product, but just one for the first violating product. The counterexample shows that indeed, the *Sleep* feature can cause the system to indefinitely stay idle. Furthermore, line 8 says that without the *Sleep* feature, this is impossible.

As a comparison, if we use NuSMV without the `-fbdd` parameter, the result will be the following.

```
$ ./NuSMV fts.smv
*** This is NuSMV 2.5.0 [...]
-- specification AG (request -> AF state = busy)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  f.fSleep = TRUE
  request = FALSE
  state = idle
  sleep = FALSE
-- Loop starts here
-> State: 1.2 <-
  request = TRUE
  sleep = TRUE
-> State: 1.3 <-
```

This corresponds to the MC_{CTL} decision problem, as it just identifies a single violating product, given by the initial values of the feature variables in the counterexample (line 8). The question whether other products violate or satisfy the property is left open by this check.

In addition to `compose.php` and fNuSMV, our toolset comprises a quantification script, `quantify.php`. This script allows to create parameterised models. In the case of the controller, for example, one could imagine that the controller is capable of treating several requests in parallel. The parameter of the model would be the number of requests. For the value two, the model would be the following:

```
1 MODULE main
2 VAR
3   request1: boolean;
4   request2: boolean;
5   state:   {idle, busy};
6 ASSIGN
```

```
7     init(state) := idle;
8     next(state) :=
9       case state = idle & request1 & request2: busy;
10           1: {idle, busy};
11      esac;
```

In the SMV (and thus fSMV) syntax, it is not possible to specify the model for a number of requests n. This capability is added with the quantification script which acts as a preprocessor. Inside a preprocessed model file, the syntax `[forall p=i..j]body[/forall]` can be used:

`i` **and** `j` are integers that define a range. Instead of an integer, the letter `n` can be used, which will be substituted by a user-defined value when running the script. The bounds can also be given with expressions, e.g. `p=n-1..n`.

`body` is the piece of text to be repeated.

`p` is the name of the index. It can be referenced inside `body` by writing `%p%` (the index enclosed by percent signs). Simple expressions such as `%p-1%` are also supported.

In the case of the controller example, this would be used as follows.

```
1   MODULE main
2   VAR
3     [forall i=1..n]
4       request%i%: boolean
5     [/forall]
6     state:   {idle, busy};
7   ASSIGN
8     init(state) := idle;
9     next(state) :=
10      case state = idle [forall i=1..n] & request%i%
11                        [/forall]: busy;
12           1: {idle, busy};
13      esac;
```

To produce the example with two requests, the quantification script is used to process the above file.

```
$ php quantify.php 2 base.smv > base-2requests.smv
```

Quantifiers have to be instantiated before a file can be used in the composition script. This can make the use of the tools rather cumbersome. In practice, it is helpful to encapsulate all quantification and composition commands in a shell script. Moreover, most models do not require quantifiers.

8.3.2 Implementing composition

The composition script implements feature composition and lifted feature composition as specified in Definitions 8.3 and 8.4. It is slightly more involved than the definitions suggest, mainly because it has to deal with modules. In both cases, the composition script has to respect the scope of the module to which a change is applied. Moreover, for lifted feature composition, the composition script has to add feature variables so that they can be referenced in all modules uniformly. NuSMV does not allow access to variables in parent modules. This is only possible by parameterising modules. To make the feature variables available in all modules, the composition script therefore creates a new module with all features, instantiates it in the `main` module, and adds it as a parameter to all other modules.

The module containing the features is called `features`, which means that 'features' cannot be used as an identifier in the fSMV files. To each feature corresponds one variable in this module, the variable name being the feature name (with the first letter in uppercase) prefixed by the letter `f`. The `feature` module is declared in the `main` module as a variable named `f`. Therefore, `f` is also a reserved keyword which should not be used inside fSMV models. The parameter added to all other modules is also called `f`, so that the features can be referenced the same way inside the whole model.

For the controller example in Listings 8.1 and 8.2, the lifted feature composition would result in the following SMV model.

```
1  MODULE features
2  VAR
3    fSleep: boolean;
4  ASSIGN
5    init(fSleep) := {0,1};
6    next(fSleep) := fSleep;
7
8  MODULE main
9  VAR
10   f:       features;
11   request: boolean;
12   state:   {idle, busy};
13   sleep:   boolean;
14 ASSIGN
15   next(state) :=
16     case f.fSleep & sleep & state = idle: idle;
17          1: case state = idle & request: busy;
18                  1: {idle, busy};
19             esac;
20     esac;
```

Note that the feature variables are initialised non-deterministically and that

their value is defined to be constant.

Naming conventions allow us to easily distinguish feature variables from other Boolean variables. All feature variables have the prefix `f.f`: the first `f` identifies the variable of the main module that holds the feature module and the second `f` is the one prefixed to every feature variable. We need to be able to distinguish feature variables from the other variables when calculating the products for which a certain property holds. An alternative to the naming convention would have been to extend the SMV language. We chose a naming convention as this necessitates far less changes to the NuSMV codebase.

Furthermore, the naming convention means that fCTL properties can be written and attached to any module in the same way. E.g., in the case of the controller, the property that every request will eventually result in a busy controller might not be relevant for controllers with the *Sleep* feature. In fCTL, this can be written $[!sleep]A\Box\ request \Rightarrow A\Diamond\ state = busy$. In SMV, this becomes `SPEC !f.fSleep -> AG (request -> AF state=busy)`. As expected, the property is satisfied by all products:

```
$ ./NuSMV -fbdd fts.smv
*** This is NuSMV 2.5.0 [...]
-- Computing fbdd init.. done.
-- specification (!f.fSleep -> AG (request -> AF state = busy))
is true
```

8.3.3 Implementing FTS model checking

The output of the composition tool is a normal SMV model and can be model checked directly by NuSMV. However, as shown before, standard NuSMV model checking does not fully exploit the feature encoding. Since NuSMV executes the standard CTL model checking algorithm, it will report $false$ if it finds a counterexample. More precisely, it will return $false$ if just one of the products violates the property.

Basically, given a property ϕ, the algorithm will compute a Boolean function $\chi_{Sat(\phi)}(\overline{s}, \overline{p})$, where $\overline{s}$ (resp. $\overline{p}$) is the Boolean encoding of some state (resp. some product). $\chi_{Sat(\phi)}$ is true for all states and products that satisfy the property. The normal model checking algorithm will just check whether there exists some initial state for which $\chi_{Sat(\phi)}(\overline{s}, \overline{p})$ is $false$. Unable to distinguish between feature variables (belonging to $\overline{p}$) and normal variables (belonging to $\overline{s}$), the test will existentially quantify over the feature variables which corresponds to considering a single product only.

As discussed in Section 7.3, there is sufficient information to determine exactly which products violate and which satisfy the property. The idea is to only quantify $\chi_{Sat(\phi)}$ existentially over the state variables (i.e., those that do not represent features). The result is a Boolean function over the feature variables that represents exactly the products for which the property holds. Implementing this calculation is the only significant change we made to the

NuSMV code (the other being the addition of the command-line parameter). Still, it accounts for just about 44 lines of additional code. One fragment of code is executed once for all properties in the model, it consists in creating the BDDs of the feature variables which are used in the quantification. The second fragment is executed for each property, after the model checking algorithm is finished. It calculates the quantifications and prints the information about satisfying and violating products on `stdout`.

Currently, FDs are not implemented as part of the toolset. However, this would be quite straightforward. Following Definition 7.12, it would be sufficient to test whether the conjunction of the returned function and $\mathbb{B}(d)$, the Boolean function equivalent of the FD d, is satisfiable. An alternative would be to limit the check to the set of valid products of the FD from the outset. This can be done by adding constraint `IVAR` $\mathbb{B}(d)$ to the model before it is analysed. NuSMV considers it as an invariant, which will effectively prevent it from considering invalid products.

8.4 Experiments

In [Plath and Ryan, 2001], the authors propose the fSMV language along with a verification technique for CTL that is based on the naïve algorithm, i.e., an exhaustive enumeration of the set of products (although they limit themselves to couples of features). Such a product-enumerative approach is exactly what we intend to avoid. While both approaches produce equivalent results, we argue that model checking a single model with variability (i.e., the model of the whole SPL) is in general more efficient. Experiments with the Haskell FTS library also suggest that [Classen et al., 2010b]. Here, we test this hypothesis in the symbolic context through benchmarks that compare the runtime of the naïve algorithm and the FTS algorithm.

8.4.1 Elevator system

For these experiments we used the elevator system from [Plath and Ryan, 2001]. We extended the SMV models provided with the original paper in two ways. First, we made the number of floors (fixed at five in [Plath and Ryan, 2001]) variable. Secondly, we added four more features to the system, giving a total of nine features. All features are independent, which means that there are 2^9 products. The elevator system is comprised of a number of platform buttons and a number of cabin buttons. There is a single button on each platform, which calls the elevator. The button press is modelled non-deterministically, and a pressed button remains pressed until the elevator has served the floor and its doors are opened. The elevator will always serve all requests in its current direction before it stops and changes direction. When serving a floor, the lift doors open and close again. There are nine features that modify the behaviour of the lift. Those marked with an asterisk were added by us.

Anti-prank.* Normally, a lift button will remain pushed until the corresponding floor is served. With this feature, the lift buttons have to be held pushed by a person.

Empty. If the lift is empty, then all requests made in the cabin will be cancelled.

Executive floor. One floor of the building has priority over the other floors and will be served first, both for cabin and platform requests.

Open when idle.* When idle, the lift opens its doors.

Overload The lift will refuse to close its doors when it is overloaded.

Park. When idle, the lift returns to the first floor.

Quick close.* The lift door cannot be kept open by holding the platform button pushed.

Shuttle.* The lift will only change direction at the first and last floor.

Two-thirds full. When the lift is two-thirds full, it will serve cabin calls before platform calls.

To test the correctness of our approach we reduced the example to the five features from [Plath and Ryan, 2001] and managed to reproduce the feature interactions reported in the original paper. Subsequently, we made some minor modifications to the model to accommodate the additional features. The models are distributed with the toolset on the FTS website [Classen, 2010b].

The properties used for our benchmarks are those of the base system shown in Table 8.1 (mostly combined safety and liveness properties). The property numbers reported in the statistics refer to the numbers in Table 8.1, and can also be used to identify the properties in the NuSMV code.

8.4.2 Experimental setup

The goal of our experiments is to evaluate the performance of our algorithms. To this end, we compare the FTS algorithm with the naïve algorithm.

The experiments consist in using both algorithms to check the fifteen properties of the base system given in Table 8.1 against an elevator model with the number of floors ranging from 4 to 8. Each property was benchmarked in a separate run of the model checker. The benchmarks were run on an Ubuntu machine with an Intel Core2 Duo at 2.80 GHz with 4 Gb of RAM.

The reported benchmarks compare (for each property)

- the total runtime of 2^9 model checks that enumerate all products explicitly (column 'Enumerative' in Tables 8.2, 8.3, 8.4, 8.5 and 8.6).
- the runtime of a single NuSMV model check following our method (column 'Single' in Tables 8.2, 8.3, 8.4, 8.5 and 8.6);

Table 8.1: Benchmarked properties

ID	Property
01	$A\Box(landingBut2.pressed \Rightarrow A\Diamond(lift.floor = 2 \wedge lift.door = open))$
01'	$\neg A\Box(landingBut2.pressed$ $\Rightarrow A\Diamond(lift.floor = 2 \wedge lift.door = open \wedge lift.direction = down))$
02	$A\Box(liftBut3.pressed \Rightarrow A\Diamond(floor = 3 \wedge door = open))$
03a	$A\Box(floor = 2 \wedge liftBut6.pressed \wedge direction = up$ $\Rightarrow A[direction = up U floor = 6])$
03b	$A\Box(floor = 6 \wedge liftBut1.pressed \wedge direction = down$ $\Rightarrow A[direction = down U floor = 1])$
04	$\neg A\Box(door = closed \Rightarrow A\Diamond door = open)$
05a	$EF(floor = 1 \wedge idle \wedge door = closed \wedge AX(door = closed))$
05b	$A\Box(floor = 1 \wedge idle \wedge door = closed \wedge A \bigcirc (door = closed)$ $\Rightarrow EG(floor = 1 \wedge door = closed))$
05-part	$EF(A \bigcirc (door = closed))$
05c	$EF(floor = 3 \wedge idle \wedge door = closed \wedge A \bigcirc (door = closed))$
05d	$A\Box(floor = 3 \wedge idle \wedge door = closed \wedge A \bigcirc (door = closed)$ $\Rightarrow EG(floor = 3 \wedge door = closed))$
05e	$EF(EG(door = closed))$
05'	$\neg A\Box(floor = 4 \wedge idle \Rightarrow E[idle U floor = 1])$
06	$\neg A\Box((floor = 3 \wedge \neg liftBut3.pressed \wedge direction = up)$ $\Rightarrow door = closed)$
07	$\neg A\Box((floor = 3 \wedge \neg liftBut3.pressed \wedge direction = down)$ $\Rightarrow door = closed)$

The size of the NuSMV model of the product with all features ranges from 2^{17} states for four floors, to 2^{27} states for eight floors. These are the upper bounds for the size of the models analysed in the *enumerative* benchmarks. As explained earlier, our algorithm only needs one check, but requires an additional variable for each feature. Its models are thus much larger, from 2^{26} states to 2^{36}.

An important factor in BDD based model checking is the variable ordering. In order to avoid computing static variable orderings and still be efficient, NuSMV has the parameter `-dynamic`, which causes the BDD package to reorder

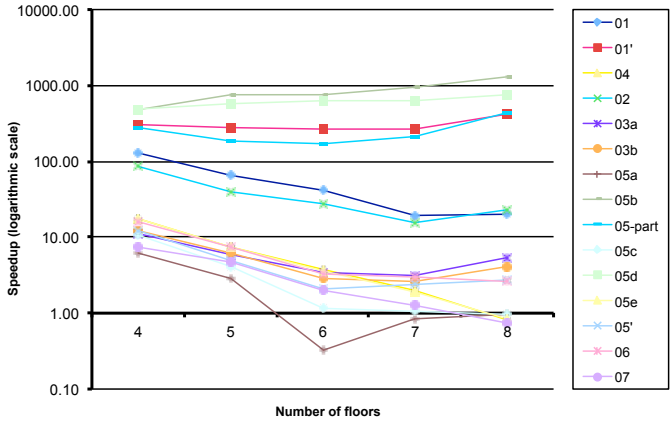


Figure 8.1: Evolution of speedup with the number of floors (logarithmic scale).

the variables during verification in case the BDD size grows beyond a certain threshold. When using this method for the single model check, it works well on small to medium models (up to six floors). However, its limitations become more and more apparent as the size of the models grows. In the case of the single model check for eight floors (i.e., a model of size 2^{36}), NuSMV would spend more time reordering variables than actually verifying the property.

In consequence, we computed variable orderings for each number of floors, and used these in all subsequent benchmarks. The model checks of the *single* approach were run with parameters `-df -i orderfile`. Those of the *enumerative* approach were run with `-df -dynamic`. It is important to note that the variable orderings computed for the *single* approach cannot be reused for the *enumerative* case. This is due to the fact that the *enumerative* approach produces 2^9 models with different sets of variables, which would require 2^9 variable orderings. However, due to the absence of the nine feature variables, the individual models of the *enumerative* cases are much smaller than the single model in the *single* case. Therefore, the dynamic variable ordering, while being the only option, should still be rather efficient for the *enumerative* case.

8.4.3 Results

The results of the benchmarks are given in Tables 8.2, 8.3, 8.4, 8.5 and 8.6. They show that our approach achieves order-of-magnitude speedups over the enumerative approach. These observations are reported for each property in Figure 8.1, where we show how speedup evolves when the number of floors grows. Three clusters appear: four high outliers, with speedups greater than 250 and up to 1000; five low outliers with speedups below two or three and sometimes negative; and six stable properties with speedups around ten. A trend that we observed is that with an increasing number of floors, the outliers tend to become more extreme (the high speedups grow, the low speedups de-

scend). This is most likely due to the importance of the static variable ordering for larger models, although we cannot exclude other factors.

We conducted a second experiment in which we used the dynamic variable ordering for both algorithms. In this case, when the number of floors was larger than six, the smaller models of the naïve algorithm caused it to be more efficient than the FTS algorithm on more than half of the properties. This means that the improvement in speedup seen above can be attributed to a large extent to the use of an optimised static variable ordering. The crucial advantage of our algorithm is therefore that it just needs one variable ordering. Note that both algorithms could be combined. First, our algorithm would be used to find a good variable ordering. Since this variable ordering comprises all the variables that are used in the naïve checks, a variable ordering for each naïve check can be obtained from this ordering by removing irrelevant variables. The naïve algorithm could then be used with static orderings (obtained with the FTS algorithm). This procedure has not been tested or subjected to benchmarks.

8.4.4 Threats to validity

In order to limit bias, we went to great lengths to ensure that the *enumerative* benchmarks were as efficient as possible. For instance, the computation of the 2^9 feature compositions (to create the files that were model checked) for each property was not included in the runtime. Furthermore, the large volume of log files from these runs was cleaned after each run since it would slow down the tool after several runs (because of huge inode lists in the parent folder).

NuSMV has an extensive set of optimisations and parameters of which we only used the most basic ones, `-dynamic` and `-df` (preventing the computation of reachable states, which was found to be slowing down both algorithms). The naïve algorithm might benefit more from some of these optimisations, since even a small improvement can accumulate over the $O(2^n)$ runs of the the tool. On the other hand, since the set of variables is different for most products, some optimisations cannot be used in the naïve algorithm. For example, NuSMV allows hard caching of BDDs, so that they can be reused between checks. This caching cannot be used between products, because most products have different variables and thus different BDDs. Using the basic algorithm allowed us to make sure that the only difference in runtime was due to the use of the FTS algorithm and a static variable ordering.

8.5 Conclusion

We presented fNuSMV, a toolset for symbolic FTS model checking implementing the algorithms of Chapter 7. Its input language, fSMV, was taken from [Plath and Ryan, 2001]. It is based on the compositional FOSD paradigm (see Section 1.3): features are specified individually and composed to create the model of a product. For fNuSMV we proposed a new form of composition. It

creates a symbolic FTS, to which our symbolic model checking algorithm can be applied. We also showed that fSMV and FTS are expressively equivalent modelling languages.

The fNuSMV toolset allowed us to conduct experiments showing that the FTS algorithms are orders-of-magnitude faster than the naïve algorithm. Most of this improvement can be attributed to the fact that the FTS algorithm needs just one variable ordering, which can be optimised.

Table 8.2: Benchmark results for the elevator system with four floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	$false$	17.84s	0.14s	127.43
01'	$true$	15.37s	0.05s	307.40
04	$false$	18.19s	1.06s	17.16
02	$false$	19.23s	0.22s	87.41
03a	$false$	20.48s	1.84s	11.13
03b	$false$	21.23s	1.76s	12.06
05a	$false$	20.09s	3.23s	6.22
05b	$true$	14.36s	0.03s	478.67
05-part	$true$	16.47s	0.06s	274.50
05c	$false$	19.94s	1.86s	10.72
05d	$true$	14.68s	0.03s	489.33
05e	$false$	18.3s	1.06s	17.26
05'	$false$	19.89s	1.62s	12.28
06	$true$	18.89s	1.2s	15.74
07	$true$	19.27s	2.57s	7.50

Table 8.3: Benchmark results for the elevator system with five floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	*false*	29.38s	0.44s	66.77
01'	*true*	24.76s	0.09s	275.11
04	*false*	34.02s	4.62s	7.36
02	*false*	33.16s	0.82s	40.44
03a	*false*	37.98s	6.3s	6.03
03b	*false*	39.43s	6.32s	6.24
05a	*false*	39.77s	13.99s	2.84
05b	*true*	22.7s	0.03s	756.67
05-part	*true*	29.25s	0.16s	182.81
05c	*false*	35.52s	8.66s	4.10
05d	*true*	23.44s	0.04s	586.00
05e	*false*	34.09s	4.63s	7.36
05'	*false*	40.21s	8.14s	4.94
06	*true*	34.55s	4.56s	7.58
07	*true*	35.9s	7.57s	4.74

Table 8.4: Benchmark results for the elevator system with six floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	*false*	44s	1.05s	41.90
01'	*true*	34.02s	0.13s	261.69
04	*false*	67.76s	18.44s	3.67
02	*false*	52.36s	1.87s	28.00
03a	*false*	76.67s	22.42s	3.42
03b	*false*	77.98s	27.21s	2.87
05a	*false*	105.07s	322.53s	0.33
05b	*true*	30.67s	0.04s	766.75
05-part	*true*	54.63s	0.32s	170.72
05c	*false*	88.63s	78.36s	1.13
05d	*true*	30.93s	0.05s	618.60
05e	*false*	67.45s	18.39s	3.67
05'	*false*	131.78s	63.61s	2.07
06	*true*	68.36s	20.42s	3.35
07	*true*	73.06s	36.89s	1.98

Table 8.5: Benchmark results for the elevator system with seven floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	*false*	66.89s	3.45s	19.39
01'	*true*	44.34s	0.17s	260.82
04	*false*	214.75s	109.67s	1.96
02	*false*	86.98s	5.58s	15.59
03a	*false*	160.43s	51.35s	3.12
03b	*false*	169.91s	66.45s	2.56
05a	*false*	487.98s	571.69s	0.85
05b	*true*	38.39s	0.04s	959.75
05-part	*true*	114.38s	0.55s	207.96
05c	*false*	269.19s	257.98s	1.04
05d	*true*	38.62s	0.06s	643.67
05e	*false*	214.13s	112.79s	1.90
05'	*false*	568.56s	241.53s	2.35
06	*true*	142.42s	48.37s	2.94
07	*true*	160.3s	128.84s	1.24

Table 8.6: Benchmark results for the elevator system with eight floors.

Property	**Value**	**Enumerative**	**Single**	**Speedup**
01	*false*	99.14s	4.96s	19.99
01'	*true*	62.71s	0.15s	418.07
04	*false*	337.47s	414.32s	0.81
02	*false*	139.58s	6.06s	23.03
03a	*false*	312.05s	57.65s	5.41
03b	*false*	332.49s	81.35s	4.09
05a	*false*	2180.58s	2232.39s	0.98
05b	*true*	51.26s	0.04s	1281.50
05-part	*true*	211.63s	0.48s	440.90
05c	*false*	851.58s	899.2s	0.95
05d	*true*	52.27s	0.07s	746.71
05e	*false*	337.81s	407.84s	0.83
05'	*false*	2441.67s	887.8s	2.75
06	*true*	263.68s	102.39s	2.58
07	*true*	325.31s	439.25s	0.74

Chapter 9

SNIP and fPromela

> *Von Neumann gave me an interesting idea: that you don't have to be responsible for the world that you're in. So I have developed a very powerful sense of social irresponsibility as a result of von Neumann's advice. It's made me a very happy man ever since.*
>
> Richard Feynman, *Surely You're Joking, Mr. Feynman!*, 1985

In this chapter, we present SNIP, a model checker that implements the semi-symbolic algorithms of Chapter 6. SNIP was developed after fNuSMV, and our motivations for its development were (i) to test the semi-symbolic algorithms on larger models and (ii) to have a procedural modelling language based on annotation rather than composition. The result is fPromela, a modified version of Promela, the well-known language of the SPIN model checker [Holzmann, 2004]. SNIP is the first model checker dedicated to SPLs.

Our presentation begins in Section 9.1, with an introduction and a motivation. In Section 9.2, we present fPromela, SNIP's modelling language. In Section 9.3, we give examples of how SNIP can be used, and we describe its architecture and other implementation details. In Section 9.4, we report on the experiments conducted with SNIP. We conclude in Section 9.5.

9.1 Introduction

After working with fNuSMV for several months, conducting benchmarks and creating models, we became more attentive to the requirements of the engineer using the tool. During this time, we identified several drawbacks of the fNuSMV toolset. First, the fact that the toolset consists of several tools, all of which have to be used to verify a property, makes it somewhat cumbersome to use. This is especially problematic when elaborating a model. During this phase, the work of the engineer generally consists in refining the model (making corrections, additions, or other changes) and then verifying it to inspect the impact of the

change. Short feedback cycles and flexible command-line instructions are thus paramount. Secondly, the fSMV modelling language has two particularities that can make it unintuitive in certain cases.

First, SMV is declarative and very close to the symbolic transition relation required for the symbolic model checking algorithm. The behaviour has to be specified directly in terms of the changes to the variable values, one by one. This is very different from the way problems in the real world are described, which is generally done in a procedural style (e.g., program code, flow diagrams or automata). To address this point, we decided to base our language on a procedural modelling language, thereby shrinking the cognitive gap between the model and the problem being modelled. A perfect candidate for this was Promela, which is the modelling language of the popular SPIN model checker [Holzmann, 2004]. Its syntax is close to that of programming languages like C, making it easy to learn. Promela also offers many facilities for modelling distributed systems (e.g., processes and communication channels).

The second particularity that can make fSMV unintuitive in certain cases is that it requires modular features. While modularity is an ideal goal, not all features can easily be modularised. In [Boucher et al., 2010b], we worked on an industrial SPL where features would just change single instructions in the middle of a function. While this and similar small-granularity features can be modularised,[1] this introduces overhead and is contrary to the goal of feature modularity, the ability to develop features in isolation. If features are sufficiently small and specialised, they will not make sense in isolation. Furthermore, in many cases, it is clear from the outset that two features need 'glue code' to work together, or that features manipulating the same variable have a priority order. Such things are hard to express in modular languages such as fSMV. Generally, the only way to achieve modularity in this case is by duplicating features, or encapsulating 'glue code' inside artificial features—all things that are contrary to the goal of modularity. To address this second point, we thus decided to base the modelling language on annotation, rather than modularity and composition. A further motivation for this choice is the use of similar techniques in practice (e.g., `#ifdefs` [Kästner et al., 2008, Liebig et al., 2010] or code tags [Boucher et al., 2010b]).

The result is fPromela and the model checker SNIP. In contrast to fNuSMV, which was implemented by modifying the existing NuSMV model checker, SNIP was implemented from scratch. While the fully symbolic FTS algorithm can largely be reduced to the one for transition systems, this is not the case for the semi-symbolic algorithm. Implementing it by changing an existing model checker would require major changes in many parts of the code, requiring a good knowledge of the code, or well-placed extension points which would have to anticipate all the places at which changes have to be made. Furthermore, modifying the SPIN source code is far from trivial, even for simpler modifica-

[1] In aspect-oriented programming, for instance, a 'hook' (a call to an empty method) can be placed where the feature's code is supposed to be injected, which an then be addressed in a point-cut.

tions. These considerations led us to re-implement SNIP from scratch. The development took about four man-months, which is considerably more than the time spent on the development of the Haskell FTS library or fNuSMV.

To the best of our knowledge, SNIP is the first model checker specifically dedicated to SPLs. It supports both model checking problems, Mc and ExtMc for fLTL, assertion and deadlock checking. In contrast to fSMV, SNIP is an integrated tool with a single command-line interface, requiring just one call to launch a verification. Thanks to the development of the TVL modelling language [Classen et al., 2011a] and its associated libraries, SNIP also has built-in support for FDs.

9.2 The fPromela modelling language

SNIP's modelling language, fPromela, is based on the procedural modelling language Promela. In this section, we first give an overview of the language. We then discuss its semantics in terms of FTS and study its expressiveness.

9.2.1 Syntax

The syntax of fPromela is the same as the one of Promela [Holzmann, 2004]. Many constructs are similar to C. This should result in a gentle learning curve, even for people that are not familiar with Promela. For example, consider the following system consisting of a sender and a receiver.

Listing 9.1: A Promela model with a sender and a receiver.

```
1  chan buffer = [3] of { int };
2
3  active proctype sender() {
4    int p;
5    do :: true;
6          if :: p = 0;
7             :: p = 1;
8          fi;
9          buffer!p;
10       od;
11 }
12
13 active proctype receiver() {
14   do :: true;
15         buffer?_;
16       od;
17 }
```

The key element in Promela and fPromela are processes, specified with the `proctype` keyword. A process has to be started by another process or declared

`active`, which means that it is active in the initial state of the system. If several processes are active, their executions are interleaved. The sender and the receiver are both active processes. Processes can communicate through shared variables, as in fSMV, or more explicitly through *channels*. The global variable `buffer` is a channel of integers with a capacity of three. This means that it can hold three messages. A channel will refuse new messages to be sent once it is full; it will also refuse being read from once it is empty.

A process can have local variables, like `p` at line 4, which cannot be accessed by other processes. The behaviour of a process is specified in a procedural style. The `do` statement is used to declare a loop. A `do` loop can have several loop bodies (called *options*), each introduced with a double colon, `::`. The first statement of an option is the condition under which it can be executed. The loop at line 5 has one option with the condition `true`, which means that it can always be executed, effectively making the loop infinite.[2] `if` statements work similarly. The `if` statement at line 6 has two options. Their first and only statements are assignments, which can always be executed. When there are several options that can be executed like this, the choice is non-deterministic (this is different from `case` statements in SMV, where the first option would be taken). The `if` statement at lines 6 to 8 is thus used to assign a value to `p` non-deterministically.[3]

Unlike in a programming language, every statement in Promela will be blocked if it is not executable. At line 9, the content of the variable `p` is written into the channel `buffer`. If the channel is full, execution will be blocked here until there is space in the channel. In essence, the sender process non-deterministically writes zeros and ones into a buffer. The receiver process indefinitely reads from the buffer (line 15). When reading from a channel, the underscore means that a message is discarded. If a variable is used, e.g., `buffer?var`, the message is written into the variable. In both cases, the message is removed from the channel, which frees up space.

Promela has a rather extensive syntax, much richer than that of SMV. This brief introduction only covered the most important constructs. Almost all constructs that exist in Promela are available in fPromela and SNIP, too. A full list of unsupported constructs is distributed with SNIP. Let us now proceed to fPromela. fPromela extends Promela with a new type, *feature variables*. Feature variables can be used to *guard* statements with *feature expressions*. The following example illustrates this.

Listing 9.2: A simple fPromela model.

```
1 // Declare features
2 typedef features {
```

[2] A loop can only be left with the `break` statement, even if it is not infinite. When the conditions of all options are $false$, execution will be blocked until one of them becomes $true$.

[3] When all conditions of an `if` statement are $false$, execution will be blocked until one of them becomes $true$. To avoid this, the `else` expression can be used in one of the conditions.

```
3     bool Foo;
4     bool Bar
5   };
6   features f;
7
8   active proctype toto() {
9     int i = 0;
10    // Guarded increment statement
11    gd :: f.Foo || f.Bar;
12          i++;
13       :: else;
14          skip;
15    dg;
16    // Test assertion
17    assert(i == 1);
18  }
```

The features used in a model have to be declared as fields of the special type `features`, which is done at lines 2-6. The reason for this is twofold: it serves as an interface that identifies the features used in the model and it ensures compatibility with Promela. The features can then be referenced by declaring any variable with this type (`f` in the example).

The example system consists of one process, specified at lines 8-18. As discussed in Section 9.1, variability in fPromela is expressed by *guarding* statements. Guard blocks use the `gd` keyword. The `i++` statement at line 12, for instance, is guarded with the expression `f.Foo || f.Bar` (line 11). This means that the `i++` statement is only part of products containing features *Foo* or *Bar*. The other products (line 13) do nothing (line 14). A `gd` statement works like an `if` statement, except that only feature variables or the `else` keyword can be used in the first statement (the condition) of its options. In fact, this is the only place where feature variables may be used. They cannot be accessed anywhere else, be written to or printed. In the language of the C preprocessor, the above guard would be written

```
1  #ifdef FOO || BAR
2    i++;
3  #endif
```

Where `FOO` and `BAR` are directives that are set at compile time if the corresponding features are to be included. Variability in fPromela is thus expressed in a way that is very similar to how it would be expressed in a programming language such as C. Like in C, any fPromela statement can be guarded and guards can be nested. However, unlike `#ifdef`s, the `gd` statements in fPromela are part of the language and its grammar. This way, we avoid the problems that exist when parsing C code with `#ifdefs` [Garrido and Johnson, 2005, Kästner

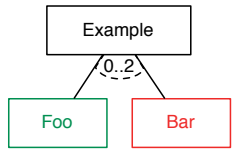


Figure 9.1: Graphical rendering of the TVL FD in Listing 9.3.

et al., 2010], and any product is guaranteed to be syntactically correct. Note that the `else` in fPromela has to be specified (it is not required by the C preprocessor). Otherwise, execution of all other products will be blocked at this point (which is consistent with the `if` in Promela).

Independently from this, we should note that directives of the C preprocessor can be used in fPromela, too—not to specify variability, but to simplify the model, define constants, decompose it into several files, and so on. This is very helpful for specifying properties, and is required to make SNIP compatible with SPIN. Further note that `gd` and `dg` are just aliases for `if` and `fi`. SNIP will distinguish guards from normal `if`s on its own. This way, a syntactically valid and well-typed fPromela file is also a syntactically valid and well-typed Promela file, provided that all `gd`s are replaced by `if`s.

Let us go back to the example from Listing 9.2. As expected, a guarded statement is part of the model of a product if its guard evaluates to *true* in the product. In the example, this means that `i` is only incremented in products containing features `f.Foo` or `f.Bar`. At line 17, the property that `i` equals one is specified using an assertion. Alternatively, properties can be specified using LTL, fLTL or directly as automata (that is, using *never claims* as in SPIN).

In addition to an fPromela file, SNIP requires an FD specified with TVL. A TVL feature model for the previous example would be the following.

Listing 9.3: TVL equivalent of the FD in Figure 9.1

```
1 root Example group [0..2] {
2   Foo,
3   Bar
4 }
```

In the graphical syntax introduced in Chapter 1, this would be rendered as shown in Figure 9.1. All features declared in the fPromela file have to exist in the TVL file; otherwise SNIP will report an error.

9.2.2 Semantics

The syntax of Promela (and hence fPromela) is rather vast, so that it would be tedious to define its full semantics here. We thus omit the less relevant details

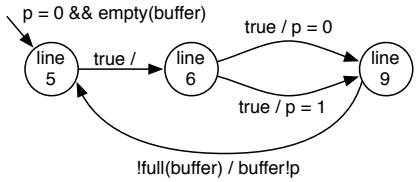


Figure 9.2: Program graph of the sender process from Listing 9.1.

of the semantics. The interested reader is referred to [Holzmann, 2004], which contains a precise account of Promela's semantics (in fact, [Holzmann, 2004] is the only reference used for the implementation of SNIP).

In order to see more clearly what a Promela or fPromela model represents, we define the abstract syntax of both languages. Each `proctype` of a model defines a *program graph*.[4] As an example, the program graph corresponding to the sender process of Listing 9.1 is shown in Figure 9.2.

A program graph is defined over a set of typed variables. The vertices of this graph are the control locations (i.e., the program counter, represented by the line number in Figure 9.2) and its transition relation defines the control flow. Each transition has a condition under which it can be executed, and an effect, i.e., a function that defines its effect on the set of variables. In Figure 9.2, the transitions are annotated with *condition / effect*. In Promela, the control statements such as `do`, `if` or the semicolon define the control flow. The only statements that end up on transitions are expressions, assignments (including channel reads and writes), assertions and print statements (not shown in our examples, but supported by SNIP). For the purpose of this discussion, we just consider expressions and assignments.

Definition 9.1. *Let types be the set of types in Promela,* $V = \{v_1, \ldots, v_k\}$ *a set of variables, and* $\tau : V \to$ *types their type function:* $expr(V)$ *denotes all Promela expressions over* V*, and* $asgn(V)$ *all assignments. Assuming that the variables are ordered,* $v \in \tau(v_1) \times \ldots \times \tau(v_k)$ *denotes a valuation of the variables, let* $val(V)$ *be the set of all valuations. For* $e \in expr(V)$*, we write that* $v \models e$ *if the expression evaluates to true for the values* v*. For* $a \in asgn(V)$*,* $apply(a, v) \in val(V)$ *denotes the valuation obtained after applying the assignment* a *to* v*.*

We define a program graph as a graph in which each transition is labelled with an expression (the condition) and an assignment (its effect). If a statement in a model has no effect on the variables, its assignment is the identity function. If a statement can be executed at all times, its condition is simply *true*. In addition, a program graph has an expression characterising the variable values in the initial state.

[4] Holzmann uses the term 'finite state automaton' [Holzmann, 2004]. We use 'program graph' [Baier and Katoen, 2008], since 'finite state automaton' is used for something else.

Definition 9.2. *A program graph over* (V, τ) *is a tuple* $(S, trans, I, init)$*, where* S *is a set of states and* $I \subseteq S$ *a set of initial states,* $trans \subseteq S \times expr(V) \times asgn(V) \times S$ *is the transition relation, and* $init \in expr(V)$ *is an expression characterising the variable values in the initial state.*

The semantics of a program graph G*, noted* $[\![G]\!]$*, is a transition system* $(S', Act', trans', I', expr(V), L')$ *where*

- $S' = S \times val(V)$*, that is, each state denotes a control location and a variable valuation of the program graph;*
- $I' = \{(i, v) \mid i \in I, v \in val(V) \bullet v \models init\}$*, the initial states are the initial control locations and valuations satisfying init;*
- $Act' = \{\epsilon\}$*, actions are not required here, so each transition is labelled with a dummy action* ϵ*;*
- $L'\big((s, val)\big) = \{e \in expr(V) \mid val \models e\}$*, each state is labelled with the expressions satisfied by its variable valuation;*
- *Transitions can only be executed when the expression evaluates to true for the variable valuation in the start state; they change the variable valuation in the end state according to the assignment: for* $e \in expr(V), a \in asgn(V)$ *and* $v \in val(V)$*,*

$$\frac{(s, e, a, s') \in trans \wedge val \models e}{(s, val) \xrightarrow{\epsilon}{}' (s', apply(a, val))}.$$

The program graph corresponding to a Promela file is obtained by transforming control statements into vertices. In a nutshell, this is done as follows. Single expressions terminated by a semicolon correspond to states with single outgoing transitions. An example is the transition from *line 5* to *line 6* in Figure 9.2. An `if` statement becomes a state with one outgoing transition per option. This transition is labelled with the first expression of the option. The last state of all options leads back to a common state. In the example of the sender, there is just one transition per option, which is why both transitions leaving *line 6* lead to the same state in Figure 9.2. A `do` statement is similar to an if statement, except that the last transition of all options leads back to the beginning of the `do` statement. Assignment statements are always executable, their condition is thus *true*. Channel writes are only executable if the channel is not full. In Promela, variables are implicitly initialised at zero and channels are empty; hence the expression on the initial transition in Figure 9.2.

A fragment of the transition system corresponding to the program graph of the sender is shown in Figure 9.3. The transition system has no infinite behaviours. Every behaviour stops in a state where the control location is *line 9* and the buffer contains three elements. The buffer is thus full and execution blocks at *line 9*. To obtain a system with no finite behaviours, the program graph of the sender and the one of the receiver have to be put in parallel. The

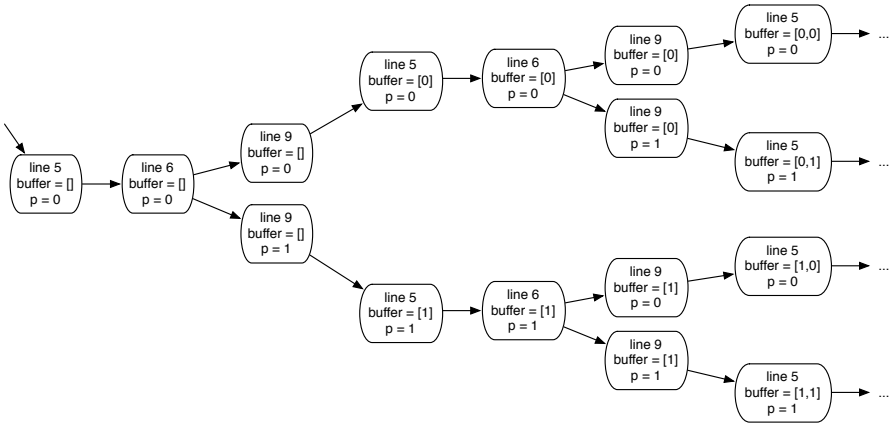


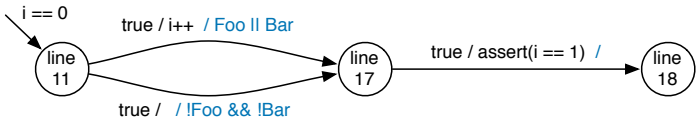
Figure 9.3: Transition system of the program graph of Figure 9.2.

parallel composition of two program graphs is obtained by interleaving their executions. One exception are *rendez-vous channels*, which are channels of capacity zero. Reads and writes of rendez-vous channels have to occur together, at the same time, which means that these transitions are synchronised. The parallel composition of two program graphs results in a program graph over the union of their variables. We do not go into the details of this definition. It is very similar to Definition 2.4, except that it takes shared variables into account. (Note that the parallel composition of two program graphs is not equivalent to the parallel composition of their transition systems; the latter is inconsistent as it does not take shared variables into account.)

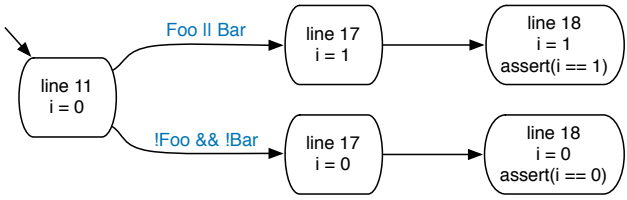
As expected, the semantics of a Promela model is a transition system. The actual semantics is much more intricate, but Definition 9.2 captures the gist of it. An fPromela model, in turn, describes a *featured program graph*. A featured program graph is a program graph in which transitions are annotated with feature expressions.

Definition 9.3. *A featured program graph over variables (V, τ), and an FD d with features N, is a tuple $(S, trans, I, init, \gamma)$, where $S, I, trans$ and $init$ are defined as in Definition 9.2 and $\gamma : trans \to \mathbb{B}(\{f_1, \ldots, f_n\})$ annotates transitions with a feature expression.*

The semantics of a featured program graph is an FTS $(S', \{\epsilon\}, trans', I', expr(V), L', d, \gamma')$ where S', I' and $trans'$ are defined as in Definition 9.2. Note that there is a clear correspondence between transitions of the program graph and those of the transition system. In the FTS, γ' is such that for any transition $t' \in trans'$, let $t \in trans$ be the corresponding transition of the program graph, then $\gamma'(t') = true$ if $\gamma(t)$ is not defined (i.e., for an unguarded transition) and $\gamma'(t') = \gamma(t)$ otherwise.



(a) The featured program graph of the model.



(b) The corresponding FTS.

Figure 9.4: The fPromela example from Listing 9.2.

The featured program graph of an fPromela file is obtained in a way similar to obtaining the program graph of a normal Promela file. The only difference is the treatment of `gd` statements. The featured program graph of a Promela file would be equivalent to its program graph. Remember, the first statement of an option of a `gd` statement is the feature expression. The other statements are the guarded behaviour. In terms of the featured program graph, this means that a `gd` statement is treated like an `if` statement, except that the fist statement acts as the feature expression of the *second* statement. If the first statement of an option block is `else`, the feature expression is the negation of the disjunction of the feature expressions of the other options in the `gd` statement.

To illustrate this, the featured program graph corresponding to the fPromela example from Listing 9.2 is shown in Figure 9.4(a). The feature expression labels were added in colour behind the existing labels. All unguarded statements, like the `assert` at line 17, are not labelled. The corresponding FTS is shown in Figure 9.4(b).

The semantics of an fPromela model is an FTS. As for Promela, the actual semantics is much more intricate than Definition 9.3. In fact, it follows the semantics of Promela (given in [Holzmann, 2004]) exactly, just adding feature expressions from the featured program graph to the transitions. The parallel composition of two featured program graphs is defined in the same way as for program graphs, and is similar to Definition 4.7. Each transition of the resulting featured program graph corresponds to a transition of one of the featured program graphs, whose feature expression it inherits. An exception are rendez-vous transitions, whose feature expressions are the conjunction of those of the transitions being executed in parallel.

Just like FTS and transition systems, featured program graphs and program graphs are related by a projection operation. The following definition is analogous to Definition 4.2.

Definition 9.4. *Given a featured program graph* $fG = (S, trans, I, init, \gamma)$ *with FD* d*, its projection to a product* $p \in [\![d]\!]_{FD}$*, noted* $fG_{|p}$*, is the program graph* $(S, trans', I, init)$ *where* $trans' \triangleq \{t \in trans \mid t \notin dom(\gamma(t)) \vee p \models \gamma(t)\}$.

Syntactically, the projection operation can be accomplished as follows.

Algorithm 9.5. *Given an fPromela model with FD* d*, its projection to a product* $p \in [\![d]\!]_{FD}$ *can be obtained as follows:*

(1) remove all feature variable declarations;

(2) replace the feature expressions of all `gd` *statements by the value they take for* p*;*

(3) remove the feature expressions that evaluate to true;

(4) replace all `gd` *statements by* `if` *statements.*

The obtained model is a syntactically valid Promela model. Since feature variables are guaranteed to only appear in feature expressions, removing them will not lead to bad references.

Theorem 9.6. *Given an fPromela model with FD* d *and featured program graph* fG*, the reduction to a product* $p \in [\![d]\!]_{FD}$ *computed according to Algorithm 9.5 yields a Promela model whose program graph is semantically equivalent to* $fG_{|p}$.

Proof sketch. It should be clear that without steps (2) and (3) of Algorithm 9.5, the resulting program graph would have at least the transitions and states of $fG_{|p}$. The effect of step (2) is that all feature expressions that evaluate to $false$ become transitions which can never be executed. This is equivalent to removing them. Furthermore, the transitions in fG whose feature expression evaluated to *true* (that are thus in $fG_{|p}$) are now all prefixed with a single transition with the expression *true* (the evaluated feature expression). The effect of step (3) is to remove these transitions. The resulting program graph is thus indeed semantically equivalent to $fG_{|p}$. □

An immediate consequence of this is that the fPromela semantics preserves the Promela semantics: an fPromela model without any guarded statements can be interpreted as an fPromela or as a Promela model. The first interpretation yields an FTS and the second yields a transition system, both of which have exactly the same behaviours. For the fPromela example from Listing 9.2, the projection to the product $\{Example, Foo\}$ is the following:

```
1 active proctype toto() {
2   int i = 0;
3   i++;
4   assert(i == 1);
5 }
```

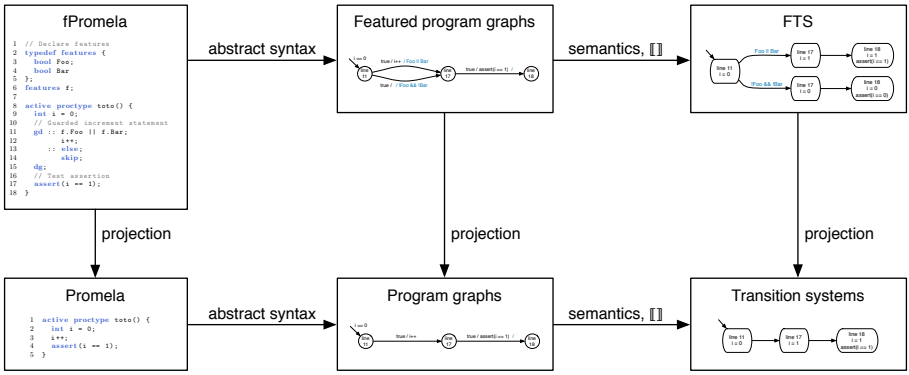


Figure 9.5: Relation of fPromela, Promela, FTS and transition systems.

The relation between fPromela and Promela is thus very similar to that between FTS and transition systems, as is the relation between fPromela and FTS and Promela and transition systems. This is best illustrated by the diagram in Figure 9.5. The diagram is commutative, as Theorem 9.6 and the following theorem establish.

Theorem 9.7. *For any fPromela model with FD d and featured program graph fG, for any product $p \in [\![d]\!]_{FD}$,*

$$[\![fG\,|_p]\!] \equiv [\![fG]\!]\,|_p$$

where $\equiv$ means that both transition systems are trace equivalent.

Proof sketch. The semantics of a program graph is defined in the same way as the semantics of a featured program graph. Furthermore, feature expressions are treated in the same way by FTS projection (Definition 4.2) and by the projection of a featured program graph (Definition 9.4). In consequence, the order in which these operations are applied does not matter. Hence, $[\![fG\,|_p]\!]$ is syntactically equivalent to $[\![fG]\!]\,|_p$. □

This concludes our discussion of the semantics of fPromela. Before we proceed to study its expressiveness, we would like to point out that there is also a much easier method to implement syntactic projection.

Algorithm 9.8. *Given an fPromela model with FD d, its projection to a product $p \in [\![d]\!]_{FD}$ can be obtained as follows:*

(1) initialise all feature variables according to p;

(2) replace all `gd` *statements by* `if` *statements.*

The advantage of this method is that it does not require the whole fPromela file to be parsed. As `gd` can be replaced by `if` in an fPromela file anyway, all that needs to be done is to initialise the feature variables. For the example from Listing 9.2, this would yield the following.

```
1  typedef product {
2    bool Foo = 1;     // initialise
3    bool Bar = 0      // initialise
4  };
5  product f;
6
7  active proctype toto() {
8    int i = 0;
9
10   if :: f.Foo || f.Bar;      // gd becomes if
11         i++;
12      :: else;
13         skip;
14   fi;
15
16   assert(i == 1);
17 }
```

The principal difference to Algorithm 9.5 is that the feature expressions at line 10 and line 12 remain as transitions. This changes nothing for those that evaluate to *false* since they can never be taken. Those that evaluate to *true*, however, are additional transitions that have to be executed before the actual guarded statement is executed. This leads to two discrepancies: first, these transitions will create stutter transitions in the underlying transition system (i.e., transitions that do not modify the atomic propositions). Secondly, they might introduce new non-determinism. This is more problematic. Consider the following case:

```
1  gd :: f.Foo;
2        chan!1;
3     :: true;
4        skip;
5  dg;
```

The channel write statement is only part of products with feature A, whereas the `skip` is part of all products. A discrepancy arises in products with feature A. First, consider projection according to Algorithm 9.5: when the execution gets to the above statements and the channel is full, the system will always take the `skip` transition. If projected according to Algorithm 9.8, the channel write will be prefixed by a `true` transition. Now, when the execution gets to the guarded

statement, the system has the non-deterministic choice of taking the `skip` or the `true` transition. If the `true` transition is taken, the execution will be blocked at the channel write statement. If the channel remains full indefinitely, this leads to a deadlock which does not exist in the other projection. The problem is that the `true` transition introduced non-determinism which did not exist in the actual system. A necessary condition for this problem to occur is that the guarded statement is not *exclusive*, i.e., some of the product sets defined by its feature expressions overlap. Otherwise, the `true` transition would be the only one, and thus cannot introduce new non-determinism. This is formalised by the following theorem.

Theorem 9.9. *Given an fPromela model with FD d and featured program graph fG, the reduction to a product $p \in [\![d]\!]_{FD}$ computed according to Algorithm 9.8 yields a Promela model whose program graph G is* stutter trace equivalent *[Baier and Katoen, 2008] to $fG_{|p}$ if all* `gd` *statements are* exclusive. *Exclusive means that the sets of products defined by the feature expressions of a* `gd` *statement are disjoint.*

Proof sketch. In the resulting Promela model, the feature variables are normal Boolean variables. By definition of fPromela, they are never written to, which means that all feature expressions will always evaluate to the same value. The resulting program graph G thus corresponds to that of Algorithm 9.5 in which step (3) was not executed. As stated in the proof of Theorem 9.6, the transitions in fG whose feature expression evaluated to *true* are now prefixed with a transition (the former feature expression) whose expression always evaluates to *true*. These *true* transitions lead to a stutter transition in $[\![G]\!]$, since they do not change the variables. However, $[\![G]\!]$ is only stutter trace equivalent to $[\![fG_{|p}]\!]$ if the *true* transitions do not introduce new non-determinism. If the sets of products of a `gd` statement are disjoint, it will have at most one *true* transition in any product. In this case, the *true* transitions do not introduce new non-determinism. □

In practice, the feature expressions of a `gd` statement are almost always disjoint. Furthermore, stutter trace equivalence preserves all LTL properties that do not use the $\bigcirc$ operator, which is almost never used. Because of this, and because of its ease of implementation, we use Algorithm 9.8 to implement projection (the input to SPIN) in our benchmarks.

The above theorem has a corollary which yields an alternative, much more intuitive, semantics for fPromela.

Corollary 9.10. *Each fPromela model is semantically equivalent to the non-deterministic choice between 2^n Promela models (where n is the number of feature variables) obtained by varying the initial values of the features. (Provided that the feature expressions of all* `gd` *statements are disjoint.)*

9.2.3 Expressiveness

By Definition 9.3, any fPromela model can be translated into an equivalent FTS. The fPromela modelling language is thus a subset of the FTS language. It is rather easy to show that the converse holds as well, i.e., that both languages are expressively equivalent (except for the action labels which exist in FTS).

Theorem 9.11. *Any FTS can be translated into fPromela.*

Proof sketch. Let $(S, Act, trans, I, AP, L, d, \gamma)$ be an FTS. An equivalent fPromela model with FD d can be obtained by encoding the transition relation with `goto` statements (which work similar to C). Basically, each state becomes a program location and `goto`s are used to jump from location to location reflecting the transition relation. Each `goto` thus corresponds to one transition and is guarded with the feature expression of the transition. For each state $s \in S$ with its outgoing transitions, this yields:

```
1 // One label identifying the state:
2 state_s:
3 // If the state is an initial state, a second label:
4 init_s:
5   // For each target state one option with a goto:
6   gd :: feature expression;
7         goto state_target;
8      :: ...
9      dg;
```

The initial states are modelled as follows:

```
1 // One goto per initial state
2 if :: goto init_state;
3    :: ...
4    fi;
```

The features of the FTS are declared under the `features` type as shown in Section 9.2.1. The program graph G of an active `proctype` with this behaviour will have $|S| + 1$ control locations, one for each state, plus the additional initial state. Since there are no variables, $[\![G]\!]$ is syntactically identical to the input FTS, except for the additional initial state. □

9.3 SNIP

Let us now introduce SNIP. First, we present its user interface and illustrate it with several examples. We then discuss its architecture, various implementation

choices and third-party libraries used. Finally, we describe how the algorithms of Chapter 6 were implemented.

9.3.1 User interface and illustration

The user interface of SNIP is designed to take into account all the SPL model checking use cases presented in the introduction to Part III. It also addresses a variety of practical concerns that the user might have, like simulation, bounded checking, layout of counterexamples, and so on.

As for most model checkers, SNIP's use consists in launching checks with certain parameters (property, execution bound). A very efficient interface for this is the command line; it remembers past commands and keeps a trace of inputs and outputs. SNIP is thus a command-line application. The list of its parameters is shown when launching SNIP without parameter.

Introductory example and assertion checking

As input, SNIP requires an fPromela file, a TVL file and a property. For our first illustration, we use the example from the previous section, where the fPromela file is given in Listing 9.2 and the TVL file in Listing 9.3. In this case, the property is the assertion at line 17 of the model. To check it, SNIP would be executed as follows.

```
$ ./snip -check -fm features.tvl model.pml
No never claim, checking only asserts and deadlocks..
Assertion at line 17 violated [explored 5 states, re-explored 0].
 - Products by which it is violated (as feature expression):
   (!Foo & !Bar)

 - Stack trace:
   features                              = /
   pid 00, toto                          @ NL11
   toto.i                                = 0
    --
   features                              = (!Foo & !Bar)
   pid 00, toto                          @ NL14
    --
   features                              = (!Foo & !Bar)
   pid 00, toto                          @ NL17
    --
    -- Final state repeated in full:
   features                              = (!Foo & !Bar)
   pid 00, toto                          @ NL17
   toto.i                                = 0
    --
```

The `-check` parameter activates SNIP's model checker. If it is set, SNIP automatically checks all assertions and looks for deadlocks. The `-fm` parameter specifies the feature model. This parameter can be omitted if the TVL file

has the same name as the fPromela file. That is, if the TVL file were named `model.tvl`, the preceding command-line can be shortened to:

```
$ ./snip -check model.pml
```

The output consists of two parts. First, SNIP reports the products for which the property is violated in the form of a feature expression (line 5). Second, SNIP gives a counterexample, that is, an execution of the fPromela model which proves the property violation (line 7 and following). It is presented as a sequence of states separated by double dashes. For each state, SNIP prints the products that can reach the state as a feature expression ('`/`' means all products), the position inside each process (`pid 00, toto @ NL11` means the process with id 0, of type `toto` is at line 11), and the values of the variables. At line 3, SNIP also reports two statistics: the number of states that were explored and re-explored. The explored states are the states that were visited and stored in memory; the re-explored states are visited states that had to be explored again (see also Section 6.3).

To make counterexamples shorter and easier to understand, variables are only printed if their value changed. Furthermore, the last state is repeated in full so that the user can work backwards. There are two options to control the output of counterexamples: `-nt` disables them (very useful if the user is only interested in the satisfying products), and `-st` prints only states in which variable values changed (i.e., states in which processes do nothing are not shown.). Since SNIP's output is text and can be interpreted immediately (no need for an additional tool), it can be piped to other command-line tools such as *cat* or *grep*. This is very useful to filter the relevant variables out of long counterexamples.

For the example, SNIP reports that the assertion is violated by products that satisfy `!Foo & !Bar`. This is as expected, since only those products lack the `i++` statement at line 12. In contrast to fNuSMV, SNIP implements both Mc and ExtMc. If only `-check` is specified, SNIP computes Mc. This means that SNIP stops as soon as it finds a violation. To compute ExtMc, the parameters `-check` and `-exhaustive` have to be set.

```
$ ./snip -nt -check -exhaustive -fm features.tvl model.pml
No never claim, checking only asserts and deadlocks..
Assertion at line 17 violated [explored 5 states, re-explored 0].
 - Products by which it is violated (as feature expression):
5    (!Foo & !Bar)

Exhaustive search finished  [explored 5 states, re-explored 0].
 - One problem found covering the following products (others
   are ok):
10 (!Foo & !Bar)
```

In this case, SNIP will print a violation upon finding it (line 3), and continues searching for violations in the other products. In the example, we disabled

printing of counterexamples using `-nt`, otherwise, SNIP will print a counterexample for each violation. When the search terminates, SNIP prints a summary with all the products found to violate (line 7). In this case, those are the same as before. However, we now have the certitude that all products satisfying `Foo | Bar` are free from violations.

Sender/receiver example and deadlock checking

The previous example is rather basic. It does not have infinite behaviours and does not use parallel composition. Let us modify the sender/receiver example given in Listing 9.1 by making each of the two processes optional. The FD in this case would be the following.

```
1 root Main group someOf {
2   Send,
3   Receive
4 }
```

The Promela model is transformed into the following fPromela model.

```
1  typedef features {
2    bool Send;
3    bool Receive
4  };
5  features f;
6
7  chan buffer = [3] of { int };
8
9  proctype sender() {
10   int p;
11   do :: true;
12         if :: p = 0;
13            :: p = 1;
14         fi;
15         buffer!p;
16      od;
17 }
18
19 proctype receiver() {
20   do :: true;
21         buffer?_;
22      od;
23 }
24
25 active proctype boot() {
26   gd :: f.Send;
27         run sender();
```

```
28        :: else;
29           skip;
30        dg;
31     gd :: f.Receive;
32           run receiver();
33        :: else;
34           skip;
35        dg;
36  }
```

Instead of declaring the sender and the receiver processes `active`, they are now started explicitly by the `boot` process, using the `run` statement. Each `run` statement is guarded by the respective feature. This way, only products with the *Send* feature have a sender process, and likewise for the *Receive* feature. Checking this model yields the following.

```
$ ./snip -check -exhaustive -nt sendrcv.pml
No never claim, checking only asserts and deadlocks..
Found deadlock [explored 139 states, re-explored 0].
 - Products by which it is violated (as feature expression):
   (Send & !Receive)

Found deadlock [explored 202 states, re-explored 0].
 - Products by which it is violated (as feature expression):
   (!Send & Receive)

Exhaustive search finished  [explored 202 states, re-explored 0].
 - 2 problems were found covering the following products (others
   are ok):
(!Send & Receive) | (Send & !Receive)
```

SNIP finds two deadlocks, as expected. The counterexamples (disabled here for brevity) identify the deadlocked states. In the first case, the sender is started without a receiver. It will thus send messages until the buffer is full, at which point it waits indefinitely at line 15; a deadlock state. In the second case, the receiver is started without a sender. It deadlocks immediately at line 21 because the buffer will always remain empty.

It might seem that this is inconsistent with the previous example. It too has only finite behaviours, and yet SNIP did not report a deadlock. This is because its finite behaviours all end with a terminal state of the program graph (in this case, the end of the process specification). In Promela and fPromela, a state with no outgoing transitions is not a deadlock state if all processes are in terminal states. In the deadlock states of the sender/receiver example, the `boot` process is in a terminal state, whereas the `sender` (or `receiver`) is not.

In Section 5.4.2, we remarked that in FTS, deadlocks can also stem from erroneous feature expressions. Consider the following example in which A is a single optional feature.

```
1  typedef features {
2    bool A
3  };
4  features f;
5
6  active proctype foo() {
7    int i = 0;
8    gd :: f.A;
9          i++;
10   dg;
11   i++;
12 }
```

The guard at line 8 only considers products with feature A. For all other products, there will be no transition in this state. Those products are deadlocked, as the `foo` process is blocked in a non-terminal state. In SNIP, such special deadlocks states are called *trivially invalid end states*; 'trivially', because they can be very easily avoided by making sure each `gd` statement has an `else` option. By default, SNIP will not check for trivially invalid end states. In contrast to simple deadlock checking, it requires a small computation each time, which might be costly. If SNIP is run normally, this yields.

```
$ ./snip -check -nt deadlock.pml
No never claim, checking only asserts and deadlocks..
No assertion violations or deadlocks found [explored 2 states,
re-explored 0].
```

To activate checking of trivially invalid end states, and compute CheckDeadlock as defined in Definition 5.6, the `-fdlc` parameter has to be set.

```
$ ./snip -check -exhaustive -fdlc -nt deadlock.pml
No never claim, checking only asserts and deadlocks..
Found trivially invalid end state; the following set of products
can reach the state, but has no outgoing transition. [explored 1
states, re-explored 0].
 - Products by which it is violated (as feature expression):
   (!A)
```

SNIP then correctly identifies the products without feature A as violating.

Mine pump example and fLTL model checking

So far, we have shown how assertions and deadlocks are checked. Of course, properties can also be specified using LTL, fLTL or directly as *never claims*.

To illustrate this, we use the mine pump example from Section 4.5.2. The fPromela model is not shown here. The use of fPromela allowed us to be more detailed and faithful to the CONIC code in [Kramer et al., 1983]. The state space of the model is thus considerably larger than that of the FTS in

Section 4.5.2. In terms of code, it consists of about 200 lines of fPromela. Recall that the 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 TVL FD of this model is the following.

Listing 9.4: FD of the mine pump controller product line.

```
root MinePump {
  group allOf  {
    opt Command group someOf {
      Start,
      Stop
    },
    opt MethaneSensor group someOf {
      MethaneAlarm,
      MethaneQuery
    },
    WaterSensor group [0..*] {
      Low,
      Normal,
      High
    }
  }
}
```

It is slightly different from the earlier one in Figure 4.21. We split up methane detection into two features, corresponding to the two mechanisms used. With the *MethaneAlarm* feature, the controller is notified when there is methane in the mine (it is passive). With the *MethaneQuery*, the controller queries the methane sensor each time before starting the pump (it is active).

The model contains a large number of properties (42). Here, we focus on one such property: *"There is never a situation in which the pump runs indefinitely even though there is methane."*; in LTL this becomes $\neg\Diamond\Box(pumpOn \wedge methane)$, and in the syntax used by SNIP: `!<>[] (pumpOn && methane)`.

Checking this property with SNIP yields the following.

```
$ ./snip -check -exhaustive -nt
         -ltl '!<>[] (pumpOn && methane)' minepump.pml
Checking LTL property !<>[] (pumpOn && methane)..
Property violated [explored 481 states, re-explored 0].
 - Products by which it is violated (as feature expression):
   (Start & Stop & MethaneQuery & MethaneAlarm & Low & High)

[...]

Property violated [explored 12806 states, re-explored 65409].
 - Products by which it is violated (as feature expression):
```

```
   (Start & !Stop & !MethaneQuery & !MethaneAlarm & !Low & High)

Exhaustive search finished  [explored 17325 states,
re-explored 179937].
 - 16 problems were found covering the following products (others
   are ok):
(Start & High)
```

The property is specified with the `-ltl` parameter. SNIP finds 16 violations and concludes that all products with `Start & High` violate the property. This is not what we expected, as the property is supposed to be satisfied by the system. Products without *Start* or *High* will never even start the pump, which is why they satisfy the property.

A look at the counterexamples reveals a problem with the property. Basically, the controller has a central loop, in which it can receive three types of messages: user commands (start and stop), methane alarm messages, and water level readings. The counterexamples 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. This assumption can be specified in LTL as follows: `(([]<> readCommand) && ([]<> readAlarm) && ([]<> readLevel))`.

```
$ ./snip -check -exhaustive -nt
         -ltl '([]<> read..) -> (!<>[] pump..)' minepump.pml
Checking LTL property ([]<> read..) -> (!<>[] pump..)..
Property violated [explored 27428 states, re-explored 125153].
 - Products by which it is violated (as feature expression):
   (Start & Stop & MethaneQuery & !MethaneAlarm & Low & High)

[...]

Property violated [explored 30157 states, re-explored 162316].
 - Products by which it is violated (as feature expression):
   (Start & !Stop & !MethaneQuery & !MethaneAlarm & !Low & High)

Exhaustive search finished  [explored 34356 states, re-explored
274456].
 - 8 problems were found covering the following products (others
   are ok):
(Start & !MethaneAlarm & High)
```

This result can be interpreted as saying that the *MethaneAlarm* feature is responsible for making the property true. This corresponds to what we expected, as the *MethaneAlarm* feature alerts the controller of methane, leading it to shut off the pump. Following the guidelines of Section 5.4.1, the assumption has to be discharged. This is done by checking its negation.

```
$ ./snip -check -exhaustive -nt
         -ltl '!(([]<> read..)' minepump.pml
```

```
Checking LTL property !(([]<> read..)..
Property violated [explored 2169 states, re-explored 0].
 - Products by which it is violated (as feature expression):
   (Start & Stop & MethaneQuery & MethaneAlarm & Low & High)

[...]

Exhaustive search finished [explored 8091 states, re-explored
37323]
 - 38 problems were found covering every product.
```

The assumption is thus discharged by all products.

Normally, the example property is not expected to hold for products that do not have the *MethaneAlarm* feature. It corresponds to a requirement implemented by the feature. This can be expressed with a quantifier in fLTL:

$$[MethaneAlarm]\ \neg\Diamond\Box(pumpOn \wedge methane).$$

In SNIP, the quantifier of an fLTL property is specified in TVL syntax, separately from the LTL property with the `-filter` parameter.

```
$ ./snip -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 27893 states, re-explored 248254].
```

The property is thus indeed satisfied by all relevant products. SNIP recalls in the output that the property is checked over a subset of the products (line 5).

9.3.2 Architecture and third-party libraries

SNIP is entirely written in the C programming language. An overview of its architecture is shown in Figure 9.6. The core of SNIP is divided into layers, so that lower layers are unaware of and have no dependency on upper layers. Each layer has access to a set of wrapped libraries. A wrapper consists of an interface (a header file) against which other code is written, and one or more implementations of the interface depending on the third-party library used. Third-party libraries are thus all wrapped and can easily be replaced. The library to be used for a wrapper is chosen at compile time. Let us first look at the core of SNIP, before we survey the third-party libraries used.

To create the fPromela parser, we use the parser generators Flex and Bison. These tools are highly efficient and the de-facto standard for creating parsers in C. To make sure that a Promela file in SNIP is parsed in the same way as it is in SPIN, we reused the Bison grammar specification from the SPIN source code.[5] As the model is parsed, SNIP fills a symbol table with global variables

[5]This is indeed the only piece of SPIN source code reused in SNIP.

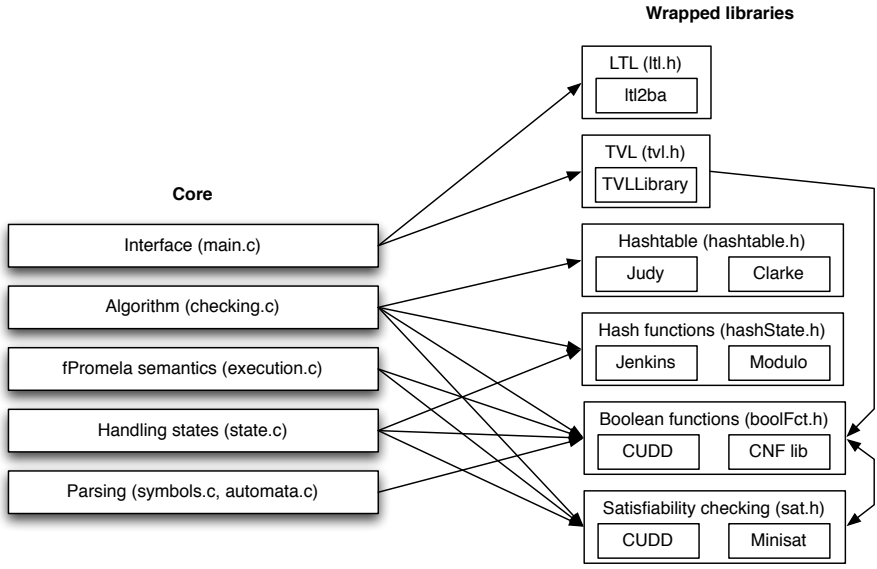


Figure 9.6: Architecture of SNIP.

and process definitions. The body of a process is represented by a featured program graph, which is created at the same time the model is parsed (it is built backwards). We make extensive use of doubly linked lists, one of the primary data structures in SNIP. As the featured program graph is built, all feature expressions are transformed into Boolean function objects (the actual type depends on the library chosen for representing Boolean functions). Furthermore, all references inside expressions are resolved and replaced by pointers to the respective symbols.

The state layer implements functions for the representation and manipulation of system states in memory. We do not use state compression [Holzmann, 1997] in SNIP. To make state manipulation reasonably efficient nonetheless, all variables are stored in a block of memory (the *payload*), rather than in linked lists. Blocks of memory can be copied and compared efficiently with built-in functions. The payload holds the global variables as well as those of the processes. Since all variables have a fixed size, we only need to keep track of the address in the payload at which the variables of a process start. A state further contains a Boolean function, which characterises the products for which it is reachable. The state layer handles dynamically created processes and channels.

The execution layer implements the semantics of Promela and fPromela. To make sure that the Promela semantics is correctly implemented, we follow the operational Promela reference [Holzmann, 2004] very closely. We thus have a function `executables`, which determines the transitions that can be executed in a state. It takes the feature expression of the state into account, as well as

the feature expressions of the candidate transitions. It can be thought of as an implementation of the *Post* operator from Definition 6.16. The other functions that are derived from the Promela specification are `eval`, which evaluates an expression, and `apply`, which executes a transition. The execution layer has also functions for simulation and for managing the execution stack of the DFS.

On top of this sits the model checking layer. It implements the procedure `Reachables` from Section 6.3, both as a nested DFS (for LTL) and as a normal DFS (for asserts and deadlocks). Visited states are stored in a hash table. The model checking layer will be discussed in more detail in the following section.

The interface layer pieces all of the other layers together. It is also responsible for most preparatory tasks. It interprets the command-line parameters and writes them to global variables. It transforms the LTL property into a Büchi automaton which it appends as a *never claim* to the input file. It then runs the C preprocessor on the input file and launches the parser. The interface layer also guesses the name of the TVL model, appends the fLTL guard (`-filter` parameter) to the model (as described in Section 6.2.4) and transforms it into a Boolean function. A number of temporary files are generated for this, which can be preserved if SNIP is executed with the `-t` parameter.

SNIP uses several third-party libraries. As noted before, the Promela grammar is taken from SPIN. To automate the transformation from LTL to Büchi automata, we use LTL2BA,[6] a very efficient implementation based on the results of [Gastin and Oddoux, 2001]. To parse TVL models and transform them into DIMACS, we use the TVL library.[7] DIMACS is a data exchange format for Boolean functions in CNF. The TVL library is written in JAVA and rather inefficient (even small models take a second to be parsed and transformed). Therefore, SNIP also allows the user to specify the FD in DIMACS directly. For this, SNIP has a command-line switch `-fmdimacs`, which has to be followed by a file in DIMACS format, and a dictionary file. Variable names in a DIMACS file are all integers. The dictionary file lists the feature names of the integers used in the DIMACS file. The TVL library can export these files.

To store large sets of states with efficient lookup times, we use a resizable hash table implementation by Clark.[8] We manage collisions ourselves using linked lists. As an alternative to this hash table implementation, we started the development of a hash table based on Judy arrays.[9] However, at the time of writing, this implementation is work in progress and not ready for usage. The hash function we use is due to Jenkins.[10] A simpler alternative hash function (repeated application of modulus) remains for testing purposes.

For the internal representation of Boolean functions, we currently have two alternatives: BDDs or CNFs. For BDDs, we use the CUDD package,[11] which

[6] www.lsv.ens-cachan.fr/~gastin/ltl2ba
[7] www.info.fundp.ac.be/~acs/tvl
[8] www.cl.cam.ac.uk/~cwc22/hashtable
[9] www.judy.sourceforge.net
[10] www.burtleburtle.net/bob/hash/doobs.html
[11] vlsi.colorado.edu/~fabio/CUDD

is also used in NuSMV for instance. The representation of Boolean functions is decoupled from the SAT checking of these functions. SAT checking in BDDs is accomplished in constant time. Thus, if CUDD is used, it has to be used for both (representation and SAT checking). The alternative representation, CNFs, relies on a self-written data structure. CNFs were mostly used during the early phases of development. A CNF quickly grows out of proportion, since there is (as of now) almost no minimisation. For SAT checking of CNFs we use MiniSat,[12] but any other SAT checker could be used as well. A challenge for SAT checking is that many checks have to be executed against the FD (See Section 6.2.3). The CNF representation of the FD is likely to be larger than the CNF being checked against it. To avoid having to load the CNF of the FD into the SAT solver each time, we use MiniSat's ability to check satisfiability under an assumption (a literal). Basically, the CNF of the FD is loaded once. For each CNF checked against it, a temporary variable is created which is appended as a literal to each clause of the CNF. The result is then checked under the assumption that the literal is false. After this, a new clause with the temporary variable as a single negative literal is added, which corresponds to removing the clauses added before. The SAT solver is reinitialised after a number of properties were checked (a constant set to 1000 currently), to keep the number of temporary variables reasonably low.

9.3.3 Implementing the model checking algorithms

Following this overview of SNIP's architecture, we discuss some of the implementation details, and relate them to the theoretical results of Chapter 6.

To conduct model checking, SNIP simulates the execution of the fPromela model. This means that (i) the calculation of the parallel composition of the processes, (ii) the calculation of the synchronous product of the processes and the never claim, and (iii) the generation of the resulting FTS according to Definition 9.3; are all conducted on the fly, i.e., on a per-need basis as the model checking algorithm is executed.

The model checking algorithm itself follows the `Reachables` procedure from Section 6.3 very closely. For simplicity, the procedure is implemented twice. Once as a nested DFS, which is used when an LTL property was specified (even if the LTL property is a reachability property, and thus the inner DFS is never started). In this implementation, the synchronous product with the Büchi automaton has to be calculated. This is not required when no LTL property is specified, which is why we implemented a simple DFS separately, which is used when no LTL property was specified. Checking of assertions and deadlocks is done in both cases and cannot be disabled (except for the `-fdlc` parameter discussed before) as there would be no noticeable speed gain.

In Section 6.2.3, we discussed two alternatives for making sure that only valid products are considered. One possibility is to seed the initial states with

[12] www.minisat.se

the Boolean function corresponding to the FD, the other is to test for each state whether its feature expression represents at least one valid product. In SNIP we use the latter: each time a new state is created, its feature expression is intersected with the BDD of the FD; if the intersection is empty, it is rejected. This has a number of advantages over the *seeding* method. Firstly, with the *seeding* method, the feature expression characterising the violating products that is returned as part of the output will also contain the Boolean function encoding of the FD, rendering it useless to the engineer. Secondly, the *seeding* method needs a data structure that exploits overlap in several instances. Basically, when the feature expression of all states contains the Boolean function equivalent of the FD, there will be a lot of redundancy. If the data structure used to represent feature expressions does not exploit this overlap to reduce the overall memory requirements, it will not scale. This would preclude using CNFs in this case. The CUDD package, however, does exploit overlap. After conducting experiments with both methods, though, we could not observe a noticeable difference in performance. We thus dropped the *seeding* method.

In Section 6.5, we discussed an optimisation for the ExtMc algorithm. It consists in maintaining a Boolean function characterising all violating products encountered so far, and avoiding these products in the search. SNIP has to maintain such a Boolean function already to be able to produce the summary information printed when the extended model check ends. The optimisation itself is combined with the check whether a state is reachable in valid products. As we said in the previous paragraph, the feature expression of each new state is intersected with the BDD of the FD to make sure that it contains at least one valid product. To implement the optimisation for the ExtMc algorithm, we exclude all violated products from the BDD of the FD. This way, the check required for the optimisation is conducted automatically when a new state is created, i.e., one BDD intersection instead of two.

The quantifiers of fLTL properties are implemented as described in Section 6.2.4, by appending them to the FD. This is done before the TVL library is called, which means that we do not even have to parse the quantifier. Since quantifiers are specified with a separate parameter, they can not only be used for LTL properties, but also for checking assertions and deadlocks.

In summary, when SNIP model checks an fLTL property $[\chi]\phi$, it proceeds as follows. The initialisation consists of three step. First, SNIP translates the LTL property ϕ to a Büchi automaton and appends it to the fPromela file as a never claim. Second, SNIP appends the quantifier χ as a constraint to the TVL model and transforms it into a BDD. Third, SNIP parses the fPromela file, creating one or more featured program graphs in the process. After the initialisation, SNIP launches the model checking algorithm. The algorithm computes the FTS corresponding to the parallel composition of the featured program graphs and their synchronous product with the never claim. It uses a depth-first search to compute the reachable states (stored in the hash table) and for each, the products in which it is reachable (in form of a BDD). For each new state, SNIP makes sure that it is reachable in a valid product which is

not yet known to violate the property. When a violating state is found, SNIP prints information about the violation: the feature expression characterising the violating products is obtained from the BDD, and a counterexample. If the `-exhaustive` parameter is set, SNIP continues the search and prints a summary of all violations when the algorithm finishes.

Finally, we would like to point out that SNIP has a parameter `-spin`, which causes it to interpret any input model as a Promela file. This means that feature variables are treated like normal Boolean variables, and that `gd` statements are treated like `if`s. The input will thus be interpreted as a featured program graph without feature expressions, i.e., a normal program graph. This corresponds to the syntactic projection described in Algorithm 9.8. In this case, no BDDs (not even trivial ones) will be computed and SNIP's model checking algorithm is equivalent to the classical model checking algorithm for single systems. We use this for our benchmarks, because it allows us to use SNIP to compute the naïve algorithm (Algorithm 6.2). We can then compare the naïve algorithm to the FTS algorithm where both are implemented by the same tool (even the same code). In an experiment measuring performance, this allows us to control many variables that would be impossible to control if different tools were used.

9.4 Experiments

As shown in Section 6.6, the computational complexity of our our algorithm is worse than that of the naïve algorithm. However, an experiment conducted with the Haskell FTS library [Classen et al., 2010b] showed that in practice, the semi-symbolic FTS algorithm is up to three times faster than the naïve algorithm. There were some limitations to this experiment, which were overcome with SNIP: it considered a limited number of properties (six), a small model (457 states in the FTS), and it did not measure the state space reduction.

We thus conducted new experiments with SNIP, considering three models. The first is the mine pump system [Kramer et al., 1983] discussed in Section 9.3.1. It has 11 features and 128 products; its FTS has 21 177 states, all products combined have 889 252 states. The second model is an elevator product line, similar to the one of Section 8.4, with two persons and four floors. It has 9 features and 256 products. The FTS of the elevator has 572 815 states, all products combined have 63 051 024. The third model represents a subset of the CCSDS file delivery protocol (CFDP) [Consultative Committee for Space Data Systems (CCSDS), 2007], with 10 features and 56 products; its FTS has 1 064 840 states, and the sum of all products combined leads to a transition system of 2 780 475 states. The fPromela models, including all properties and explanations, are distributed with SNIP [Classen, 2010b]. The full set of results is also available at the FTS website [Classen, 2010b].

9.4.1 Experimental setup

Our experimental setup consists of SNIP and a script that implements the naïve algorithms using SPIN and SNIP without the FTS algorithm (referred to as

'enum (snip)' in the statistics). The script uses the TVL library to list the set of valid products. For each, it transforms the fPromela input into a Promela file that describes the behaviour of the product (following Algorithm 9.8). It then first uses SNIP without the FTS algorithm, then SPIN, to model check the file. While the script makes up for the lack of functionality in SPIN (and SNIP without the FTS algorithm), it is still inferior in terms of usability. For instance, SNIP produces a Boolean expression characterising the violating products. As shown in Section 9.3.1, this expression identifies incompatible features, or features that are required for a property to hold. The script, in contrast, only lists the products that violate the property. The list has to be analysed again to produce information comparable to that returned by SNIP.

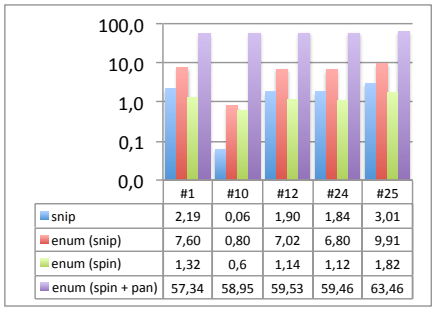
SNIP without the FTS algorithm provides a baseline to evaluate the impact of the FTS algorithm on the runtime and the size of the state space. A meaningful evaluation of the runtime cannot be done by comparison to tools such as SPIN, as it would require us to remove the bias introduced by optimisations for single systems and other implementation choices. The relevant comparison is thus between SNIP with and without the FTS algorithm. However, SPIN can be used to evaluate the ability of our algorithm to reduce the state space.

We only consider ExtMc in our experiments. The performance of the naïve Mc algorithm largely depends on the order in which products are checked, which we want to exclude as a factor.

Our experiments consist in using SNIP and the above script to compute ExtMc for all properties of the three examples. For each, we measured the runtime and the number of explored states. Recall that the FTS algorithm can re-explore states. The sum of the explored and re-explored states corresponds to the number of transitions fired. In the case of the naïve algorithm, this number is equal to the number of explored states. Henceforth, we will thus use 'number of transitions' rather than 'sum of explored and re-explored states'. To make measurements as fair as possible, the time counted for the naïve algorithm only includes the verification time. Moreover, in the case of SPIN, verification consists of three steps: (a) generating a process analyser (pan), (b) compiling it and (c) running it. The time for (b) was counted separately, as it is due to a design decision in SPIN rather than its model checking algorithm. All benchmarks were run on an Ubuntu machine with an Intel Core2 Duo at 2.80 GHz with 4 Gb of RAM.

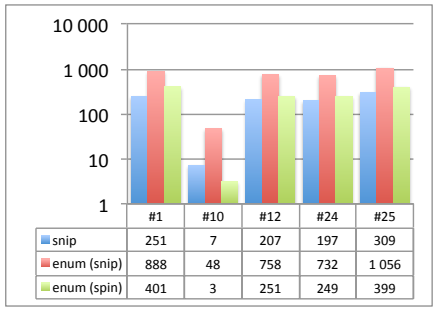
9.4.2 Mine pump

For the evaluation, we considered a deadlock check and 41 LTL properties, such as those identified in [Alrajeh et al., 2009]. The quantifiers of these fLTL properties ranged over the set of all products. The reason for this is that any fLTL property should initially be checked over all products, to make sure that use of the quantifier is warranted (e.g., the property might hold in all products, or be trivially satisfied). Of the 42 benchmark results, we only present data for (#1), the deadlock check, and for the following representative properties.



	#1	#10	#12	#24	#25
snip	2,19	0,06	1,90	1,84	3,01
enum (snip)	7,60	0,80	7,02	6,80	9,91
enum (spin)	1,32	0,6	1,14	1,12	1,82
enum (spin + pan)	57,34	58,95	59,53	59,46	63,46

(a) Runtime in seconds, logarithmic scale.



	#1	#10	#12	#24	#25
snip	251	7	207	197	309
enum (snip)	888	48	758	732	1 056
enum (spin)	401	3	251	249	399

(b) Number of transitions in thousands, logarithmic scale.

Figure 9.7: Benchmark results for the mine pump system.

(#10) The assumption that water level readings are received infinitely often, $\Box\Diamond readLevel$, is discharged by all products.

(#12) The assumption that the pump is switched on infinitely often, $\Box\Diamond pumpOn$, is discharged by products in $[\![Start \wedge High]\!]$.

(#24) A property that expresses the actual system requirement, that the system cannot be in a situation in which the pump runs indefinitely in presence of methane:

$$\neg\Diamond\Box(pumpOn \wedge methane).$$

This is the property used in the illustration in Section 9.3.1.

(#25) The same as property (#24) with the assumption that the system reads the various types of message infinitely often:

$$\big((\Box\Diamond readCommand) \wedge (\Box\Diamond readAlarm) \wedge (\Box\Diamond readLevel)\big) \\ \Rightarrow \neg\Diamond\Box(pumpOn \wedge methane).$$

This property is violated by products in $[\![Start \wedge High \wedge \neg MethaneAlarm]\!]$ and its assumption is discharged by all products. As discussed before, the *MethaneAlarm* feature prevents this kind of error. Let ϕ be property (#25), the fLTL formula $[MethaneAlarm]\phi$ is satisfied by all products.

The effect of quantification is noticeable, even if just one feature value is forced. For the extended model check of property (#25), SNIP reports eight counterexamples after exploring 34 356 states and re-exploring 274 456 in 2.96 seconds. To prove satisfaction of the quantified property, only 27 893 states are explored (248 254 re-explored), that is, 18% and 9% less; taking SNIP 2.62 seconds (11% less time).

The results for the previously mentioned properties are shown in Figure 9.7. The x-axis of both charts refers to the property IDs. Globally, the FTS algorithm in SNIP is from two to 45 times faster than the naïve algorithm implemented with SNIP. SNIP is especially efficient for properties that are violated by all products. This is due to the optimisation discussed in Section 6.5: as soon as SNIP finds a violating feature combination, it excludes it from the search. For other properties, the FTS algorithm is consistently between three and four times faster than the naïve algorithm.

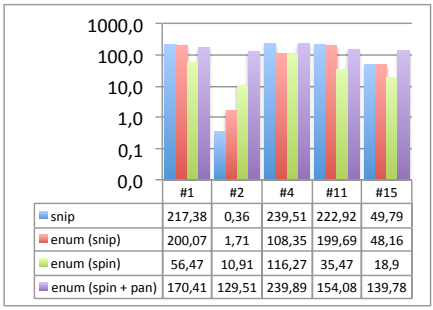
As stated before, a comparison to the naïve algorithm implemented in SPIN does not allow us to draw any conclusion as to the impact of the FTS algorithm. The results show that if the compilation time for SPIN's process analysers is not counted, SPIN generally outperforms SNIP by a factor of 1.65. Nevertheless, SNIP is generally faster on the properties violated by all products, e.g., six times faster for (#10). However, a fair comparison of SNIP and the script has to take all times into account. The compilation of a process analyser takes about 60 seconds and accounts, in average, for 99% of the total runtime. SNIP thus clearly outperforms the script of the naïve algorithm.

To measure state space reduction, we consider the baseline to be the number of transitions fired (equal to the number of states explored) by the naïve algorithm implemented with SNIP. The FTS algorithm in SNIP reduced the average number of states from 603 309 to 174 419 (71%), whereas SPIN reduced it to 206 970 (66%). This is important, as it shows that SNIP achieves greater reductions of the state space than the naïve algorithm using SPIN. A good example is given by property (#24) where the naïve algorithm (without pan compilation) is faster even though SNIP explores less states. This illustrates the extent to which SPIN is optimised.

We also measured the maximum amount of memory used by both algorithms during the verification of these properties. The property which requires the most memory is (#25). For its verification, the naïve algorithm (implemented with SNIP) needs 13.63 MB of memory whereas the FTS algorithm requires 20.12 MB. The memory requirements for the naïve algorithm depend on the product with the largest number of states. The transition system explored by the naïve algorithm has 14 954 states, which is considerably less than the 34 346 states of the FTS that were explored by the FTS algorithm. This accounts for the higher memory requirements of the FTS algorithm. Note that these numbers are rather small, especially considering that between 9.65 MB and 9.73 MB of memory is required for data structures that do not increase with the number of states (e.g., syntax trees, symbol tables, and so on).

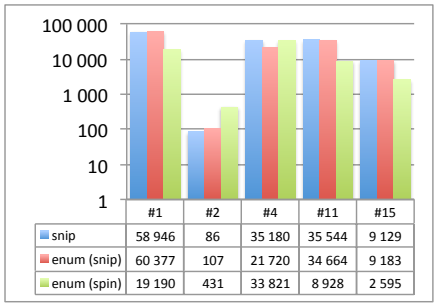
9.4.3 Elevator

The elevator system is based on the fSMV model created for the experiments in Chapter 8. All features except for the anti-prank feature were retained for the fPromela model. As before, they are independently optional, yielding 2^8 products. The behaviour of the lift and of the features is similar to what is



	#1	#2	#4	#11	#15
snip	217,38	0,36	239,51	222,92	49,79
enum (snip)	200,07	1,71	108,35	199,69	48,16
enum (spin)	56,47	10,91	116,27	35,47	18,9
enum (spin + pan)	170,41	129,51	239,89	154,08	139,78

(a) Runtime in seconds, logarithmic scale.



	#1	#2	#4	#11	#15
snip	58 946	86	35 180	35 544	9 129
enum (snip)	60 377	107	21 720	34 664	9 183
enum (spin)	19 190	431	33 821	8 928	2 595

(b) Number of transitions in thousands, logarithmic scale.

Figure 9.8: Benchmark results for the elevator.

described in Section 8.4. The main difference in the fPromela model is that persons using the elevator are modelled explicitly as processes. This fPromela model is the largest so far, with about 450 lines of code.

For the benchmarks, we considered a deadlock check and 19 fLTL properties. Here we show the results for (#1), the deadlock check, and the following representative properties.

(#2) The assumption that the main control loop is executed infinitely often, $\neg\Box\Diamond progress$, is discharged by all products.

(#4) The assumption that a person can visit each floor infinitely often:

$$\neg\Box\Diamond p0at0 \vee \neg\Box\Diamond p0at1 \vee \neg\Box\Diamond p0at2 \vee \neg\Box\Diamond p0at3$$

is also discharged by all products.

(#11) The property that it is impossible for cabin buttons to be pressed when nobody is inside:

$$\neg\Diamond(cb0 \vee cb1 \vee cb2 \vee cb3) \wedge \neg(p0in \vee p1in) \wedge dclosed).$$

This property is violated by all products that do not have the *Empty* feature (which is supposed to prevent such a situation).

(#15) The property that the door should never remain open indefinitely:

$$\Box\Diamond(progress \vee waiting) \Rightarrow (\neg\Diamond\Box dopen),$$

is violated by products with features *Park*, *OpenIfIdle* or without feature *QuickClose*, as expected.

The results are shown in Figure 9.8. In terms of runtime, for half of the properties, the FTS algorithm in SNIP is slightly slower than the naïve algorithm implemented with SNIP. It is only faster on properties that are violated

by all products. An explanation for this bad performance is given by the state space measures. On average, the FTS algorithm in SNIP explores 17 003 891 transitions, that is, even a bit more than the naïve algorithm which explored 16 780 379. SPIN explored 5 885 242, which corresponds to a reduction of 65%.

Note that it is impossible for the FTS algorithm to visit more states than the naïve algorithm. The only explanation for these measures is that there is a bug in SNIP which leads to a discrepancy in the Promela semantics of a model with feature expressions and without. The results for the elevator benchmarks should thus be taken with caution. At the time of writing, we have not been able to find this bug. The elevator model makes use of most Promela language constructs, increasing the likelihood for this kind of error to occur. This illustrates the challenges faced when implementing a complex language such as Promela. Note that the bug does not seem to affect the correctness of the properties; all properties checked in the these examples yield the same result with both algorithms.

As for the mine pump, we measured the maximum memory used by the two algorithms implemented in SNIP. In this case, the deadlock check, property (#1), required the most memory: 109.2 MB for the naïve algorithm and 155.7 MB for the FTS algorithm. These sizes correspond to 378 267 states in the transition system explored by the naïve algorithm and 533 332 states in the FTS. Note that the increase in the required memory is 43% whereas the number of states increased by 40%. The additional 3% are due to the BDDs representing the sets of products. The overhead they cause is thus rather small.

9.4.4 CFDP

The CFDP is a file delivery protocol for use in space missions [Consultative Committee for Space Data Systems (CCSDS), 2007]. The CFDP is highly configurable, and is thus suitable for a wide variety of missions. A mission usually only needs a subset of its functionality. To minimise the memory requirements of the CFDP, the non-required parts are not implemented. As part of a collaboration with Spacebel, a Belgian company that develops software for space missions, the CFDP specification was analysed and the protocol decomposed into features. This feature decomposition was subsequently used in the development of a CFDP library SPL [Boucher et al., 2010b]. We used it as the basis for our CFDP models, in which we consider a small subset of the protocol.

At the heart of the protocol is the transmission of files between CFDP entities, that is, spacecraft and ground stations. A transmission starts with the sender transferring a metadata segment to the receiver, followed by data segments composing the file to be transmitted. Once all data segments have been transmitted, the sending entity sends an *End-Of-File* (`EOF`) message and the receiver closes the transaction by sending a *Finished* (`FIN`) message.

Our experiment considers the efficiency of the different *Negative Acknowledgement* (`NAK`) procedures offered by the CFDP to detect and retransmit lost data segments. The protocol provides four `NAK` modes:

Deferred. The receiving entity waits until it receives the `EOF` message before it requests the missing data segments.

Immediate. The receiving entity requests a missing data segment as soon as it notices the loss.

Prompted. At any point in the transmission, the sender can prompt the receiver (using a `PROMPT` message) to request the retransmission of lost data segments. In addition, the receiver will request all missing data segments when the `EOF` message is received (as in *deferred* mode).

Asynchronous. At any point in the transmission, the receiver can request the retransmission of data segments lost up to this point.

Because we are concerned only by the variability in the `NAK` modes, we consider only a small subset of the FD created for [Boucher et al., 2010b]. This subset can be written in TVL as follows.

```
root CFDP {
  group allOf {
    Entity group [0..*] {
      Snd_min group [0..*] {
        Snd_min_ack group [0..*] {
          Snd_prompt_nak
        }
      },
      Recv_min group [0..*] {
        Recv_min_ack group [0..1] {
          Recv_immediate_nak,
          Recv_deferred_nak,
          Recv_prompt_nak,
          Recv_asynch_nak
        }
      }
    },
    Channel group [0..*] {
      Reliable
    }
  }
}
```

Note that the FD also models the reliability of the communication channel. Even though this is strictly speaking a property of the environment, not the system, it is useful to capture it as as a feature. The truth of any property is then automatically expressed in function of the reliability of the communication channel. It is also worth mentioning that the features corresponding to the four `NAK` modes are mutually exclusive. Consequently, the FD has 56 products.

The fPromela model of the CFDP represents the scenario in which a file is transmitted between two CFDP entities. With this model we verify under

which condition (i.e., with which features) the file will be successfully transmitted to the receiving entity. The model is based on the CFDP specification [Consultative Committee for Space Data Systems (CCSDS), 2007], rather than code developed by Spacebel. Because we are only interested in the transmission procedure, the CFDP operations that are unrelated to the transmission itself (user requests, checksum errors, ...) are ignored. Moreover, we applied some simplifications to the transmission procedure as described in the protocol specification. The model is due to Maxime Cordy, a student who also collaborated on the development of SNIP.

For the benchmarks, we considered a deadlock check, property (#1), and the following fLTL properties.

(#2) The whole file is eventually received, $\Diamond fileReceived$. This property is violated by 38 products, all those where the communication channel is not reliable, and those without the sending or without the receiving feature.

(#3) If the `EOF` message eventually reaches the receiver, the whole file is eventually received, $\Diamond eofReceived \Rightarrow \Diamond fileReceived$. This property is violated 18 products: all those where the channel is not reliable and with both a sender and a receiver (otherwise the assumption would not hold).

(#4) The same as (#3) with the additional assumption that a negative acknowledgement (`NAK`) is eventually reaches the sending entity:

$$(\Diamond eofReceived \wedge \Diamond nakReceived) \Rightarrow \Diamond fileReceived.$$

This property is violated by 9 products, those with an unreliable channel and where either

- the receiver is in asynchronous, immediate, or prompted `NAK` mode,
- or the receiver is in deferred `NAK` mode but the sender is unable to answer to the `NAK` messages.

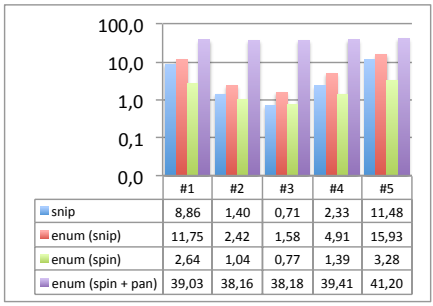
(#5) A variation of the previous property where the second assumption is that the sender receives `NAK` messages infinitely often.

$$(\Diamond eofReceived \wedge \Box\Diamond nakReceived) \Rightarrow \Diamond fileReceived.$$

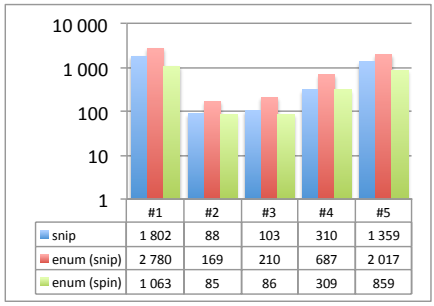
This property is violated by 4 products: those where the communication channel is unreliable, the receiver has enabled a `NAK` mode and the sender is unable to answer to `NAK` messages.

The results for the previously mentioned properties are shown in Figure 9.9. In terms of runtime, the FTS algorithm in SNIP is between 1.33 and 2.23 times faster than the naïve algorithm implemented with SNIP.

In terms of state space reduction, the FTS algorithm in SNIP reduced the average number of states from 1 440 675 to 910 497 (37%), whereas SPIN reduced it to 579 077 (60%). The small reduction in the state space explains the



(a) Runtime in seconds, logarithmic scale.



(b) Number of transitions in thousands, logarithmic scale.

Figure 9.9: Benchmark results for the CFDP.

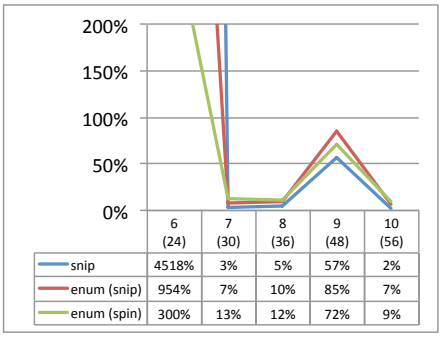
less good performance of the FTS algorithm in this case. Indeed, we observe that the greater this reduction, the larger the difference in runtime of the two algorithms. The significant reduction in the case of SPIN is most likely due to optimisations such as partial order reduction. The model in question is a distributed system in which partial order reductions can lead to significant reductions in the size of the state space.

As for the elevator, the deadlock check required the most memory: 336.9 MB for the naïve algorithm and 395.4 MB for the FTS algorithm. These sizes correspond to 933 276 states in the transition system explored by the naïve algorithm and 1 069 840 states in the FTS. Here, the FTS algorithm requires 17% more memory, for a 14% increase in the number of states. Again, there are 3% overhead for the BDDs representing sets of products.

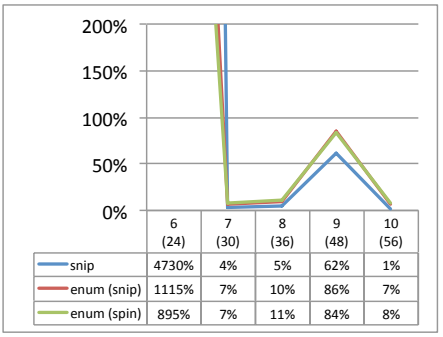
9.4.5 Incremental benchmarks

While the previous experiments compared the FTS algorithm and the naïve algorithm on a fixed number of products, we also conducted an experiment to evaluate how each algorithm behaves when the number of products increases. For this we used the CFDP model of the previous section and property (#1), the deadlock check. We first verified the model restricted to 18 products, with the following five features: *Snd_min*, *Snd_min_ack*, *Recv_min*, *Recv_min_ack*, and *Reliable*. We then reverified the model five times, each time adding one feature in the following order: *Recv_immediate_nak* (24 products), *Recv_deferred_nak* (30 products), *Recv_asynch_nak* (36 products), *Snd_prompt_nak* (48 products), and *Recv_prompt_nak* (56 products).

Figure 9.10(a) shows the increase in runtime (in percent) for each added feature. The x-axis shows the number of features and the number of products in parentheses. More formally, the shown value is $\Delta n = \frac{runtime(n) - runtime(n-1)}{runtime(n-1)}$ where $n \in [6, 10]$ is the number of considered features and $runtime(n)$ the



(a) Rate of growth of the runtime.



(b) Rate of growth of the number of transitions.

Figure 9.10: Results of the incremental benchmarks on the CFDP model.

runtime to verify the model with n features. Figure 9.10(b) similarly shows the increase in the number of fired transitions. Both figures are rather similar, indicating the correlation between runtime and transitions fired.

For both measures, the increase when adding the sixth feature, *Recv_immediate_nak*, is huge and off the charts. The reason for this increase is that in the *immediate* mode, the receiver sends a `NAK` as soon as a loss is noticed. This `NAK` itself can get lost, which leads to a large number of combinations for the lost/received data segments and lost/received `NAK` messages. Without the *Recv_immediate_nak* feature, the FTS has 16 801 states and the transition systems of all products combined have 98 112, i.e., the FTS is 78% smaller. When the feature is added, the FTS grows to 917 066 states (by 5300%) and the transition systems to 1 192 023 states (by 1114%). Now the FTS is only 15% smaller. Basically, the states added by the *Recv_immediate_nak* feature have a negative impact on the compactness of the FTS. This accounts for the large increase in the state space explored by the FTS algorithm and shown in Figure 9.10(b). The increase in the runtime shown in Figure 9.10(a) is a consequence of this.

For the other features, the increase for the FTS algorithm is consistently lower than for the naïve algorithm. For example, when the number of features increases from eight to nine, the number of products increases from 36 to 48 and the runtime of the FTS algorithm grows only by 57%, while the runtime of the naïve algorithm rises by 85% when implemented with SNIP and by 72% when implemented SPIN. We conducted these incremental benchmarks also for the other properties and other orders of features, observing similar results. As the number of features increases, the runtime for the FTS algorithm grows slower than that of the naïve algorithm. This indicates that the FTS algorithm scales better with the number of features than the naïve algorithm.

Another way to analyse the data collected during the incremental bench-

marks is to compare the rate at which the runtime increases to the rate at which the number of products increases. For the algorithm to scale with the number of features, its runtime should increase linearly in the number of features (i.e., logarithmically in the number of products), rather than exponentially. To test this, we take the runtime for checking the model consisting of 24 products as the baseline, and extrapolate the runtime for the models of 30, 36, 48 and 56 products in two ways.

Exponential. The runtime increases at the same rate as the number of products, i.e., by 25% for the increase to 30, by 50% for the increase to 36 and so on. This growth is exponential in the number of features.

Linear. The runtime increases at the same rate as log_2 of the number of products, i.e., by 7% for the increase to 30, by 13% for the increase to 36 and so on. This growth is linear in the number of features.

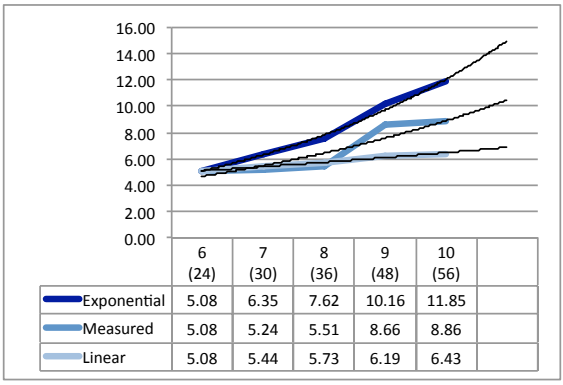
As the baseline we chose the runtime for the model of 24 products, because the runtime for the model of 18 products (i.e., without the *Recv_immediate_nak* feature) is an outlier for all three algorithms.

The result is shown in Figure 9.11, where we plot the projected runtimes as well as the measured runtime for each algorithm and implementation. To make the results easier to interpret, we further added a function that approximates each line (in black). As can be seen clearly in these figures, the runtime of the FTS algorithm increases at a rate that is between exponential and linear, whereas that of the naïve algorithms increases at the exponential rate (whether implemented with SNIP or with SPIN).

9.4.6 Discussion

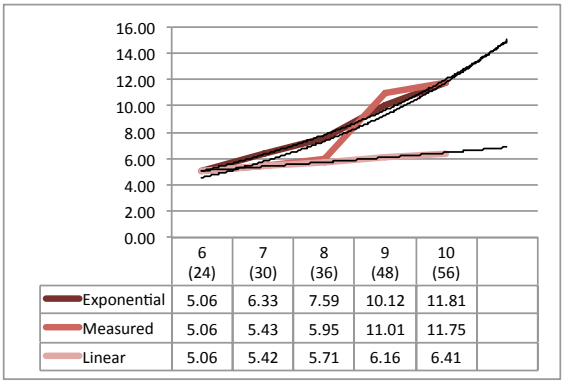
These results show that the FTS algorithm is a viable approach for state space reduction, in some cases leading to better performance than the script using SPIN, a tool that has been under development for 20 years. There is, nevertheless, room for improvement of the implementation. Furthermore, we believe that the state space reductions of SNIP and SPIN can reinforce each other, opening an exciting area of future work. One step will be to extend the optimisations currently implemented in SPIN, such as partial order reduction, to FTS. The FTS algorithm could then be integrated into SPIN. While such a project will most likely be more expensive than the development of SNIP, its prospects are promising. Moreover, the incremental benchmarks show that while the FTS algorithm is an improvement over the naïve algorithm, its runtime is not linear in the number of features. This would have to be the case for the algorithm to scale with the number of features.

As for the threats to validity, all experiments were executed by an automated script running on a dedicated machine. Moreover, each reported value is the average of three to five measures. This minimises the risks of flawed runtime measures, e.g., due to other processes. Furthermore, the runtime reported for the naïve algorithm is only the verification time. This removes the bias that



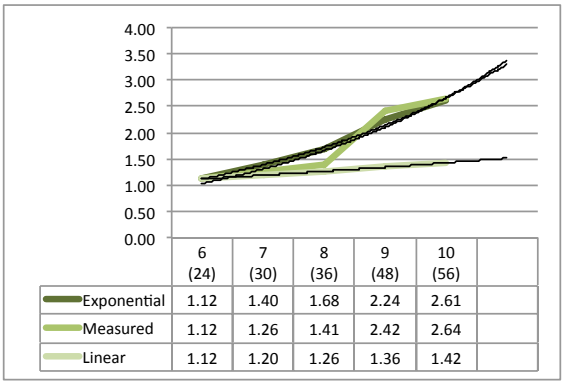
	6 (24)	7 (30)	8 (36)	9 (48)	10 (56)	
Exponential	5.08	6.35	7.62	10.16	11.85	
Measured	5.08	5.24	5.51	8.66	8.86	
Linear	5.08	5.44	5.73	6.19	6.43	

(a) FTS algorithm with SNIP.



	6 (24)	7 (30)	8 (36)	9 (48)	10 (56)	
Exponential	5.06	6.33	7.59	10.12	11.81	
Measured	5.06	5.43	5.95	11.01	11.75	
Linear	5.06	5.42	5.71	6.16	6.41	

(b) Naïve algorithm with SNIP.



	6 (24)	7 (30)	8 (36)	9 (48)	10 (56)	
Exponential	1.12	1.40	1.68	2.24	2.61	
Measured	1.12	1.26	1.41	2.42	2.64	
Linear	1.12	1.20	1.26	1.36	1.42	

(c) Naïve algorithm with SPIN.

Figure 9.11: Growth of the runtime with increasing number of features.

might have been caused by an inefficient implementation of the script iterating through the products, generating the input files, deleting them, and so forth.

As to conclusion validity, from the obtained results it is clear that the FTS algorithm is in most cases an improvement over the naïve algorithm, both in terms of runtime and in terms of state space. The use of only three models means that this is a quasi-experiment and that the extent to which it generalises cannot be concluded from its results. Nevertheless, we used a large number of properties, including both liveness and safety (mostly combinations of both) and covering properties satisfied under various circumstances (i.e., violated by different sets of products). For the incremental benchmarks we only considered a single model. It is thus also a quasi-experiment whose results cannot necessarily be generalised. In addition to the model, the outcome can be be influenced by the baseline time and the order in which the remaining features are added to the model. The baseline was chosen such that the remaining features did not increase the size of the model by more than 100%. For the four added features, we executed the benchmarks in several orders, with similar results: the increase of the FTS algorithm is between exponential and linear, whereas the naïve algorithm increases exponentially.

9.5 Conclusion

We presented SNIP, a model checker that implements the semi-symbolic algorithms of Chapter 6. Its modelling language, fPromela, is an extension of Promela, the language of the popular model checker SPIN. fPromela uses annotation rather than composition to model variability. Annotations in fPromela are similar to `#ifdef`s in the C programming language. We showed that fPromela and FTS are expressively equivalent languages.

SNIP was implemented from scratch. Although this was a time consuming and risky undertaking, it has given us many insights into the use of the semi-symbolic FTS algorithm as part of a model checker for a non-trivial language. Experiments conducted with SNIP have shown that the FTS algorithm is generally faster than the naïve algorithm, and a viable approach for state space reduction. In the case of the CFDP, we observed that the runtime of the naïve algorithm increases exponentially with the number of features, unlike the FTS algorithm, whose increase is between linear and exponential.

Part IV

Final Remarks

Chapter 10

Review and Perspectives

> *It is common sense to take a method and try it. If it fails, admit it frankly and try another. But above all, try something.*
>
> Franklin D. Roosevelt, *Looking Forward*, 1933

The starting point for this work was the question: *How can model checking be accomplished in the presence of variability?* We refined this vast question into the three research questions addressed in this thesis. These questions cover the modelling language, the model checking algorithms, and the practicality of both of them. Our answers to these questions were exhibited over the preceding two parts of the thesis. In this last part, we conclude the thesis with a review of the results, and the challenges ahead.

In Section 10.1, we review our results and relate them to the research questions. This leads us to identify limitations of our work in Section 10.2 as well as perspectives for future work in Section 10.3.

10.1 Answering the research questions

The motivation and initial driver of this thesis is the study of model checking in the presence of variability. Part I concluded with a review of the state of the art, in which we noted that this topic is largely untreated in the research literature and that the few approaches targeting the problem have considerable limitations. Most approaches do not consider features as first-class entities in their models. Moreover, there is but one approach that specifically targets model checking [Lauenroth et al., 2009], with rather inefficient algorithms.

As the question touches upon many topics, we decomposed it into three research questions. Let us now look back at these questions and formulate answers. The first research question is the most fundamental.

RQ1 *How can the behaviour of an SPL be described formally, and what does model checking of SPLs mean?*

In Chapter 4, we showed that by annotating transitions of a transition system with feature expressions, we can model the behaviour of every product as a transition system. If moreover, the features on the feature expressions are linked to an FD, we model the behaviour of all valid products, which is essentially that of the SPL. Our FTS formalism allows to do exactly this. Contrary to existing formalisms, an execution of an FTS can be linked to one or more products. This makes it possible to identify the products that exhibit certain behavioural properties. Furthermore, features in FTS can be non-monotonic, i.e., features can add or remove transitions (thereby adding or removing behaviours). This way, FTS can be exponentially more succinct than transition systems. FTS are thus our answer to the first part of this research question.

As to the second part, there are several ways in which model checking of SPLs can be understood. Since there have only been a few attempts to adapt model checking techniques to SPLs, the most common view is that SPL model checking corresponds to model checking of individual products. Analyses that take variability into account (*variability-aware* analyses) have only recently started to attract attention in the SPL and FOSD community [Apel et al., 2011]. Our discussion of SPL model checking in Chapter 5 adheres to this latter view: model checking an FTS corresponds to model checking the behaviour of all the products in the SPL. This also meets our goal of applying model checking during domain engineering, rather than application engineering. Since a model check covers several products, a failed check needs to identify the problematic products. In consequence, the MC model checking problem is defined as returning at least one violating product, and the EXTMC problem as returning the full set of violating products. Properties for an SPL can be specified in existing temporal logics, LTL and CTL. We adapted these logics slightly: fLTL and fCTL allow the property to specify the products over which it should hold.

Of course, these new decision problems bring with them algorithmic challenges, the subject of our second research question.

RQ2 *Is SPL model checking tractable? If so, how?*

In Chapters 6 and 7, we studied the complexity of the model checking problems and gave algorithms for model checking SPLs modelled as FTS. The SPL model checking problems are harder than the corresponding problems for transition systems due to (i) the conciseness of the FTS and (ii) the fact that its behaviour is defined in terms of the products allowed by the FD, which means that FTS analyses have to solve at least one SAT problem. Hence, the REACHABILITY problem which is NL-Complete for transition systems becomes NP-Complete for FTS. Both MC_{fLTL} and EXTMC_{fLTL} are FPSPACE-Complete for FTS, which is the same as for transition systems due to NPSPACE = PSPACE [Savitch, 1970]. MC_{fCTL} is FNP-Complete and EXTMC_{fCTL} is #P-Complete whereas CTL model checking for transition systems is P-Complete.

In terms of algorithms, the features in an FTS lead to an exponential increase in the algorithmic complexity compared to model checking algorithms for

transition systems. Since FTS are exponentially more succinct than transition systems, this exponential factor cannot be avoided. Nevertheless, experiments have shown that our algorithms achieve considerable speedups and have practical advantages over the naïve procedure of using classical model checking on every product. However, our experiments also showed that our algorithms are not exponentially faster than the naïve procedure. This should have been the case for our algorithms to scale linearly in the number of features.

The answer to the second research question is thus that the tractability of SPL model checking depends mainly on the number of features in the behavioural model. Moreover, in Chapter 7, we have shown that features can be considered to be a special kind of system variable. In practice, the exponential factor due to the features is likely to be dwarfed by the size of the state space. According to this view, SPL model checking is thus not inherently harder than classical model checking.

RQ3 *How can SPL model checking be applied in practice?*

As shown in the introduction to Part III, the SPL model checking use case is different from that of single systems model checking. Inputs and outputs of a model checker have to be specified in terms of features. Furthermore, the model checker has to support the quantifiers from fLTL and fCTL. Nevertheless, the first implementation of our theory as part of a state-of-the-art model checker, NuSMV, was relatively straightforward. In Chapter 7, we already showed that the symbolic FTS model checking algorithm can be largely reduced to the one for transition systems. Most of the implementation effort was thus spent on the modelling language, fSMV. The semi-symbolic FTS algorithms, in contrast, could not easily be implemented as part of an existing model checker, due to the many changes that would be required throughout the code. Nevertheless, we were able to implement an on-the-fly version of this algorithm as part of a model checker for a non-trivial language, fPromela.

Although we did not study the use of SPL model checking techniques in realistic industrial settings, we managed to show that implementations based on tools or languages that are commonly used in industrial settings are possible. NuSMV is used in industrial settings [Chiappini et al., 2010], and our extension thereof is rather conservative. Furthermore SNIP's modelling language, fPromela, combines Promela, a popular specification language, with annotation-based variability, a common way to specify variability in industrial product lines. These observations combined with the fact that SPL model checking as seen in this thesis is the intuitive extension of classical model checking to sets of systems, means that we do not see obstacles to the practical applicability of SPL model checking other than those that already exist for single system model checking. One the contrary, the return on investment of model checking is likely to be higher for SPLs than single systems because bugs (and their corrections) affect several (possibly many) systems, rather than one.

10.2 Limitations

To address the challenge of efficient reasoning noted in the introduction, the FTS model checking algorithms should have been linear in the number of features. However, all our algorithms are exponential in the number of features. Moreover, we did not observe exponential speedups over the naïve algorithms in our experiments. Concretely, this means that the algorithms proposed herein will not scale with the number of features. This can be seen as a limitation.

However, this limitation corresponds to a theoretical limit. By Theorem 4.16, a transition system with the same behaviours as an FTS is exponentially larger than the FTS. Since the complexity of our model checking algorithms is that of those for transition systems multiplied by the same exponential factor, it is impossible for a model checking algorithm to avoid it. The only way to overcome this would be to restrict the algorithm to subsets of the FTS language, or accept unsound or incomplete algorithms. Such approaches were judged to be out of scope for this thesis, and will be part of our future work.

Another limitation of the work presented in this thesis is its validation. We just considered four systems for our experiments, and all but one of them were modelled by ourselves. In addition, the results obtained in the three experiments conducted with SNIP had different factors for speedup and state space reduction. As a consequence of this (already stated in the respective sections), the power of these experiments to predict algorithmic performance on other models is rather low. However, we believe that these experiments are sufficient for this thesis, which is mostly concerned with foundational results. The goal of the implementations was not only to conduct experiments, but to show the viability and practicality of the proposed theories, which was demonstrated independently of the experiments.

10.3 Perspectives

The are several avenues for future research.

A first direction, the most promising and important in our view, is to investigate methods for **compositional reasoning**. In [Fisler and Krishnamurthi, 2001, Li et al., 2002b], the authors propose a compositional approach for CTL model checking, which is linear in the number of features. However, the approach is rather restrictive as to the kind of models that can be analysed (e.g., monotonic features only). A first step towards compositional reasoning for FTS would be to adapt these algorithms to CTL and LTL model checking of FTS. In the long term, we believe that it is possible to identify necessary and sufficient conditions on the changes made by a feature in an FTS for it to qualify for compositional reasoning. Since it is not likely that all features will be analysable compositionally, we have to investigate algorithms that combine compositional and non-compositional approaches, such as those described in Chapters 6 and 7.

Another direction for future work is closely linked to the previous one: to investigate modelling languages that **combine the compositional and annotative paradigms** from FOSD. While developing the models used in our benchmarks, we noticed the advantages and disadvantages of each paradigm. Annotations are good for cross-cutting features, which are hard to modularise or hard to understand once modularised. Annotations are also excellent to manage detailed feature interactions, i.e., cases in which a feature operates differently when another feature is present. Compositional approaches, on the other hand, have the advantage of making a first step towards compositional reasoning (as it is much easier to judge the impact a feature if it is specified as a change, rather than if it is spread over the code). They also lend themselves well to modelling features which are independent of other features. In the ideal case, a modelling language should thus offer both mechanisms. A related idea for future work would be to develop model transformations that can convert compositional style models into annotative models and the other way round, as well as methods to check that two such models are equivalent (e.g., bisimulation equivalent). A transformation from fPromela to fSMV, for instance, would allow us to compare the performance of the semi-symbolic and fully symbolic algorithms more closely.

Another general direction for future work is to investigate how methods currently used in classical model checking can be used for SPL model checking. As mentioned in Section 9.4, an optimisation technique that might be beneficial for FTS is **partial order reduction**. This technique reduces the number of states to explore by identifying redundant interleavings of parallel processes. Furthermore, a number of state space reduction techniques are based on **abstraction and refinement**. They require notions such as simulation and bisimulation, which could be defined and checked on the level of FTS. A related approach is three-valued model checking of partial models, by extending FTS with allowed/required modalities as discussed in Section 4.4.3.

In addition to investigating optimisation techniques, another direction for future work would be to extend the expressiveness of FTS. In its current state, the FTS semantics is defined for individual products. It does not cover the case where a product changes at runtime, i.e., during the execution of the model, by **activating or deactivating features**. This is the case for self-adaptive systems or dynamic product lines [Cheng et al., 2009, Classen et al., 2008b]. Moreover, an FTS always references a single FD. A possible extension would be to make **FDs first-class entities** in FTS, so that more then one FD can be referenced in the model, and that an FD can be instantiated multiple times. In the case of the CFDP discussed in Section 9.4.4, for instance, we used the same FD for both the sender and the receiver. Conceptually, however, sender and receiver are separate entities, and the model should refer to two instances of the CFDP FD, one for the sender and one for the receiver.

As we have shown in this thesis, variability is largely an orthogonal concern to existing concerns in model checking. For example, most SPLs in the automotive domain have **real-time** constraints. To be able to handle such

cases, FTS could be extended with a notion of time, as done already by several methods in classical model checking of single systems. Similarly, FTS could be extended with **probabilities** to allow for modelling of uncertainty. The algorithmic principles proposed herein could also be applied to software model checking, e.g., in JavaPathfinder [Visser et al., 2000]. A related avenue for future work is to abstract FTS or fPromela models from the source code of an SPL, e.g., C code with `#ifdef`s, similar to the way Promela is extracted from Java code in Bandera [Corbett et al., 2000].

There are thus many avenues for future research. In this context, the candidate has contributed to the writing of three research projects: Tournesol, an accepted Belgium/France collaboration project; an accepted FNRS PhD Project; and CLEVER, an EU FP7 STREP that is currently in preparation. All these projects target the extension of the results obtained in this thesis.

Conclusion

> *Many have imagined republics and principalities which have never been seen or known to exist in reality; for how we live is so far removed from how we ought to live, that he who abandons what is done for what ought to be done, will rather bring about his own ruin than his preservation.*
>
> Niccolò Machiavelli, *The Prince*, 1513

In this thesis, we sought to answer the question *How can model checking be accomplished in the presence of variability?* The question is rather vast, and it was indeed our goal to give a complete treatment of the problem, from fundamental to practical issues. The question can be divided into three parts.

The most fundamental one is to provide a formal representation for the behaviour of an SPL. For this, we proposed *Featured Transition Systems* (FTS), a semantic model for SPL behaviour. FTS are transition systems in which transitions are linked to the features of an SPL by the means of feature expressions. From the FTS, one can obtain the behaviour of any product as a transition system. An FTS is thus a concise model for the behaviour of the entire SPL. An FTS comes equipped with a feature diagram which expresses the set of valid products. This allows for separation of concerns as variability and behaviour are specified independently. Furthermore, the feature diagram allows the FTS to capture information about products that are known to be problematic. We studied three mechanisms for expressing variability in FTS, labelling transitions with feature expressions, labelling transitions with single features linked to a feature diagram, and labelling transitions with single features and the ability to specify priorities between transitions. We showed that all of these types of FTS are exponentially more succinct than transition systems.

SPL model checking, as seen in this thesis, is as a direct generalisation of classical model checking of single systems. Our goal was to apply model checking to SPLs with as few changes as possible to the concepts and notations. The result is a model checking approach that, at least to its end-user, is very similar to single system model checking. In consequence, she can apply her knowledge of existing model checking approaches to SPL model checking, she can interpret model checking results similar to the way she interpreted results in classical model checking, and she can potentially reuse or extend existing

models. This philosophy is reflected in many elements of our theory. For example, the semantics of FTS is defined in terms of transition systems. Features themselves are not part of the behaviour, but rather changes to the behaviour.

This philosophy is also reflected in our definition of the model checking problems. Our view is that properties for an SPL should be specified in existing temporal logics, LTL and CTL. In this thesis, we proposed slight extensions to these logics: *feature LTL* and *feature CTL*. Properties in these logics are a combination of an LTL (resp. CTL) property with a quantifier that expresses over which products the property should hold. Intuitively, model checking an FTS against such a property is equivalent to model checking all products covered by the quantifier, or all products of the product line if there is no quantifier. More precisely, we identified and formalised two model checking problems for SPLs, MC and EXTMC. Both return *true* if all products satisfy the property. In case the property is violated, MC returns at least one violating product, whereas EXTMC returns the full set of violating and the full set of satisfying products. The second model checking problem, EXTMC is fundamentally different from classical model checking in single systems in that it produces information about several instances of the model. It can be seen as a form of generalised model checking, in which we compute the values of model parameters (the features) for which the property holds.

New model checking algorithms are required to compute these decision problems. The second part of the answer to the initial question is thus concerned with algorithms and their efficiency. Model checking algorithms perform a search in the state space of the system. The largest impact on the runtime of these algorithms is the size of the state space, which is exponential in the number of system variables and processes, a problem known as *state explosion*. In model checking, one commonly distinguishes two types of algorithm, explicit and symbolic. In an explicit algorithm, the search in the state space visits system states one by one. In a symbolic algorithm, in contrast, symbolic data structures are used to represent sets of states. Computations are performed over these data structures, which means that the algorithm manipulates sets of states, rather than single states. Symbolic algorithms can, to some extent, address the state explosion problem. In SPLs, this problem is exacerbated by the fact that the number of products is potentially huge; exponential, in fact. Given n features, the number of products to be checked is $O(2^n)$.

For SPL model checking, we proposed a new type of algorithm, which we called *semi-symbolic*. This algorithm represents the state space explicitly, and keeps information about products in a symbolic data structure. Its goal is to reduce the state space by exploiting similarities between the different products of the SPL. This algorithm targets the fLTL model checking problems, MC_{fLTL} and EXTMC_{fLTL}. It follows the well-established approach of automata-based model checking [Vardi and Wolper, 1986]. Two suitable symbolic data structures for sets of products were presented: an encoding based on recording required and excluded features, and an encoding based on Boolean functions. For

the fCTL model checking problems, MC_{fCTL} and EXTMC_{fCTL}, we proposed a fully symbolic algorithm. We showed how this algorithm can be reduced to classical symbolic model checking of specially crafted transition systems. The symbolic algorithm puts the semi-symbolic algorithm into perspective, highlighting its position between fully explicit and symbolic model checking algorithms. In addition to these *FTS algorithms*, we also provided a fully explicit algorithm for the same decision problems, the *naïve algorithm*. It consists in model checking each product separately and serves as a baseline in the experiments we conducted.

We carried out a thorough study of the computational complexity of the decision problems and the algorithms solving them. We showed that problems that are computationally easy for transition systems are hard for FTS. The REACHABILITY problem, for instance, is NL-Complete for transition systems and NP-Complete for FTS. Both MC_{fLTL} and EXTMC_{fLTL} are FPSPACE-Complete for FTS, compared to PSPACE-Complete for classical model checking of transition systems. The similar complexity is due to NPSPACE = PSPACE [Savitch, 1970]. The model checking problems for fCTL are harder than the corresponding problem for transition systems. MC_{fCTL} is FNP-Complete and EXTMC_{fCTL} is #P-Complete whereas CTL model checking for transition systems is P-Complete. As to the algorithms, all our FTS model checking algorithms have the complexity of the corresponding problem for single systems, multiplied by an exponential factor $O(2^n)$. The FTS algorithms are thus not of lower computational complexity than the naïve algorithms. With experiments we show that in practice, our FTS algorithms are generally more efficient than the naïve algorithm and have other practical advantages.

This leads us to the third part of the answer to the initial question. While the first two parts were concerned with theory, the last part provides an assessment of the practical applicability of these theories. Concretely, we put our theoretical results into practice by implementing them as part of model checking tools. The implementation of these tools provides insights into the applicability of the theories, whereas the tools themselves allow us to evaluate the efficiency of our algorithms through experiments. Since the use case of SPL model checking differs from the one in single systems, we had to re-implement and change rather than reuse existing model checkers. The principal difference between SPL model checking and single systems model checking is *variability*, expressed in terms of features. Our model checking tools recognise them as a first-class concept; in the modelling language, as well as in the presentation of model checking results. In this regard, our model checkers are currently unique. All tools are available (open source) at the FTS website [Classen, 2010b].

The fully symbolic FTS algorithms were implemented with the fNuSMV toolset. The modelling language used by this toolset is fSMV, a feature-oriented extension of the SMV language proposed in [Plath and Ryan, 2001]. It is based on superimposition: features are specified individually as changes to a base system. Features are composed to create the model of a product. We proposed

a new form of composition. It creates a symbolic FTS, to which our symbolic model checking algorithm can be applied. Since the fully symbolic model checking algorithm for FTS can largely be reduced to the classical algorithm for transition systems, we implemented it by extending the state-of-the-art symbolic model checker NuSMV [Cimatti et al., 2000]. We showed that fSMV and FTS are expressively equivalent modelling languages. The fNuSMV toolset allowed us to conduct experiments showing that the FTS algorithms are orders-of-magnitude faster than the naïve algorithm. NuSMV uses BDDs to encode sets of states symbolically. The ordering of variables can have a significant impact on their size, and hence on the runtime of the algorithm. Our algorithm requires one variable ordering for all products, whereas the naïve algorithm requires $O(2^n)$ orderings. The improvement in speedup can thus be attributed to the fact that the variable ordering used by the FTS algorithm can be optimised.

While the fully symbolic FTS algorithm can largely be reduced to the one for transition systems, this is not the case for the semi-symbolic algorithm. The model checker SNIP, which uses the semi-symbolic FTS algorithms, was thus implemented from scratch. SNIP's modelling language is fPromela, an extension of Promela, the language of the popular model checker SPIN [Holzmann, 2004]. Variability in fPromela is modelled by guarding statements with feature expressions. Guards in fPromela are similar to `#ifdef`s in the C programming language and other annotation-based techniques widely used in practice. Moreover, behaviour in fPromela is specified in a procedural style. We showed that fPromela and FTS are expressively equivalent languages. Furthermore, SNIP has built-in support for feature diagrams, using our TVL modelling language [Classen et al., 2011a]. Experiments conducted with SNIP have shown that the FTS algorithm is generally faster than the naïve algorithm, and a viable approach for state space reduction.

We have shown that model checking can be accomplished in the presence of variability. Our proposal is based on FTS, a semantic model for SPL behaviour, and implemented in two model checking tools, fNuSMV and SNIP. The contributions made in this thesis do not only apply to SPLs, or variability-intensive systems. As noted before, the feature expression returned by our algorithm reveals the problematic features, which generally helps to identify the fragment of the model which is responsible. This means that even for a system without variability, our algorithm can be used to determine whether a piece of code, or a function, serves its intended purpose. If this purpose can be expressed by a temporal property and the piece of code is annotated by a feature, our algorithm can be used to determine whether it does indeed play a role in the satisfaction of this property. Moreover, the idea of an algorithm combining symbolic execution with explicit state-space exploration does not have to be restricted to features and Boolean variables. As discussed before, the algorithm can be seen as a variation of generalised model checking. The model is parameterised in some way (Boolean parameters in our case), and the problem consists in calculating a Boolean function over the parameters which

characterises the parameter values for which the model violates a property. The ExtMc problem is then an optimisation problem, which consists in finding the function which characterises violating and satisfying parameter values. Seen this way, it becomes clear that the parameters do not have to be features of a variability-intensive system.

We identified two principal challenges for model checking in the presence of variability: scaleable modelling and efficient reasoning. Both are due to the complexity created by the huge number of potential products. We showed that FTS are exponentially more succinct than transition systems, which means that they do scale with the number of features, thereby solving the scaleability challenge. To solve the challenge of efficient reasoning, the model checking algorithms would have to be exponentially more efficient than classical algorithms executed over every product (i.e., the naïve algorithm). However, the FTS model checking algorithms are of the same algorithmic complexity as the naïve algorithms. While our experiments showed that they can achieve order-of-magnitude improvements over the naïve algorithm, these improvements are not exponential. The analysis of the problem complexity has shown that the SPL model checking problems are indeed harder. Significant improvements in algorithmic performance, not least algorithms that are linear in the number of features, can only be achieved at the expense of the modelling language, that is, by restricting algorithms to models that are less powerful than FTS. Such algorithms could be combined with our FTS algorithms to yield an algorithm for FTS in general. This is one of the many avenues for future work. Among the contributions of this thesis, the most important is the description and analysis of the SPL model checking problem itself. Our goal is that FTS become to SPL model checking what transition systems are to single systems model checking, the semantic model upon which theories are built. We hope that we managed to prepare the ground for future research into SPL model checking.

Bibliography

[Abele et al., 2010] Abele, A., Papadopoulos, Y., Servat, D., Törngren, M., and Weber, M. (2010). The CVM framework – a prototype tool for compositional variability management. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 101–106. University of Duisburg-Essen.

[Abran et al., 2004] Abran, A., Moore, J. W., Bourque, P., and Dupuis, R., editors (2004). *Guide to the Software Engineering Body of Knowledge (SWEBOK)*. IEEE Computer Society.

[Acher et al., 2009] Acher, M., Collet, P., Lahire, P., and France, R. (2009). Composing feature models. In *Software Language Engineering, Second International Conference, SLE 2009, Denver, CO, USA, October 5-6, 2009, Revised Selected Papers*, volume 5969 of *LNCS*, pages 62–81. Springer.

[Alpern and Schneider, 1987] Alpern, B. and Schneider, F. B. (1987). Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126.

[Alrajeh et al., 2009] Alrajeh, D., Kramer, J., Russo, A., and Uchitel, S. (2009). Learning operational requirements from goal models. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 265–275.

[Alur and Dill, 1990] Alur, R. and Dill, D. (1990). Automata for modeling real-time systems. In *17th International Colloquium on Automata, Languages and Programming (ICALP)*, volume 443 of *LNCS*, pages 322–335. Springer.

[Apel et al., 2011] Apel, S., Cook, W., Czarnecki, K., and Nierstrasz, O. (2011). Feature-Oriented Software Development (FOSD) (Dagstuhl Seminar 11021). *Dagstuhl Reports*, 1(1):27–41.

[Apel et al., 2009] Apel, S., Kästner, C., and Lengauer, C. (2009). Featurehouse: Language-independent, automated software composition. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 221–231. IEEE.

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

[Apel et al., 2005] Apel, S., Leich, T., Rosenmüller, M., and Saake, G. (2005). Featurec++: On the symbiosis of feature-oriented and aspect-oriented programming. In *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, pages 125–140.

[Asirelli et al., 2010a] Asirelli, P., ter Beek, M. H., Fantechi, A., and Gnesi, S. (2010a). A logical framework to deal with variability. In *Proceedings of the 8th international conference on Integrated formal methods (IFM)*, number 6396 in LNCS, pages 43–58. Springer.

[Asirelli et al., 2009] Asirelli, P., ter Beek, M. H., Gnesi, S., and Fantechi, A. (2009). Deontic logics for modeling behavioural variability. In *Proceedings of the Third International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09), Sevilla, Spain, January 28-30*, pages 71–76.

[Asirelli et al., 2010b] Asirelli, P., ter Beek, M. H., Gnesi, S., and Fantechi, A. (2010b). A deontic logical framework for modelling product families. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 36–44.

[Baier et al., 2009] Baier, C., Haverkort, B., Hermanns, H., and Katoen, J.-P. (2009). Model-checking algorithms for continuous-time markov chains. *IEEE Trans. Softw. Eng.*, 29(7):524–541.

[Baier and Katoen, 2008] Baier, C. and Katoen, J.-P. (2008). *Principles of Model Checking*. MIT Press.

[Ball et al., 2004] Ball, T., Cook, B., Levin, V., and Rajamani, S. K. (2004). SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In *Integrated Formal Methods, 4th International Conference, IFM 2004, Canterbury, UK, April 4-7, 2004, Proceedings*, volume 2999 of *LNCS*, pages 1–20. Springer.

[Ball and Rajamani, 2001] Ball, T. and Rajamani, S. (2001). The SLAM toolkit. In *13th Computer Aided Verification*, number 2102 in LNCS, pages 260–264. Springer.

[Batory et al., 2006] Batory, D., Benavides, D., and Ruiz-Cortes, A. (2006). Automated analysis of feature models: Challenges ahead. *Communications of the ACM*, 49(12):45–47.

[Batory, 2004] Batory, D. S. (2004). Feature-oriented programming and the ahead tool suite. In *26th International Conference on Software Engineering (ICSE'04)*, pages 702–703, Edinburgh, United Kingdom.

[Batory, 2005] Batory, D. S. (2005). Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the 9th Int. Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 7–20. Springer.

[Beer et al., 2001] Beer, I., Ben-David, S., Eisner, C., and Rodeh, Y. (2001). Efficient detection of vacuity in temporal model checking. *Formal Methods in System Design*, 18(2):141–162.

[Benavides et al., 2005] Benavides, D., Martín-Arroyo, P. T., and Cortés, A. R. (2005). Automated reasoning on feature models. In *Advanced Information Systems Engineering, 17th International Conference, CAiSE 2005*, pages 491–503, Porto, Portugal.

[Benavides et al., 2010] Benavides, D., Segura, S., and Ruiz-Cortés, A. (2010). Automated analysis of feature models 20 years later: a literature review. *Information Systems*, 35(6):615–636.

[Biere et al., 1999] Biere, A., Cimatti, A., Clarke, E., and Zhu, Y. (1999). Symbolic model checking without bdds. In *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS'99, Held as Part of ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *LNCS*, pages 193–207. Springer.

[Boehm, 1981] Boehm, B. (1981). *Software Engineering Economics*. Prentice-Hall.

[Bosch, 2005] Bosch, J. (2005). Keynote—Software Product Families in Nokia. In *Proceedings of the 9th International Software Product Line Conference (SPLC)*, volume 3714 of *LNCS*, pages 2–6. Springer.

[Boucher et al., 2010a] Boucher, Q., Classen, A., Faber, P., and Heymans, P. (2010a). Introducing TVL, a text-based feature modelling language. In *Proceedings of the Fourth International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'10), Linz, Austria, January 27-29*, pages 159–162. University of Duisburg-Essen.

[Boucher et al., 2010b] Boucher, Q., Classen, A., Heymans, P., Bourdoux, A., and Demonceau, L. (2010b). Tag and prune: A pragmatic approach to software product line implementation. In *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 333–336. ACM.

[Brooks, 1975] Brooks, F. (1975). *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley.

[Brooks, 1987] Brooks, Jr., F. P. (1987). No silver bullet: Essence and accidents of software engineering. *Computer*, 20:10–19.

[Bruns and Godefroid, 2000] Bruns, G. and Godefroid, P. (2000). Generalized model checking: Reasoning about partial state spaces. In *Proceedings of the 11th International Conference on Concurrency Theory*, volume 1877 of *LNCS*, pages 168–182. Springer-Verlag.

[Bruns and Godefroid, 2001] Bruns, G. and Godefroid, P. (2001). Temporal logic query checking. In *16th Annual IEEE Symposium on Logic in Computer Science, 16-19 June 2001, Boston, Massachusetts, USA, Proceedings*, pages 409–417. IEEE Computer Society.

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

[Burch et al., 1990] Burch, J., Clarke, E. M., McMillan, K. L., and Dill, D. L. (1990). Sequential circuit verification using symbolic model checking. In *Proceedings of the 27th ACM/IEEE Conference on Design Automation (DAC)*, pages 46–51. IEEE Computer Society.

[Burch et al., 1992] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, L. J. (1992). Symbolic model checking: 10^{20} states and beyond. *Inf. Comp.*, 98(2):142–170.

[Calder et al., 2003] Calder, M., Kolberg, M., Magill, E. H., and Reiff-Marganiec, S. (2003). Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141.

[Calder and Miller, 2001] Calder, M. and Miller, A. (2001). Using SPIN for feature interaction analysis – a case study. In *Proceedings of the 8th international SPIN workshop on Model checking of software*, volume 2057 of *LNCS*, pages 143–162. Springer.

[Cavada et al., 2004] Cavada, R., Cimatti, A., Keighren, G., Olivetti, E., Pistore, M., and Roveri, M. (2004). *NuSMV 2.2 Tutorial*. ITC-irst, Trento, Italy.

[Chan, 2000] Chan, W. (2000). Temporal-locig queries. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *LNCS*, pages 450–463. Springer.

[Cheng et al., 2009] Cheng, B. H. C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G. D. M., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H. M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller,

H. A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., and Whittle, J. (2009). Software engineering for self-adaptive systems: A research roadmap. In *Software Engineering for Self-Adaptive Systems*, volume 5525 of *LNCS*, pages 1–26. Springer.

[Chiappini et al., 2010] Chiappini, A., Cimatti, A., Macchi, L., Rebollo, O., Roveri, M., Susi, A., Tonetta, S., and Vittorini, B. (2010). Formalization and validation of a subset of the european train control system. In *Proceedings of the 32nd International Conference on Software Engineering (ICSE), Companion Volume*, pages 109–118. IEEE.

[Cimatti et al., 2000] Cimatti, A., Clarke, E. M., Giunchiglia, F., and Roveri, M. (2000). Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425.

[Clarke et al., 1999] Clarke, E., Grumberg, O., and Peled, D. (1999). *Model Checking*. MIT Press.

[Clarke and Emerson, 1982] Clarke, E. M. and Emerson, E. A. (1982). Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs, Workshop*, pages 52–71, London, UK. Springer-Verlag.

[Clarke et al., 1986] Clarke, E. M., Emerson, E. A., and Sistla, A. P. (1986). Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263.

[Clarke et al., 2000] Clarke, E. M., Grumberg, O., Jha, S., Lu, Y., and Veith, H. (2000). Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, number 1855 in LNCS, pages 154–169. Springer.

[Clarke et al., 1995] Clarke, E. M., Grumberg, O., McMillan, K. L., and Zhao, X. (1995). Efficient generation of counterexamples and witnesses in symbolic model checking. In *32nd ACM/IEEE Conference on Design Automation (DAC)*, pages 427–432. IEEE CS.

[Classen, 2007] Classen, A. (2007). Problem-oriented modelling and verification of software product lines. Master's thesis, Computer Science Department, University of Namur, Belgium.

[Classen, 2010a] Classen, A. (2010a). CTL model checking for software product lines in NuSMV. Technical Report P-CS-TR SPLMC-00000002, PReCISE Research Center, University of Namur.

[Classen, 2010b] Classen, A. (2010b). www.info.fundp.ac.be/~acs/fts.

[Classen, 2010c] Classen, A. (2010c). Modelling with FTS: a collection of illustrative examples. Technical Report P-CS-TR SPLMC-00000001, PReCISE Research Center, University of Namur, Namur, Belgium.

[Classen et al., 2010a] Classen, A., Boucher, Q., Faber, P., and Heymans, P. (2010a). The TVL specification. Technical Report P-CS-TR SPLBT-00000003, PReCISE Research Center, University of Namur, Namur, Belgium.

[Classen et al., 2011a] Classen, A., Boucher, Q., and Heymans, P. (2011a). A text-based approach to feature modelling: Syntax and semantics of TVL. *Science of Computer Programming, Special Issue on Software Evolution, Adaptability and Variability*, 76(12):1130–1143.

[Classen et al., 2011b] Classen, A., Cordy, M., Heymans, P., Schobbens, P.-Y., and Legay, A. (2011b). SNIP: An efficient model checker for software product lines. Technical Report P-CS-TR SPLMC-00000003, PReCISE Research Center, University of Namur.

[Classen et al., 2008a] Classen, A., Heymans, P., and Schobbens, P.-Y. (2008a). What's in a feature: A requirements engineering perspective. In Fiadeiro, J. L. and Inverardi, P., editors, *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), Held as Part of ETAPS'08*, volume 4961 of *LNCS*, pages 16–30. Springer.

[Classen et al., 2011c] Classen, A., Heymans, P., Schobbens, P.-Y., and Legay, A. (2011c). Symbolic model checking of software product lines. In *33rd International Conference on Software Engineering, ICSE 2011, May 21-28, 2011, Waikiki, Honolulu, Hawaii, Proceedings*, pages 321–330. ACM.

[Classen et al., 2010b] Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2010b). Model checking lots of systems: Efficient verification of temporal properties in software product lines. In *32nd International Conference on Software Engineering, ICSE 2010, May 2-8, 2010, Cape Town, South Africa, Proceedings*, pages 335–344. ACM.

[Classen et al., 2011d] Classen, A., Heymans, P., Schobbens, P.-Y., Legay, A., and Raskin, J.-F. (2011d). Modelling and model checking variability-intensive systems with fts. Submitted for review to IEEE Trans. Softw. Eng.

[Classen et al., 2009a] Classen, A., Heymans, P., Tun, T. T., and Nuseibeh, B. (2009a). Towards safer composition. In Harald C. Gall, A. O., editor, *Proceedings of the 31st International Conference on Software Engineering (ICSE), Companion Volume, New Ideas and Emerging Results Track*, pages 227–230, Vancouver, Canada. ACM, IEEE CS.

[Classen et al., 2009b] Classen, A., Hubaux, A., and Heymans, P. (2009b). Analysis of feature configuration workflows (poster). In *Proceedings of the 17th IEEE International Requirements Engineering Conference (RE'09), Atlanta, Georgia, USA*, pages 381–382. IEEE.

[Classen et al., 2009c] Classen, A., Hubaux, A., and Heymans, P. (2009c). A formal semantics for multi-level staged configuration. In *Proceedings of the Third International Workshop on Variability Modelling of Software-intensive Systems (VaMoS'09), Sevilla, Spain, January 28-30*. University of Duisburg-Essen.

[Classen et al., 2008b] Classen, A., Hubaux, A., Sanen, F., Truyen, E., Vallejos, J., Costanza, P., Meuter, W. D., Heymans, P., and Joosen, W. (2008b). Modelling variability in self-adaptive systems: Towards a research agenda. In *Workshop on Modularization, Composition and Generative Techniques for Product Line Engineering (McGPLE), held in conjuction with GPCE / OOPSLA 2008, Nashville, TN, October 19-23*.

[Classen et al., 2008c] Classen, A., Laney, R., Tun, T. T., Heymans, P., and Hubaux, A. (2008c). Using the event calculus to reason about problem diagrams. In *Proceedings of the Third International Workshop on Advances and Applications of Problem Frames (IWAAPF'08). Co-located with ICSE 2008.*, Leipzig, Germany.

[Clements and Northrop, 2001] Clements, P. C. and Northrop, L. (2001). *Software Product Lines: Practices and Patterns*. SEI Series in Software Engineering. Addison-Wesley.

[Coffman et al., 1971] Coffman, E. G., Elphick, M., and Shoshani, A. (1971). System deadlocks. *ACM Comput. Surv.*, 3:67–78.

[Cohen et al., 2006] Cohen, M. B., Dwyer, M. B., and Shi, J. (2006). Coverage and adequacy in software product line testing. In *Workshop on Role of Software Architecture for Testing and Analysis, ROSATEA 2006, held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2006), Proceedings*, pages 53–63. ACM.

[Cohen et al., 2007] Cohen, M. B., Dwyer, M. B., and Shi, J. (2007). Interaction testing of highly-configurable systems in the presence of constraints. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, Proceedings*, pages 129–139. ACM.

[Consultative Committee for Space Data Systems (CCSDS), 2007] Consultative Committee for Space Data Systems (CCSDS) (2007). *CCSDS File Delivery Protocol (CFDP): Blue Book, Issue 4*. National Aeronautics and Space Administration (NASA). Number CCSDS 727.0-B-4.

[Corbett et al., 2000] Corbett, J., Dwyer, M., Hatcliff, J., Pasareanu, C., Robby, Laubach, S., and Zheng, H. (2000). Bandera : Extracting finite-state models from java source code. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE), Limeric, Ireland*, pages 439–448. ACM Press.

[Courcoubetis et al., 1992] Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. (1992). Memory-efficient algorithms for the verification of temporal properties. *Form. Methods Syst. Des.*, 1(2-3):275–288.

[Czarnecki and Antkiewicz, 2005] Czarnecki, K. and Antkiewicz, M. (2005). Mapping features to models: A template approach based on superimposed variants. In *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering (GPCE)*, pages 422–437.

[Czarnecki and Eisenecker, 2000] Czarnecki, K. and Eisenecker, U. W. (2000). *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley.

[Czarnecki et al., 2005] Czarnecki, K., Helsen, S., and Eisenecker, U. W. (2005). Staged configuration through specialization and multi-level configuration of feature models. *Software Process: Improvement and Practice*, 10(2):143–169.

[Czarnecki and Pietroszek, 2006] Czarnecki, K. and Pietroszek, K. (2006). Verifying feature-based model templates against well-formedness ocl constraints. In *GPCE '06: Proceedings of the 5th international conference on Generative programming and component engineering*, pages 211–220, New York, NY, USA. ACM Press.

[Davis, 1987] Davis, S. M. (1987). *Future Perfect*. Addison-Wesley.

[De Wulf et al., 2006] De Wulf, M., Doyen, L., Henzinger, T. A., and Raskin, J.-F. (2006). Antichains: A new algorithm for checking universality of finite automata. In *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *LNCS*, pages 17–30.

[Dijkstra, 1969] Dijkstra, E. W. (1969). Notes on structured programming. Technical Report T.H.-Report 70-WSK-03, Technological University Eindhoven, The Netherlands, Department of Mathematics, Eindhoven, The Netherlands.

[Dijkstra, 1970] Dijkstra, E. W. (1970). Structrured programming. In *Software Engineering Techniques: Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27th to 31st October 1969*, pages 84–87.

[Dijkstra, 1972] Dijkstra, E. W. (1972). The humble programmer. *Commun. ACM*, 15:859–866.

[D'Ippolito et al., 2008] D'Ippolito, N., Fischbein, D., Chechik, M., and Uchitel, S. (2008). Mtsa: The modal transition system analyser. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 475–476.

[Easterbrook, 1996] Easterbrook, S. (1996). The role of independent v&v in upstream software development processes. In *Proceedings, 2nd World Conference on Integrated Design and Process Technology (IDPT)*, Austin, Texas. Keynote address to the track entitled "The Process Road from Requirements to System Architectures (and back)".

[Ebert and Jones, 2009] Ebert, C. and Jones, C. (2009). Embedded software: Facts, figures, and future. *Computer*, 42(4):42–52.

[Eden, 2007] Eden, A. H. (2007). Three paradigms of computer science. *Minds Mach.*, 17:135–167.

[Emerson and Lei, 1987] Emerson, E. A. and Lei, C.-L. (1987). Modalities for model checking: branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306.

[Emerson and Namjoshi, 1996] Emerson, E. A. and Namjoshi, K. S. (1996). Automatic verification of parameterized synchronous systems (extended abstract). In *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, July 31 - August 3, 1996, Proceedings*, volume 1102 of *LNCS*, pages 87–98. Springer.

[Fantechi and Gnesi, 2007] Fantechi, A. and Gnesi, S. (2007). A behavioural model for product families. In *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*, pages 521–524, New York, NY, USA. ACM.

[Fantechi and Gnesi, 2008] Fantechi, A. and Gnesi, S. (2008). Formal modeling for product families engineering. In *12th International Conference on Software Product Lines (SPLC 2008)*, pages 193–202.

[Fischbein et al., 2006] Fischbein, D., Uchitel, S., and Braberman, V. (2006). A foundation for behavioural conformance in software product line architectures. In *ROSATEA '06: Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, pages 39–48, New York, NY, USA. ACM Press.

[Fisler and Krishnamurthi, 2001] Fisler, K. and Krishnamurthi, S. (2001). Modular verification of collaboration-based software designs. In *8th*

European Software Engineering Conference held jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2001, Vienna, Austria, Proceedings, pages 152–163. ACM.

[Francez and Forman, 1990] Francez, N. and Forman, I. R. (1990). Superimposition for interacting processes. In *CONCUR'90, Theories of Concurrency: Unification and Extension*, volume 458 of *LNCS*, pages 230–245. Springer.

[Garrido and Johnson, 2005] Garrido, A. and Johnson, R. (2005). Analyzing multiple configurations of a C program. In *International Conference on Software Maintenance, ICSM 2005, Proceedings*, pages 379–388. IEEE Computer Society.

[Gastin and Oddoux, 2001] Gastin, P. and Oddoux, D. (2001). Fast ltl to büchi automata translation. In *13th International Conference on Computer Aided Verification, CAV 2001, Paris, France, July 18-22*, number 2102 in LNCS, pages 53–65.

[Giannakopoulou and Magee, 2003] Giannakopoulou, D. and Magee, J. (2003). Fluent model checking for event-based systems. In *Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-11, pages 257–266. ACM.

[Giannakopoulou et al., 2005] Giannakopoulou, D., Pasareanu, C. S., Lowry, M., and Washington, R. (2005). Lifecycle verification of the nasa ames k9 rover executive. In *Verification and Validation meets Planning and Scheduling (co-located with ICAPS 2005)*, pages 75–85.

[Godefroid et al., 2001] Godefroid, P., Huth, M., and Jagadeesan, R. (2001). Abstraction-based model checking using modal transition systems. In *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR)*, volume 2154 of *LNCS*, pages 426–440. Springer-Verlag.

[Godefroid and Piterman, 2009] Godefroid, P. and Piterman, N. (2009). LTL generalized model checking revisited. In *VMCAI '09: Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 89–104, Berlin, Heidelberg. Springer-Verlag.

[Goldman and Katz, 2007] Goldman, M. and Katz, S. (2007). Maven: Modular aspect verification. In *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of ETAPS 2007*, pages 308–322, Braga, Portugal.

[Gruler et al., 2008a] Gruler, A., Leucker, M., and Scheidemann, K. (2008a). Calculating and modeling common parts of software product lines. In *SPLC '08: Proceedings of the 2008 12th International Software Product*

Line Conference, pages 203–212, Washington, DC, USA. IEEE Computer Society.

[Gruler et al., 2008b] Gruler, A., Leucker, M., and Scheidemann, K. (2008b). Modeling and model checking software product lines. In *FMOODS '08: Proceedings of the 10th IFIP WG 6.1 international conference on Formal Methods for Open Object-Based Distributed Systems*, pages 113–131, Berlin, Heidelberg. Springer-Verlag.

[Harel and Rumpe, 2000] Harel, D. and Rumpe, B. (2000). Modeling languages: Syntax, semantics and all that stuff - part I: The basic stuff. Technical Report MCS00-16, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Israel.

[Havelund et al., 2001] Havelund, K., Lowry, M., and Penix, J. (2001). Formal analysis of a space-craft controller using spin. *IEEE Trans. Softw. Eng.*, 27(8):749–765.

[Hawking, 1988] Hawking, S. (1988). *A Brief History of Time*. Bantam Dell.

[Hay and Atlee, 2000] Hay, J. D. and Atlee, J. M. (2000). Composing features and resolving interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 110–119, New York, NY, USA. ACM.

[Hennessy and Milner, 1980] Hennessy, M. and Milner, R. (1980). On observing nondeterminism and concurrency. In *Proceedings of the 7th Colloquium on Automata, Languages and Programming*, pages 299–309, London, UK. Springer-Verlag.

[Hoare, 1969] Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580.

[Holzmann, 1997] Holzmann, G. J. (1997). State compression in spin. In *Proceedings of the Third Spin Workshop, Twente University, The Netherlands.*

[Holzmann, 2004] Holzmann, G. J. (2004). *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley.

[Holzmann and Smith, 1999a] Holzmann, G. J. and Smith, M. H. (1999a). A practical method for verifying event-driven software. In *Proceedings of the 21st international conference on Software engineering (ICSE 1999)*, pages 597–607. ACM.

[Holzmann and Smith, 1999b] Holzmann, G. J. and Smith, M. H. (1999b). Software model checking: extracting verification models from source code. In *Proceedings of the IFIP TC6 WG6.1 Joint Int. Conference on Formal Description Techniques for Distributed systems and Communicaton Protocols*

(FORTE XII) and Protocol Specification, Testing and Verification (PSTV XIX), volume 156 of *IFIP Conference Proceedings*, pages 481–497. Kluwer.

[Hopcroft et al., 2000] Hopcroft, J. E., Motwani, R., and Ullman, J. D. (2000). *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison-Wesley.

[Hubaux et al., 2009] Hubaux, A., Classen, A., and Heymans, P. (2009). Formal modelling of feature configuration workflows. In McGregor, J. D. and Muthig, D., editors, *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*, volume 446 of *ACM International Conference Proceeding Series*, pages 221–230. ACM.

[IEEE, 1990] IEEE (1990). IEEE standard glossary of software engineering terminology. IEEE std 610.12-1990.

[Jackson and Zave, 1998] Jackson, M. and Zave, P. (1998). Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Trans. Softw. Eng.*, 24(10):831–847.

[Kang et al., 1990] Kang, K., Cohen, S., Hess, J., Novak, W., and Peterson, S. (1990). Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University.

[Kang et al., 1998] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E., and Huh, M. (1998). Form: A feature-oriented reuse method with domain-specific reference architectures. *Annales of Software Engineering*, 5:143–168.

[Kästner and Apel, 2008] Kästner, C. and Apel, S. (2008). Type-checking software product lines - a formal approach. In *Proceedings on the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 258–267. IEEE Computer Society.

[Kästner et al., 2008] Kästner, C., Apel, S., and Kuhlemann, M. (2008). Granularity in software product lines. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 311–320, New York, NY, USA. ACM.

[Kästner et al., 2010] Kästner, C., Giarrusso, P. G., and Ostermann, K. (2010). Partial preprocessing C code for variability analysis. In *Fifth International Workshop on Variability Modelling of Software-intensive Systems, VaMoS 2011, Proceedings*, pages 127–136.

[Katz and Katz, 2008] Katz, E. and Katz, S. (2008). Incremental analysis of interference among aspects. In *Proceedings of the 7th Workshop on Foundations of Aspect-Oriented Languages 2008, held with AOSD'08, Brussels, Belgium*, pages 29–38. ACM.

[Keck and Kuehn, 1998] Keck, D. O. and Kuehn, P. J. (1998). The feature and service interaction problem in telecommunications systems: A survey. *IEEE Trans. Softw. Eng.*, 24(10):779–796.

[Khurshid et al., 2003] Khurshid, S., Pasareanu, C. S., and Visser, W. (2003). Generalized symbolic execution for model checking and testing. In *Proceedings of TACAS 2003, Warsaw, Poland, April 2003*, volume 2619 of *LNCS*, pages 553–568. Springer.

[Kiczales et al., 1997] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *ECOOP'97 - Object-Oriented Programming, 11th European Conference, Jyväskylä, Finland, June 9-13, 1997, Proceedings*, volume 1241 of *LNCS*, pages 220–242. Springer.

[Kim et al., 2010] Kim, C. H. P., Bodden, E., Batory, D. S., and Khurshid, S. (2010). Reducing configurations to monitor in a software product line. In *Runtime Verification - First International Conference, RV 2010, St. Julians, Malta, November 1-4, 2010. Proceedings*, volume 6418 of *LNCS*, pages 285–299.

[King, 1976] King, J. C. (1976). Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394.

[Kishi and Noda, 2006] Kishi, T. and Noda, N. (2006). Formal verification and software product lines. *Commun. ACM*, 49(12):73–77.

[Koscher et al., 2010] Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., and Savage, S. (2010). Experimental security analysis of a modern automobile. In *IEEE Symposium on Security and Privacy, Oakland, CA, USA*, pages 447–462. IEEE.

[Kramer et al., 1983] Kramer, J., Magee, J., Sloman, M., and Lister, A. (1983). Conic: an integrated approach to distributed computer control systems. *Computers and Digital Techniques, IEE Proceedings E*, 130(1):1–10.

[Kripke, 1963] Kripke, S. (1963). Semantical considerations on modal logic. *Acta Philosophica Fennica*, 16:83–94.

[Krishnamurthi et al., 2004] Krishnamurthi, S., Fisler, K., and Greenberg, M. (2004). Verifying aspect advice modularly. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 137–146, New York, NY, USA. ACM.

[Laddad, 2003] Laddad, R. (2003). *AspectJ in Action – Practical Aspect-Oriented Programming*. Manning Publication Co.

[Lamport, 1977] Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3:125–143.

[Larsen, 1989] Larsen, K. G. (1989). Modal specifications. In *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 232–246. Springer.

[Larsen et al., 2007] Larsen, K. G., Nyman, U., and Wasowski, A. (2007). Modal i/o automata for interface and product line theories. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of ETAPS 2007, Braga, Portugal, March 24 - April 1, 2007, Proceedings*, pages 64–79.

[Larsen and Thomsen, 1988] Larsen, K. G. and Thomsen, B. (1988). A modal process logic. In *Proceedings of the Third Annual symposium on Logic in Computer Science (LICS), Edinburgh*, pages 203–210. IEEE Computer Society.

[Lauenroth and Pohl, 2008] Lauenroth, K. and Pohl, K. (2008). Dynamic consistency checking of domain requirements in product line engineering. In *16th IEEE International Requirements Engineering Conference, RE 2008, 8-12 September 2008, Barcelona, Catalunya, Spain*, pages 193–202. IEEE Computer Society.

[Lauenroth et al., 2009] Lauenroth, K., Töhning, S., and Pohl, K. (2009). Model checking of domain artifacts in product line engineering. In *Proceedings on the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 269–280. IEEE.

[Lee et al., 2002] Lee, K., Kang, K. C., and Lee, J. (2002). Concepts and guidelines of feature modeling for product line software engineering. In *ICSR-7: Proceedings of the 7th International Conference on Software Reuse*, pages 62–77, London, UK. Springer-Verlag.

[Leveson and Turner, 1993] Leveson, N. G. and Turner, C. S. (1993). An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41.

[Leveson and Weiss, 2004] Leveson, N. G. and Weiss, K. A. (2004). Making embedded software reuse practical and safe. In *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2004*, SIGSOFT '04/FSE-12, pages 171–178. ACM.

[Li et al., 2002a] Li, H. C., Krishnamurthi, S., and Fisler, K. (2002a). Interfaces for modular feature verification. In *17th IEEE International Conference on Automated Software Engineering (ASE 2002)*, pages 195–204.

[Li et al., 2002b] Li, H. C., Krishnamurthi, S., and Fisler, K. (2002b). Verifying cross-cutting features as open systems. In *Proceedings of the 10th*

ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2002, pages 89–98.

[Liebig et al., 2010] Liebig, J., Apel, S., Lengauer, C., Kästner, C., and Schulze, M. (2010). An analysis of the variability in forty preprocessor-based software product lines. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 105–114.

[Liu et al., 2007] Liu, J., Dehlinger, J., and Lutz, R. (2007). Safety analysis of software product lines using state-based modeling. *J. Syst. Softw.*, 80(11):1879–1892.

[Magee and Kramer, 2006] Magee, J. and Kramer, J. (2006). *Concurrency: State Models & Java Programs*. Wiley.

[Mannion, 2002] Mannion, M. (2002). Using first-order logic for product line model validation. In *Proceedings of the Second Software Product Line Conference (SPLC'02)*, LNCS 2379, pages 176–187, San Diego, CA. Springer.

[McIlroy, 1968] McIlroy, M. D. (1968). Mass-produced software components. In Naur, P. and Randell, B., editors, *Software Engineering, Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October*, pages 138–155.

[McMillan, 1993] McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer.

[Mendonca, 2009] Mendonca, M. (2009). *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, Waterloo, Ontario, Canada, 2009.

[Mendonca et al., 2009] Mendonca, M., Wasowski, A., and Czarnecki, K. (2009). Sat-based analysis of feature models is easy. In *Proceedings of the 13th International Software Product Lines Conference (SPLC'09), San Francisco, CA, USA*, pages 231–240.

[Metzger et al., 2005] Metzger, A., Bühne, S., Lauenroth, K., and Pohl, K. (2005). *Feature Interactions in Telecommunications and Software Systems VIII. (ICFI'05)*, chapter Considering Feature Interactions in Product Lines: Towards the Automatic Derivation of Dependencies between Product Variants, pages 198–216. IOS Press, Leicester, UK.

[Metzger et al., 2007] Metzger, A., Heymans, P., Pohl, K., Schobbens, P.-Y., and Saval, G. (2007). Disambiguating the documentation of variability in software product lines: A separation of concerns, formalization and automated analysis. In *Proceedings of the 15th IEEE International Requirements Engineering Conference (RE'07)*, pages 243–253, New Delhi, India.

[Mezini and Ostermann, 2004] Mezini, M. and Ostermann, K. (2004). Variability management with feature-oriented programming and aspects. In *Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 127–136. ACM.

[Milner, 1980] Milner, R. (1980). *A Calculus of Communicating Systems*. Springer.

[Morin et al., 2009] Morin, B., Barais, O., Nain, G., and Jézéquel, J.-M. (2009). Taming dynamically adaptive systems using models and aspects. In *ICSE '09: Proceedings of the 31st international conference on Software engineering*, pages 122–132. IEEE.

[Naur and Randell, 1968] Naur, P. and Randell, B., editors (1968). *Software Engineering: Report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7th to 11th October*.

[Oster et al., 2010] Oster, S., Markert, F., and Ritter, P. (2010). Automated incremental pairwise testing of software product lines. In *Software Product Lines: Going Beyond - 14th International Conference, SPLC 2010, Proceedings*, volume 6287 of *LNCS*, pages 196–210. Springer.

[Papadimitriou, 1994] Papadimitriou, C. H. (1994). *Computational Complexity*. Addison-Wesley.

[Parnas, 1976] Parnas, D. (1976). On the design and development of program families. *Transactions on Software Engineering*, SE-2(1):1–9.

[Parnas, 1971] Parnas, D. L. (1971). Information distribution aspects of design methodology. In *IFIP Congress (1)*, pages 339–344.

[Parnas, 1972] Parnas, D. L. (1972). On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058.

[Perrouin et al., 2010] Perrouin, G., Sen, S., Klein, J., Baudry, B., and Traon, Y. L. (2010). Automated and scalable t-wise test case generation strategies for software product lines. In *Third International Conference on Software Testing, Verification and Validation, ICST 2010, Proceedings*, pages 459–468. IEEE Computer Society.

[Plath and Ryan, 2001] Plath, M. and Ryan, M. (2001). Feature integration using a feature construct. *Sci. Comput. Program.*, 41(1):53–84.

[Plath and Ryan, 2000] Plath, M. and Ryan, M. D. (2000). The feature construct for smv: Semantics. In *Feature Interactions in Telecommunications and Software Systems VI, May 17-19, 2000, Glasgow, Scotland, UK*, pages 129–144. IOS Press.

[Pnueli, 1977] Pnueli, A. (1977). The temporal logic of programs. In *Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 46–57.

[Pohl et al., 2005] Pohl, K., Bockle, G., and van der Linden, F. (2005). *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer.

[Pohl and Metzger, 2006] Pohl, K. and Metzger, A. (2006). Variability management in software product line engineering. 28th International Conference on Software Engineering (ICSE'06). Tutorial F4.

[Post and Sinz, 2008] Post, H. and Sinz, C. (2008). Configuration lifting: Verification meets software configuration. In *Proceedings on the 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 188–197.

[Prehofer, 1997] Prehofer, C. (1997). Feature-oriented programming: A fresh look at objects. In *In Proc. of European Conf. on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 419–443. Springer.

[Prior, 1967] Prior, A. (1967). *Past, Present and Future*. Oxford University Press.

[Queille and Sifakis, 1982] Queille, J. P. and Sifakis, J. (1982). Specification and verification of concurrent systems in cesar. In *International Symposium on Programming*, volume 137 of *LNCS*, pages 337–351.

[Rabiser et al., 2007] Rabiser, R., Grunbacher, P., and Dhungana, D. (2007). Supporting product derivation by adapting and augmenting variability models. In *SPLC '07: Proceedings of the 11th International Software Product Line Conference*, pages 141–150, Washington, DC, USA. IEEE Computer Society.

[Reiser, 2009] Reiser, M.-O. (2009). Core concepts of the compositional variability management framework (CVM). Technical report, Technische Universität Berlin.

[Reiser et al., 2007] Reiser, M.-O., Tavakoli, R., and Weber, M. (2007). Unified feature modeling as a basis for managing complex system families. In *Proceedings of the First International Workshop on Variability Modelling of Software-intensive Systems*, pages 79–86, Limerick, Ireland.

[Reiser and Weber, 2006] Reiser, M.-O. and Weber, M. (2006). Managing highly complex product families with multi-level feature trees. In *IEEE International Conference on Requirements Engineering (RE'06)*, pages 146–155, Los Alamitos, CA, USA. IEEE Computer Society.

[Savitch, 1970] Savitch, W. J. (1970). Relationships between nondeterministic and deterministic tape complexities. *Journal of Computer and System Sciences*, 4(2):177–192.

[Schnoebelen, 2002] Schnoebelen, P. (2002). The complexity of temporal logic model checking. In *Advances in Modal Logic 4, papers from the fourth conference on Advances in Modal logic, held in Toulouse (France) in October 2002*, pages 393–436.

[Schobbens et al., 2006] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y. (2006). Feature Diagrams: A Survey and A Formal Semantics. In *Proceedings of the 14th IEEE International Requirements Engineering Conference (RE'06)*, pages 139–148, Minneapolis, Minnesota, USA.

[Schobbens et al., 2007] Schobbens, P.-Y., Heymans, P., Trigaux, J.-C., and Bontemps, Y. (2007). Generic semantics of feature diagrams. *Computer Networks, Special Issue on Feature Interactions in Emerging Application Domain*, 51(2):456–479.

[Sistla and Clarke, 1985] Sistla, A. P. and Clarke, E. M. (1985). The complexity of propositional linear temporal logics. *J. ACM*, 32:733–749.

[Smaragdakis and Batory, 2002] Smaragdakis, Y. and Batory, D. (2002). Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Trans. Softw. Eng. Methodol.*, 11(2):215–255.

[Tarski, 1955] Tarski, A. (1955). A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math.*, 5:285–309.

[Thaker et al., 2007] Thaker, S., Batory, D. S., Kitchin, D., and Cook, W. (2007). Safe composition of product lines. In *Generative Programming and Component Engineering, 6th International Conference, GPCE 2007, Salzburg, Austria, October 1-3, 2007, Proceedings*, pages 95–104.

[Thiel and Hein, 2002] Thiel, S. and Hein, A. (2002). Modeling and using product line variability in automotive systems. *IEEE Software*, 19(4):66–72.

[Trigaux, 2008] Trigaux, J.-C. (2008). *Quality of Feature Diagram Languages: Formal Evaluation and Comparison.* PhD thesis, Faculty of Computer Science, University of Namur (FUNDP).

[van der Aalst et al., 2008] van der Aalst, W. M. P., Dumas, M., Gottschalk, F., ter Hofstede, A. H. M., Rosa, M. L., and Mendling, J. (2008). Correctness-preserving configuration of business process models. In *Proceedings of the 11th International Conference on Fundamental Approaches to Software Engineering (FASE'08), Held as Part of ETAPS'08*, pages 46–61.

[van Deursen and Klint, 2002] van Deursen, A. and Klint, P. (2002). Domain-Specific Language Design Requires Feature Descriptions. *Journal of Computing and Information Technology*, 10(1):1–17.

[van Gurp et al., 2001] van Gurp, J., Bosch, J., and Svahnberg, M. (2001). On the Notion of Variability in Software Product Lines. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, pages 45–54. IEEE.

[Vardi, 1985] Vardi, M. Y. (1985). Automatic verification of probabilistic concurrent finite-state programs. In *26th IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 327–338. IEEE Computer Society.

[Vardi, 2001] Vardi, M. Y. (2001). Branching vs. linear time: Final showdown. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–22, London, UK. Springer-Verlag.

[Vardi and Wolper, 1986] Vardi, M. Y. and Wolper, P. (1986). An automata-theoretic approach to automatic program verifiication. In *Proceddings of the 1st IEEE Symp. Logic in Computer Science (LICS'86)*, pages 332–344. IEEE CS.

[Visser et al., 2003] Visser, W., Havelund, K., Brat, G., Park, S., and Lerda, F. (2003). Model checking programs. *Automated Software Engineering Journal*, 10(2):203–232.

[Visser et al., 2000] Visser, W., Havelund, K., Brat, G. P., and Park, S. (2000). Model checking programs. In *Proceedings of the 15th International Conference on Automated Software Engineering (ASE), Grenoble, France*, pages 3–12.

[Voelter and Groher, 2007] Voelter, M. and Groher, I. (2007). Product line implementation using aspect-oriented and model-driven software development. In *Software Product Lines, 11th International Conference, SPLC 2007, Proceedings*, pages 233–242. IEEE Computer Society.

[Wegner, 1976] Wegner, P. (1976). Research paradigms in computer science. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 322–330, Los Alamitos, CA, USA. IEEE Computer Society Press.

[Wei et al., 2009] Wei, O., Gurfinkel, A., and Chechik, M. (2009). Mixed transition systems revisited. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI '09, pages 349–365, Berlin, Heidelberg. Springer-Verlag.

[Zave, 2008] Zave, P. (2008). Understanding sip through model-checking. In *Principles, Systems and Applications of IP Telecommunications. Services and Security for Next Generation Networks, Second International Conference, IPTComm 2008, Heidelberg, Germany, July 1-2, 2008. Revised Selected Papers*, volume 5310 of *LNCS*, pages 256–279. Springer.

[Zave and Jackson, 1997] Zave, P. and Jackson, M. A. (1997). Four dark corners of requirements engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30.

[Ziadi et al., 2003] Ziadi, T., Hélouët, L., and Jézéquel, J.-M. (2003). Towards a uml profile for software product lines. In *Software Product-Family Engineering, 5th International Workshop, PFE 2003, Siena, Italy, November 4-6, 2003, Revised Papers*, pages 129–139.

Index

T