Efficient Reasoning Techniques for Large Scale Feature Models

by

Marcílio Mendonça

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2009

Abstract

In Software Product Lines (SPLs), a feature model can be used to represent the similarities and differences within a family of software systems. This allows describing the systems derived from the product line as a unique combination of the features in the model. What makes feature models particularly appealing is the fact that the constraints in the model prevent incompatible features from being part of the same product.

Despite the benefits of feature models, constructing and maintaining these models can be a laborious task especially in product lines with a large number of features and constraints. As a result, the study of automated techniques to reason on feature models has become an important research topic in the SPL community in recent years. Two techniques, in particular, have significant appeal for researchers: SAT solvers and Binary Decision Diagrams (BDDs). Each technique has been applied successfully for over four decades now to tackle many practical combinatorial problems in various domains. Currently, several approaches have proposed the compilation of feature models to specific logic representations to enable the use of SAT solvers and BDDs.

In this thesis, we argue that several critical issues related to the use of SAT solvers and BDDs have been consistently neglected. For instance, satisfiability is a well-known NP-complete problem which means that, in theory, a SAT solver might be unable to check the satisfiability of a feature model in a feasible amount of time. Similarly, it is widely known that the size of BDDs can become intractable for large models. At the same time, we currently do not know precisely whether these are real issues when feature models, especially large ones, are compiled to SAT and BDD representations.

Therefore, in our research we provide a significant step forward in the state-of-the-art by examining deeply many relevant properties of the feature modeling domain and the mechanics of SAT solvers and BDDs and the sensitive issues related to these techniques when applied in that domain. Specifically, we provide more accurate explanations for the space and/or time (in)tractability of these techniques in the feature modeling domain, and enhance the algorithmic performance of these techniques for reasoning on feature models. The contributions of our work include the proposal of novel heuristics to reduce the size of BDDs compiled from feature models, several insights on the construction of efficient domain-specific reasoning algorithms for feature models, and empirical studies to evaluate the efficiency of SAT solvers in handling very large feature models.

Acknowledgements

First and foremost, I offer my sincerest gratitude to my supervisor Donald Cowan for all his support and guidance during my PhD program and for being a close friend during the past couple of years. I still recall my first meeting with Prof. Cowan when he welcomed me to Canada and to the University of Waterloo with a sincere and encouraging smile on his face.

I also acknowledge my co-supervisor Prof. Krzysztof Czarnecki for his valuable feedback and for raising my research to new levels. Despite his busy schedule, he managed to review and discuss with me many of the ideas in this thesis.

To my parents Carlos and Edilma Mendonça without whom nothing, literally, would be possible. I love you so much! My big thanks to my Big Brother Fabio Mendonça for being a great source of inspiration for me and to my lovely sister Ana Carolina (Caroline) for her love and care.

I would like to send a big hug to my best friend at the university, William Malyk (Bill). Bill taught me a lot about the Canadian culture and gave me many English lessons for free (possibly paid with drinks). I felt so proud to be his best man in his unforgettable wedding. I also recall the memorable moments at Dooly's and our routinary "Friday research meetings" at the Grad House or at the kick-off pub where we would discuss everything from minding a high-tech business to planning his very first trip to Brazil.

A big thanks to my friends Prof. Toacy Oliveira and Prof. Paulo Alencar for their guidance in the preliminary stages of my research. I am also thankful to Prof. Gladimir Baranoski for his words of encouragement and for his valuable hints on how to succeed as a PhD student. Also, thanks to my friend Eduardo Barrenechea (Dudu) for being such a nice and helpful guy in and out of the lab.

My thanks also go to my friends at the Generative Software Development lab including Michal, Thiago, Rafael, Abbas, Matthew, Herman, Steven, and Mohamed.

Thanks to all members of my PhD defence committee for agreeing to serve as examiners.

Thanks also to CAPES, the Brazilian Ministry of Education Agency, and to the School of Computer Science through the David R. Cheriton graduate student scholarship for supporting me financially during the PhD program.

Obrigado a todos os amigos Brasileiros em Waterloo com os quais tive o enorme prazer de conviver por todos esses anos.

Dedication

I would like to dedicate this thesis to my "girls", i.e., my wife Érika Mendonça and my two princesses Anazilda and Marina Mendonça. Their love was absolutely indispensable to the construction of this work. In particular, I am thankful to Érika for her love and for working hard to ensure that we would always have enough to provide for our family; to Anazilda for her friendship, passion, love, and care all in one package; to Marina for always reminding me how many days were passed in my PhD program, i.e., Marina was born 1 day before I left Brazil to start my studies in Canada. I love you Marina, my "pernambucana arretada".

To my mom Edilma Mendonça,

my greatest source of inspiration.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The Software Product Line approach (SPLs) [25, 73, 26] is a contemporary paradigm for software development that exploits the similarities and differences within a family of systems in a particular domain of interest in order to provide a common infrastructure for deriving members of this family, i.e., software products, in a timely fashion, with high-quality standards, and at lower costs. To some extent, SPLs are similar to automotive product lines where it is possible to take a basic car model and choose such items as the engine, transmission, upholstery, and color. Substantial gains in productivity reported by industrial sectors adopting software product lines along with the rapid emergence of supporting approaches, techniques and tools have turned SPLs into a very attractive software approach that fits well in the current highly competitive market. Currently, the research field of software product lines is very active and academic research is focusing on real issues in the software industry.

In software product lines, *feature models* [48, 26] are used to represent the similarities and differences of system families in terms of *features*. This allows systems produced from the product line to be described as a unique combination of these features. The term *feature* is referred to in the literature as *"any prominent and distinctive aspect or characteristic that is visible to various stakeholders"* [48]. For instance, modern Web search engine systems usually provide a "search by language" feature that allows users to search for Web pages written in a specific language such as English or Portuguese. Another common feature of these systems focuses on the "types of searchable documents" supported such as text, images and videos. Therefore, a feature model can be used to represent "search by language" and "types of searchable documents" as features of a Web search engine product line. What makes feature models particularly appealing is the fact that they can be used to prevent the derivation of illegal product specifications. That is, the relations in the feature model constrain the way in which features can be combined into products. For instance, it might be the case that feature “search by language” is incompatible with particular types of searchable documents, say images and videos, in the Web search engine product line. In this case, a mutual-exclusion relation can be added to the feature model to prevent the generation of incorrect Web search engine systems. The process of selecting features in the feature model for a new system is commonly referred to as *product configuration*. The last decade has seen an ever growing number of approaches to software product lines supporting the notion of feature models [5, 75, 89, 50, 74, 15, 9, 59, 36, 72].

However, constructing and maintaining feature models can be a laborious task especially when the scale grows to models containing thousands of features. This has made the provision of automated tools and techniques critical for the effective manipulation of feature models. There are several practical contexts in which automated support for feature models is necessary as we discuss next.

First, feature models might contain errors and thus need to be debugged [92, 8, 12]. For instance, a feature model is incorrect if it is *unsatisfiable*, i.e., it does not contain any legal configuration. In fact, this is equivalent to stating that no valid products can be derived from the product line. In addition, a feature model is incorrect if it contains features that can never be part of any valid product, i.e., so called “dead” features. In this case, the relations in the feature model need to be revisited and fixed to eliminate these “dead” features. Manually checking the satisfiability of feature models or the presence of “dead” features is cumbersome hence automated support is definitely required.

Second, feature models might need to be adjusted over time to follow the evolution of the corresponding product line. When this happens, the changes made to the models must be checked formally for soundness. For instance, there are cases in which the new feature model must be backward-compatible with the original model in the sense that every legal configuration in the original model is also valid in the new model. In this case, we say the new model is a refactoring of the old one [3, 41]. Checking the soundness of refactorings manually is complex and error-prone, therefore there is a need for automated techniques.

Third, there are scenarios in which product configuration is performed interactively [30, 65] which might require the enforcement of certain properties. For instance, in *interactive configuration* [45], a feature model is configured through successive steps, by one or more users, until a final configuration is obtained. In this context, the configuration system must enforce that only valid choices are shown to users as decisions are made. This is important to enforce that the configuration process is backtrack-free, i.e., users are never required to revisit past decisions. Hence, appropriate automated techniques are required to enforce backtrack-freeness in interactive configuration.

Finally, there exist important metrics associated with feature models that can be used to measure different aspects of the corresponding product lines [10, 88]. For instance, the number of valid products in the product line can be computed by counting the number of legal configurations in the feature model. The computation of such metrics demands the use of appropriate automated techniques. We refer generally to the process of examining distinct aspects related to feature models across different contexts as *feature model reasoning*.

The need for effective techniques to reason on feature models has attracted the attention of researchers and practitioners in the field, notably in the past five years. In particular, the provision of rules to translate feature models to Boolean formulas [8, 12, 32] has opened many interesting research opportunities especially in the use of formalisms to improve automated support for feature model reasoning. For instance, several approaches have proposed the use of specific logic-based systems such as Alloy [40], SAT solvers [8, 33], constraint solvers [12, 33, 92], Binary Decision Diagrams (BDDs) [5, 33], Z [83], and Prolog [17] to reason on feature models. There are many trade-offs involved in using these systems and perhaps the best approach is to use them in combination. For example, while a SAT solver can efficiently check the satisfiability of a feature model, it can take an unfeasibly long time to compute the number of legal configurations in the model. Instead, a BDD can quickly count the legal configurations in the model but the size of the BDD structure can easily become intractable. Research contrasting some techniques exists [10, 33].

Two techniques are particularly appealing: SAT solvers and BDDs, since they are somewhat complementary. First, both techniques are very mature as they have been applied for over four decades now to tackle several practical combinatorial problems including scheduling, planning, configuration, logic synthesis, verification, and optimization. Therefore, we can take full advantage of all the improvements made to SAT and BDD technologies to provide improved support for feature model reasoning. Second, these techniques usually serve as a basis for other more general techniques. For instance, Alloy uses a SAT solver to perform formal analysis. Hence, the evaluation of SAT solvers can somehow be extended to Alloy and other SAT embedding techniques. Third, SAT and BDDs can be viewed as complementary techniques that together cover a variety of reasoning operations applied to feature models. Not surprisingly, the use of SAT solvers and BDDs to reason on feature models has already been tackled in the field [8, 12, 5, 66].

1.1 Problem Statement

Despite the relative success of approaches that make use of SAT solvers and BDDs to automate support for feature model reasoning there are several critical issues related to the use of these two systems that have been consistently neglected. For instance, it is well known that SAT solvers and BDDs can lead to space and/or time intractability problems. For instance, satisfiability is one of the most famous NP-complete problems discussed in the literature. This means that a SAT solver might take an infeasible amount of time to check the satisfiability of a Boolean formula, i.e., the algorithm might not terminate in feasible time. It is also widely known that the size of BDDs can vary dramatically depending on the order specified for its variables. In worst cases, the size of the BDD grows exponentially in the size of the Boolean formula and eventually becomes intractable. At the same time, we still do not know precisely whether these are real issues when SAT- and BDD-based techniques are applied in the feature modeling domain. If so, what are the current limits in algorithmic space and time and what can be done to improve these limits? If these are not issues, why is this the case and when, if ever, can these become issues? Similarly important, how can we take advantage of domain knowledge to boost automated support for feature model reasoning? We argue that current approaches to feature model reasoning based on SAT solvers and BDDs have adopted a "black-box" strategy, i.e., they have refrained from delving into the intricacies of these techniques and thus have failed to tackle related crucial sensitive issues. While this might be convenient, this also significantly decreases our level of confidence in using these techniques in the feature modeling domain as the following questions remain unanswered:

1. Can the size of BDDs for realistic feature models ever become intractable? If so, what are the current limits?
2. Can these limits be improved? If so, how and by how much?
3. Are SAT solvers always efficient in checking the satisfiability of feature models? If so, can we provide some explanations for that?
4. How can we take advantage of domain knowledge to improve the performance of algorithms to reason on feature models?

5. What kind of feature-modeling-specific algorithms can be developed and what are the improvements in performance in comparison to pure SAT solutions?

1.2 Research Hypothesis

Our research hypothesis is the following:

By exploring domain knowledge in the feature modeling domain and by better understanding the mechanics of BDDs and SAT solvers and the sensitive issues related to these techniques, we can i) provide more accurate explanations for the space and/or time (in)tractability of these techniques in the feature modeling domain, and ii) enhance the algorithmic performance of these techniques for reasoning on feature models.

1.3 Contributions

The main contributions of this thesis are the following:

1. **Reasoning with BDDs**
    - We empirically compare several state-of-the-art heuristics for ordering BDD variables and identify the one that is most effective in the feature modeling domain, i.e., the one that usually produces the smallest BDDs.
    - We identify several relevant properties of feature models that should be considered when ordering the variables of BDDs for feature models. Based on the insights provided we propose two novel heuristics to order the variables of BDDs compiled from feature models.
    - We show empirically that the new heuristics can compile, in average, feature models twice as large as those compiled by previous heuristics. The heuristics can be easily embedded in any BDD-based configuration tool which demonstrates the practical contribution of our research. We also expect that the insights we provide can influence the development of even better heuristics in the future.

2. **Reasoning with Domain-Specific and SAT-Based Systems**

    - We explore several properties of feature models and show how these properties can be used to develop efficient domain-specific algorithms to reason on a subset of feature models.
    - We also show how these domain-specific algorithms can be combined with existing SAT algorithms into hybrid solutions that can deliver improved performance for certain kinds of reasoning operations on feature models.
    - We empirically show that some of the domain-specific algorithms can indeed be much more attractive than a pure SAT-based solution. We expect that the insights provided can lead to many new domain-specific algorithms in the future.
    - We show empirically that SAT instances derived from a certain class of feature models represent "easy" problem instances for a SAT solver. Since we expect that most of realistic instances will yield much easier SAT problems we claim that we have improved the confidence on the general use of SAT solvers to handle feature models.

3. **Feature Model Benchmarks and Support Tools**

    - Currently, a major challenge to overcome in the field is the lack of publicly-available large real feature models. The vast majority of models available are small (100 features or less) and fit well in research papers but are rarely ideal for evaluating the scalability of feature model reasoning techniques. It is known that large models exist [7] but access to them is usually not granted by third parties. Hence, in this thesis we analyze several real feature models available in the literature in order to identify as many similarities as possible among them. Next, we build a benchmark tool that is capable of generating feature models that mirror as much as possible real models. The generated models are used in our empirical experiments to support scalability analysis. The models and the benchmark generation tool were made public as we want to encourage other researchers to take advantage of the infrastructure we have built. In fact, a research group is already using the benchmarks to perform empirical analysis on feature models using Alloy. In addition, two other research groups have already contacted us to download the benchmark tools and some of the models we have used in our experiments in order to compare their techniques to ours.

- We developed a comprehensive software library for manipulating feature models and various reasoning techniques including SAT solvers and BDDs. The API contains about 157 classes and 17,144 lines of code and is extensible in at least two aspects: first, it allows new reasoning techniques to be easily embedded and contrasted with existing techniques; second, it allows new BDD variable ordering heuristics to be developed and compared with existing ones, including the novel heuristics proposed in this thesis.

We claim that our research provides a significant step forward in the state-of-the-art by examining deeply many relevant properties of the feature modeling domain and the mechanics of SAT solvers and BDDs and the sensitive issues related to these techniques when applied in that domain. Specifically, we provide more accurate explanations for the space and/or time (in)tractability of these techniques in the feature modeling domain, and enhance the algorithmic performance of these techniques for reasoning on feature models.

Ultimately, we expect the ideas discussed in this thesis to raise awareness in our research field of the importance of taking domain knowledge into account for analyzing and enhancing automated techniques for feature model reasoning.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 provides the background information necessary for reading this thesis including the subjects of feature models, constraint satisfaction problems, constraint solvers, and binary decision diagrams. Chapter 3 reviews related work presented in the literature and classifies the feature model reasoning activities examined into four major areas based on the contexts where they are applied. Following, work on configuration systems and automated techniques for feature model reasoning is discussed and contrasted with our research. Chapter 4 explores several properties of feature models and, based on these properties, two new heuristics are proposed to order the variables of BDDs compiled from feature models. Chapter 5 examines further properties of the feature modeling domain as a basis for the development of new domain-specific algorithms for reasoning on a subset of feature models. The algorithms proposed are further integrated with SAT algorithms to form hybrid solutions that can be applied to the entire feature model. In addition, insights are provided that can help assessing the hardness of SAT instances derived from feature models. The results of empirical experiments to evaluate the ideas proposed in the thesis are reported in Chapter 6. First, the models that supported the experiments are presented including the real models gathered from the literature as well as those generated automatically. Following, experimental results are discussed to compare the quality and the scalability of the two BDD variable ordering heuristics proposed in Chapter 4 against existing state-of-the-art heuristics. In addition, the performance of the hybrid algorithms proposed in Chapter 5 is compared with that of pure SAT solutions. Finally, the results of empirical experiments to evaluate the hardness of feature model SAT instances are presented and discussed. Chapter 7 concludes the thesis and proposes future research directions.

1.5 Bibliographical Notes

The research ideas presented in this thesis were influenced by or extended previous publications of the thesis author. In the following, we list the most relevant related research papers.

- Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald D. Cowan. Efficient compilation techniques for large scale feature models. In the Proceedings of the Seventh International Conference on Generative Programming and Component Engineering (GPCE'08), pages 13-22, 2008 [66]. (Chapter 4)

- Marcilio Mendonca, Thiago Bartolomei, and Donald Cowan. Decision-making coordination in collaborative product configuration. In 23rd Annual ACM Symposium on Applied Computing. ACM, March 2008 [64]. (Chapter 5)

- Marcilio Mendonca and Donald Cowan. Decision-making coordination and efficient reasoning techniques for feature configuration. Science of Computer Programming Journal (Elsevier). Submitted to a special issue of the Journal as an extended version of the paper "Decision-Making Coordination in Collaborative Product Configuration" published at ACM SAC, 2008. (Chapters 5 and 6)

- Marcilio Mendonca, Toacy Oliveira, and Donald Cowan. Collaborative product configuration in software product lines formalization and dependency analysis. Journal of Software, 3(2):69-82, 2008 [65]. (Chapter 5).

- Marcilio Mendonca, Donald D. Cowan, and Toacy Cavalcante de Oliveira. A process-centric approach for coordinating product configuration decisions. In Proceedings of HICSS, 2007 [67]. (Chapter 5)

- Marcilio Mendonca, Krzysztof Czarnecki, Toacy Cavalcante de Oliveira, and Donald D. Cowan. Towards a framework for collaborative and coordinated product configuration. In OOPSLA Companion (Doctoral Symposium and Poster Session), pages 649-650 and 774-775, 2006 [68]. (Chapter 5)

Chapter 2

Background

In this chapter, we provide the background information necessary for reading this thesis including the subjects of feature models, constraint satisfaction problems, constraint solvers, and binary decision diagrams. We introduce feature models and show how they can be used as a basis for constructing unique and valid specifications for software product line systems. In addition, we discuss translation rules for converting a feature model into an equivalent propositional formula and how this translation allows the configuration of the feature model to be addressed as a constraint satisfaction problem. Finally, we present two powerful techniques used for decades to address many practical combinatorial problems, i.e., constraint solvers and binary decision diagrams, and argue that these techniques can also be applied effectively to reason on feature models and product configuration.

2.1 Feature Models and Configuration

Feature models [48, 26] are used in software product lines as a means to represent the similarities and differences of system families in terms of features. The term *feature* is referred to in the literature as *"any prominent and distinctive aspect or characteristic that is visible to various stakeholders"* [48]. For instance, modern Web search engine systems usually provide a "search by language" feature that allows users to search for Web pages written in a specific language such as English or Portuguese. During a process called *product configuration* features in the feature model are selected and arranged into product specifications to describe unique systems in the product line. Such specifications can be further used as input for programs called generators [26, 24] to automate the generation of those systems.

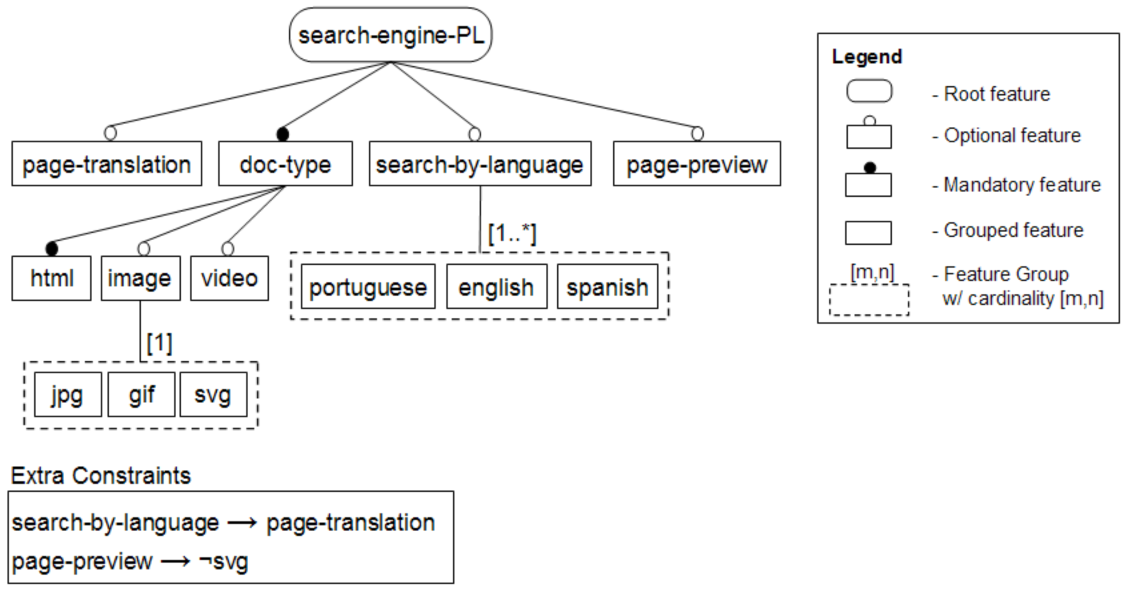


Figure 2.1: A feature model for a Web search engine product line

What makes feature models particularly appealing is the fact that they prevent the derivation of illegal product specifications. That is, the relations in the feature model constrain the way in which features can be combined into system specifications. For instance, the feature model enforces that mutually-exclusive features are never part of the same product. In this sense, a specification is said to be *invalid* or *illegal* if it violates any of the constraints in the feature model. Feature model constraints can be specified in two ways. First, the features in the product line can be arranged hierarchically in a structure called *feature tree*. The types of the features and their particular arrangement in the tree structure describes a set of feature relations. Second, additional relations can be attached to the feature model to enrich its expressiveness which we shall call *extra constraints*. The feature model is then the conjunction of the relations in the feature tree and the extra constraints.

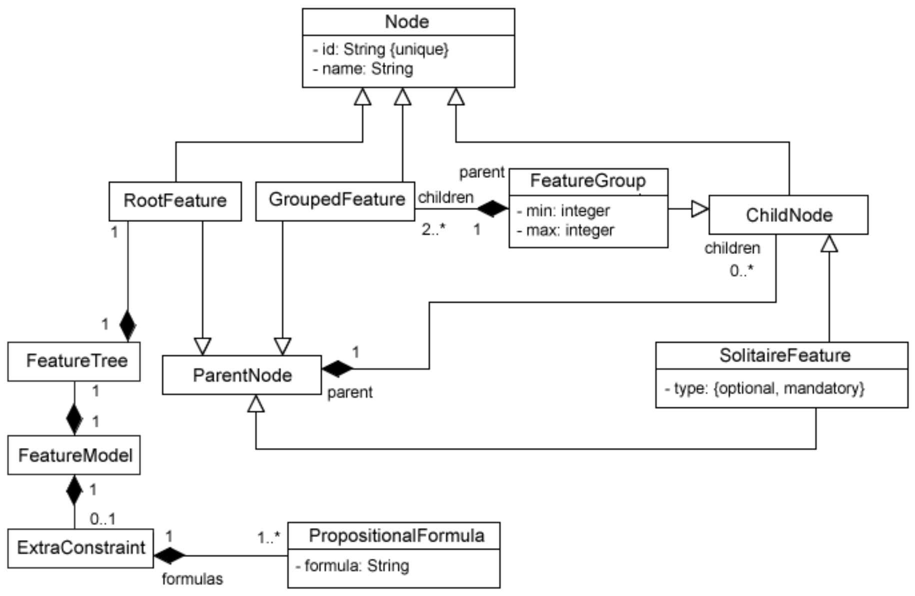
Figure 2.1 depicts a simple feature model for a Web search engine product line. The feature tree structure in the model is shown containing a hierarchy of labeled nodes, i.e., the features. Five types of features are possible: mandatory, optional, inclusive-OR, exclusive-OR and the root feature. The root feature is usually called the *concept* as it models the variabilities associated with a particular domain concept, in our case, a software product line. By convention, we always assume that the root feature is part of any legal system specification in the product line.

Optional features are illustrated in Figure 2.1 with an unfilled circle decorating the top of some labeled rectangles. For instance, features *page-translation*, *search-by-language*, *page-preview*, *image*, and *video* are optional. Optional features can only be part of a system specification if their parent features are also included in the specification. Therefore, if feature *doc-type* (parent feature) is part of a product specification (which coincidentally is always the case in the feature model) feature *video* (child optional feature) may alternatively be added to the specification.

Mandatory features are represented by labeled rectangles decorated with a filled circle on top. For instance, features *doc-type* and *html* are mandatory. Mandatory features must be included in a system specification whenever their parent feature is part of the specification. Therefore, feature *html* (child mandatory feature) must be part of any specification containing feature *doc-type* (parent feature). Similarly, feature *doc-type* must be part of any specification containing the root feature. Since the root feature is part of any valid specification, we conclude that feature *doc-type* and by transitivity feature *html* are part of any legal specification of the Web search engine product line.

Alternatively, features can be grouped together into *feature groups* to form cardinality-based relations (see dashed rectangles in Figure 2.1). Features that are part of such groups are called *grouped features* (e.g. *portuguese*, *english*, *spanish*, *jpg*, *gif* and *svg*). The cardinality [m,n] associated with a feature group indicates that at least m and at most n features in the group must be included in the product specification whenever the group's parent feature is included. When $m = n = 1$ an *exclusive-OR* group is specified that enforces that only one feature in the group can be selected (e.g. *jpg*, *gif* and *svg*). In this case, we generally omit parameter n in the cardinality (e.g. [1]). *Inclusive-OR* groups are defined by cardinality [1,*] enforcing that at least one feature in the group must be selected whenever the group's parent feature is selected. The "*" symbol is used to indicate that parameter n corresponds to the total number of features in the group (e.g., in the figure [1,*] is the same as [1,3]). We shall use the term *subfeature* to refer to the descendants of a given feature in the feature tree. For instance, features *jpg* and *video* are subfeatures of feature *doc-type*.

In addition to the feature tree, Figure 2.1 also depicts two extra constraints at the bottom rectangle labeled "Extra Constraints". Each constraint defines a relation between two or more features. For instance, relation (*page-preview* $\rightarrow \neg$*svg*) prevents features *page-preview* and *svg* from being part of the same valid specification. That is, the relation reads "the inclusion of feature *page-preview* implies the exclusion of feature *svg*". Similarly, relation (*search-by-language* $\rightarrow$ *page-translation*) requires that feature *page-translation* be always included in specifica-



```
-- Only inclusive-OR and exclusive-OR groups supported
context FeatureGroup inv:
  (min = 1) and (max = 1 or max = self.children->size or max = -1)
```

Figure 2.2: The meta-model for feature models

tions containing feature *search-by-language*. Notice that both extra constraints add new relations to the feature model not described in the feature tree.

As mentioned earlier, a system in a software product line can be specified as a combination of feature model features. Hence, specification S_1 ={*search-engine-PL*, *doc-type*, *html*} describes a unique Web search engine system for the feature model depicted in Figure 2.1. Such specification is valid as it does not violate any of the constraints in the model. Instead, specification S_2 ={*search-engine-PL*, *doc-type*, *html*, *search-by-language*, *page-preview*} is invalid since it violates a relation in the feature tree, i.e., the selection of parent feature *search-by-language* requires that at least one of its child grouped features, i.e., *portuguese*, *english* and *spanish*, be selected which is not the case. In addition, the selection of feature *search-by-language* requires the inclusion of feature *page-translation* according to extra constraint relation (*search-by-language*$\rightarrow$ *page-translation*) yet feature *page-translation* is not part of the specification. Hence, an alternative to fix specification S_2 is to add features *page-translation* and *portuguese* to it.

The feature model in Figure 2.1 is an example of a concrete model for a Web search engine product line. However, it is important that we describe more generally the class of feature models that are covered in this thesis. In the following, we use the notion of UML meta-models and OCL constraints for this purpose.

Figure 2.2 shows a meta-model for feature models, i.e., a meta specification that must be followed to build correct models. The $Node$ element in the meta-model indicates that each node in the feature tree must be uniquely identifiable (*id* attribute) and can optionally have a name (*name* attribute). The $RootFeature$ element represents the single root feature of the tree. Mandatory and optional features are represented by the $SolitaireFeature$ element. The *type* attribute is an enumeration that indicates whether the solitaire feature is optional or mandatory. The $GroupedFeature$ element represents features that are part of inclusive-or and exclusive-or groups. The $FeatureGroup$ element enforces that a set of grouped features are part of the same group. Attributes *min* and *max* refer to the minimum and maximum cardinality of the group, respectively. We only consider the cases of exclusive-or and inclusive-or groups as enforced by the OCL constraint in the bottom of the figure. For $max = -1$ we assume that max corresponds to the total number of features in the group. The relation between $GroupedFeature$ and $FeatureGroup$, labeled *parent*, enforces that only feature groups can be parent nodes of grouped features. The $ParentNode$ and $ChildNode$ elements are connected by relation *parent* that indicates that the root feature as well as grouped, mandatory and optional features, all descendants of $ParentNode$, can be parent nodes of $ChildNode$ elements such as feature groups, mandatory and optional features. The root node does not have a parent node. The $FeatureTree$ element represents a feature tree containing a single root node. The $FeatureModel$ element indicates that a feature model always have a single feature tree ($FeatureTree$) and optionally an extra constraint ($ExtraConstraint$). The extra constraint consists of one or more propositional formulas ($PropositionalFormula$) described textually through attribute $formula$. We introduce propositional formulas in the next section.

Notice that the Web search engine feature model in Figure 2.1 conforms to the meta-model just described. It has a feature tree rooted by feature *search-engine-PL* as well as mandatory, optional and grouped features arranged hierarchically. In addition, extra constraints are attached to the feature model to complement the feature tree relations.

2.2 Constraint Satisfaction

Constraint satisfaction [76] is a powerful framework in computer science that has been used for several decades to model and solve numerous practical combinatorial problems. Examples of traditional problems include scheduling, planning and configuration. More recently, other fields have been explored such as natural language processing (construction of efficient parsers), computer graphics (visual image interpretation), computational molecular biology (DNA sequencing), business applications (trading), and radio frequency planning.

A *constraint satisfaction problem* (CSP) consists of a set of variables, domains of values, and constraints that impose conditions that the variables must satisfy. A solution to the problem is an assignment to the variables that satisfies all the constraints. A classic example of a CSP is the popular number placement game Sudoku[1]. The goal is to fill a partially-filled 9x9 grid so that each column, each row, and each of the nine 3x3 boxes contains the digits from 1 to 9 only one time each. By encoding the game rules as a CSP it is possible to automate the computation of solutions and the creation of instances of the game.

Boolean Satisfiability (SAT) is a special case of constraint satisfaction problems in which all the variables in the problem are Boolean. For instance, the configuration of the Web search engine product line in Figure 2.1 can be represented as a SAT problem. Features are variables that can be either *true* (included in the product specification) or *false* (excluded from the product specification). The feature tree and the extra constraints define the constraints in the problem. A solution to the problem represents a valid product specification, i.e., a selection of features that does not violate any of the constraints in the feature model. In the following, we provide a more precise definition of Boolean satisfiability problems.

Definition 2.2.1 *A **Boolean satisfiability problem (SAT)** is a triple $\langle X, D, C\rangle$ where X is a set of variables over domain $D = \{0, 1\}$, and C is a set of constraints over X. Every constraint $c_i \in C$ restricts the combined values of its variables, denoted by $Vars(c_i)$. An assignment $A(S)$ is a set of tuples $\langle s_i, v_i\rangle$ such that $S \subseteq X$, $s_i \in S$, $v_i = 0$ or 1, and s_i appears at most once in $A(S)$. We say that $A(S)$ satisfies a constraint $c_i \in C$, if $Vars(c_i) \subseteq S$ and the assignments made to c_i's variables in $A(S)$ cause this constraint to evaluate to 1 (true). A solution to the SAT problem is an assignment $A(X)$ that satisfies all constraints in C in which case the problem is said to be **satisfiable**. Instead, if no solutions can be found the problem is **unsatisfiable**.*

A SAT problem can be encoded as a *a Boolean (or propositional) formula.* This formula is constructed by using Boolean variables and the logic operators $\vee$

[1] http://www.websudoku.com/

Table 2.1: Rules for translating feature model relations into Boolean formulas

Feature model relation	**Corresponding formula**
r is the root feature	r
p is parent of optional feature c	$c \rightarrow p$
p is parent of mandatory feature c	$c \leftrightarrow p$
p is parent of grouped features $g_1, \ldots, g_n$, and group cardinality is [1..*] (Inclusive-Or)	$p \leftrightarrow (g_1 \vee \ldots \vee g_n)$
p is parent of grouped features $g_1,\ldots,g_n$, and group cardinality is [1] (Exclusive-Or)	$p \leftrightarrow ((g_1 \wedge \neg g_2 \wedge \ldots \wedge \neg g_n) \vee$ $(g_2 \wedge \neg g_1 \wedge \ldots \wedge \neg g_n) \vee$ $\ldots \vee$ $(g_n \wedge \neg g_1 \wedge \ldots \wedge \neg g_{n-1}))$
Extra constraints	already propositional formulas

(or), $\wedge$ (and), $\rightarrow$ (implication), $\leftrightarrow$ (bi-implication), and $\neg$ (not). For example, $f = a \rightarrow (b \wedge c)$ is a Boolean formula. A possible solution for f is the assignment $(\langle a{=}1\rangle, \langle b{=}1\rangle, \langle c{=}1\rangle)$ as it causes f to evaluate to *true*. A Boolean formula is in *conjunctive normal form* (CNF) if it represents a conjunction of clauses in which a clause is a disjunction of literals and a literal is a variable or its negation. For instance, formula f can be converted into the following equivalent CNF formula containing four literals and two clauses: $(\neg a \vee b) \wedge (\neg a \vee c)$. CNF formulas are important in practice as there are well-known algorithms [35, 34] for solving SAT problems encoded as a CNF formula.

As mentioned earlier, the configuration of a feature model can be viewed as a SAT problem. This can be accomplished by translating the model into an equivalent Boolean formula [8, 32], i.e., features represent variables and the relations in the model represent the constraints of the SAT problem as shown in Table 2.1. In this case, a solution to the problem represents a valid configuration in the feature model. For instance, if the translation rules in Table 2.1 are applied to the Web search engine feature model in Figure 2.1 formula f shown in the next page is obtained. Formulas 1-10 in f represent the feature tree relations while formulas 11 and 12 represent the extra constraints in the feature model. Formula f is then a formal representation of the feature model which enables the use of SAT systems to reason on the feature model. For instance, a SAT system can be used to search for valid configurations in the model.

$f =$

(1) $(\textit{search-engine-PL}) \wedge$

(2) $(\textit{page-translation} \rightarrow \textit{search-engine-PL}) \wedge$

(3) $(\textit{doc-type} \leftrightarrow \textit{search-engine-PL}) \wedge$

(4) $(\textit{search-by-language} \rightarrow \textit{search-engine-PL}) \wedge$

(5) $(\textit{page-preview} \rightarrow \textit{search-engine-PL}) \wedge$

(6) $(\textit{html} \leftrightarrow \textit{doc-type}) \wedge$

(7) $(\textit{image} \rightarrow \textit{doc-type}) \wedge$

(8) $(\textit{video} \rightarrow \textit{doc-type}) \wedge$

(9) $(\textit{image} \leftrightarrow ((\textit{jpg} \wedge \neg \textit{gif} \wedge \neg \textit{svg}) \vee$
$(\textit{gif} \wedge \neg \textit{jpg} \wedge \neg \textit{svg}) \vee$
$(\textit{svg} \wedge \neg \textit{jpg} \wedge \neg \textit{gif}))) \wedge$

(10) $(\textit{search-by-language} \leftrightarrow (\textit{portuguese} \vee \textit{english} \vee \textit{spanish})) \wedge$

(11) $(\textit{search-by-language} \rightarrow \textit{page-translation}) \wedge$

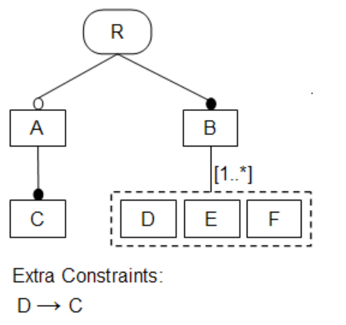
(12) $(\textit{page-preview} \rightarrow \neg \textit{svg})$

In the next sections we discuss two important techniques to address SAT problems and show how they can be used to reason on feature models and configuration.

2.3 Constraint Solvers

Constraint solvers are systems used to reason on constraint satisfaction problems. Different algorithimic techniques are applied by modern constraint solvers such as backtracking search, local search, and dynamic programming. We focus on backtracking search as it is currently the most important in practice and largely supported. Besides, backtracking search implements a *complete* algorithm, i.e., it guarantees that a solution will be found if one exists. This is particularly useful in the context of configuration in which enforcing the satisfiability of feature models is important.

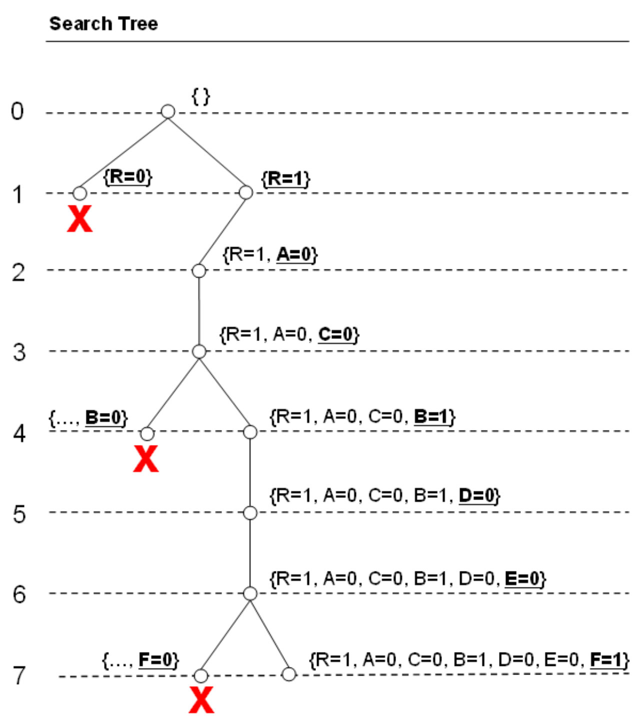
A backtracking search algorithm dynamically builds and traverses in depth-first order a hierarchical structure called *search tree* in an attempt to find a solution for the CSP problem. Figure 2.3(b) shows a search tree for the feature model in Figure 2.3(a). Alternatively, the constraints in the feature model are shown as a conjunction of 6 propositional formulas at the bottom of Figure 2.3(a). The dashed horizontal lines in Figure 2.3(b) represent the 8 levels of the search tree. Level 0 contains a single node that roots the tree. An initially-empty set of variable assignments is attached to the root (see the curly brackets at level 0 in the figure).



Feature model as formulas :

(1)	$(R) \wedge$
(2)	$(A \rightarrow R) \wedge$
(3)	$(C \leftrightarrow A) \wedge$
(4)	$(B \leftrightarrow R) \wedge$
(5)	$(B \leftrightarrow (D \vee E \vee F))$
(6)	$(D \rightarrow C)$

(a)



(b)

Figure 2.3: A feature model (graphical view and formulas)(a) and its search tree (b)

At level 1 the root node is expanded to add variable R to the set. Since R can assume different truth values, two new branches are created in the tree in a process known as *branching*. Each branch expands the initially-empty set at level 0 based on the two possible values for R, i.e., $\{R{=}0\}$ (*false*) and $\{R{=}1\}$ (*true*). This *instantiates* variable R. The same expand-and-branch process repeats for all other levels causing new variables to be instantiated and added to the partial assignments set (see underlined text at each level in Figure 2.3(b)). In this context, the nodes in the tree represent a partial assignment to the variables analyzed that may or may not lead to a solution.

Constraints are used to check whether or not a node leads to a solution. If the partial set of assignments at a node violates one or more constraints the node is considered a *dead-end* (marked with a capital X in Figure 2.3(b)) and its latest assignment is retracted. For instance, node $\{R{=}0\}$ at level 1 is a dead-end since it violates constraint (1) in Figure 2.3(a) that requires R to be *true*. If all paths examined at a given level lead to dead-ends the algorithm backtracks to the previous level and attempts to explore new branches. If level 0 is reached during backtracking and there are no more branches to explore the algorithm indicates that the problem is unsatisfiable and terminates. Instead, if the last level in the tree is expanded and no constraints are violated the algorithm has found a solution. For instance, a solution for the feature model in Figure 2.3(a) is found at level 7 of the search tree in Figure 2.3(b). Three dead-ends were found during the search at levels 1, 4 and 7 for variables R, B and F which violated constraints (1), (4) and (5), respectively.

A serious drawback of the search algorithm just introduced is that it can lead to "thrashing", i.e., to examining several branches of the search tree unproductively as they will never lead to a solution. This can cause severe deterioration in the performance of the algorithm especially when the branches involved contain a large number of variables. Not surprisingly, this has become a major issue in the history of backtracking-search algorithms. As a result, several techniques have been proposed to cope with this problem, in particular, means to enforce some degree of local consistency. For instance *arc-consistency* [55, 56] maintains a certain level of local consistency by examining constraints and (temporarily) removing values from the domain of some variables that would never be part of any solution considering the values already assigned to instantiated variables. For instance, if constraint (1) in Figure 2.3(a) is processed prior to starting the search process the value 0 (*false*) would have been removed from the domain of variable R as it clearly violates that constraint. This would have prevented the branching of R for value 0 ($\{R{=}0\}$). However, it is important to notice that the term "local" in local consistency emphasizes that the pruning of the variable domains does not necessarily remove all possible dead-ends from the search. Instead, it enforces that the values left in the domains satisfy some of the relevant constraints in the problem.

Modern constraint solvers usually offer more than one alternative of local consistency algorithms. One such algorithm that is particularly efficient for SAT problems is called *forward checking* [46, 60]. Forward checking implements a restricted form of arc-consistency by considering constraints with *exactly-one* uninstantiated variable to prune the domains of uninstantiated variables so that only values consistent with the current values of instantiated variables are left. This process is also known as *constraint propagation*.

Constraint propagation is a particularly effective in the context of SAT problems since removing a value from the domain of a Boolean variable corresponds to automatically instantiating the variable to the other truth value. Figure 2.4 depicts the search tree of Figure 2.3(b) but this time considering the application of forward checking to prune variable domains. A column named *propagations* illustrates the pruning at each branch of the tree. Prior to the search process an initial propaga-

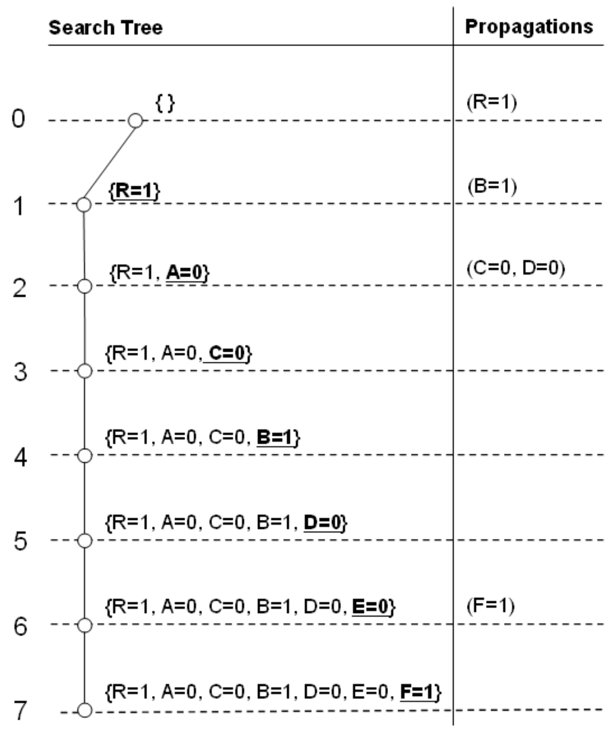


Figure 2.4: Search tree with constraint propagation for feature model in Figure 2.3(a)

tion is performed and only constraint (1) in Figure 2.3(a) is considered as it is the only constraint that has exactly-one uninstantiated variable. Variable R is then assigned 1 (see column propagation for level 0) as value 0 violates the constraint. At level 1, assignment $\{R=1\}$ causes constraints (2) and (4) to have exactly-one uninstantiated variable, i.e., A and B, respectively. No values are removed from the domain of variable A since the assignment of either truth value to this variable satisfies constraint (2). However, value 0 is removed from the domain of variable B since $R=1$ and the only possible value for B that does not violate constraint (4) is 1 (see column propagation for level 1). Constraint propagation is repeatedly applied in other levels to prune the domains of variables C and D (level 2) and variable F (level 6). Notice that dead-end branches $\{R=0\}$, $\{B=0\}$ and $\{F=0\}$ in Figure 2.3(b) were not generated in Figure 2.4 as the domains of these variables were pruned by constraint propagations in previous steps. As a consequence, 3 less nodes were visited in the search tree in Figure 2.4 when compared to the search tree in Figure 2.3(b) (8 and 11, respectively). In practice, the use of local consistency algorithms can cause a significant reduction on the number of visited nodes in the search tree and thus might have a tremendous impact on the performance of the search algorithm.

While general constraint solvers (CSP solvers) can be used to reason on SAT problems, more efficient tailored solutions are available. One such solution is a DPLL SAT solver [35, 34]. A DPLL solver also builds a search tree during the search procedure but rather uses a specialized propagation algorithm known as *unit propagation* to enforce local consistency. Unit propagation is equivalent to forward checking in the sense that it examines formulas with exactly-one uninstantiated variable.

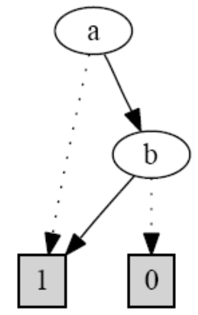
Despite the enormous advances that have been made to the SAT technology in the last decades, SAT problems are still “intractable” in worst-case scenarios. In fact, this is one of the topics addressed in this thesis. In particular, we examine domain-specific properties of Boolean formulas derived from feature models and how these properties can be used to improve the performance of SAT-based solutions in handling some traditionally “hard” operations (e.g. counting problem solutions). In addition, we study the “hardness” of feature model formulas, i.e., whether SAT instances generated from feature models can ever become “intractable”. These issues are discussed in Chapters 5 and 6.

2.4 Binary Decision Diagrams

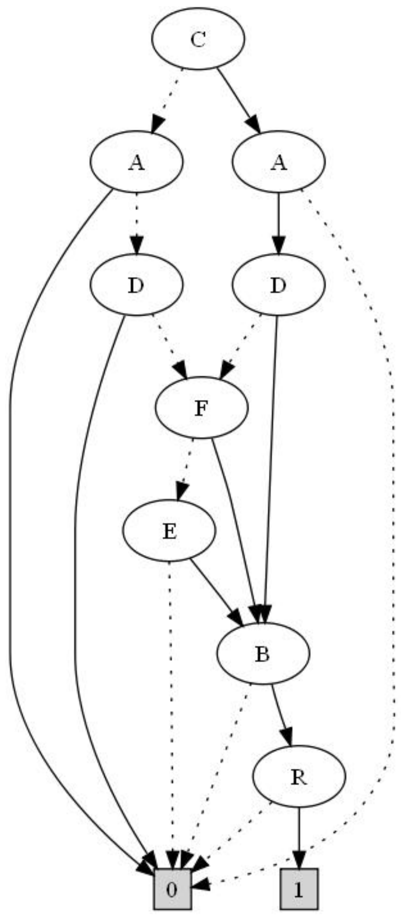
Binary decision diagrams (BDDs) [22, 4] are compact encodings for Boolean formulas that provide numerous efficient reasoning algorithms. During the last decades BDDs have been widely explored in research areas as logic synthesis, verification, configuration, constraint satisfaction and optimization. Currently, several BDD engines are freely available (e.g. JavaBDD [91], BuDDy [54], CUDD [80]).

In terms of data structure BDDs are directed acyclic graphs (DAGs) having exactly two *external nodes* representing constant functions 0 and 1, and multiple *internal nodes* labeled by variables (see Figure 2.5a). Each internal node has exactly two outgoing edges representing a decision based on an assignment to the node variable: the *low-edge* (a dotted line in the figures) represents the choice of *false*, while the *high-edge* (solid) represents the choice of *true*. A path from the root to an external node represents an assignment of values to variables. For example the rightmost path in Figure 2.5a represents a (non-satisfying) assignment (A=1, B=0). The paths terminating in the external node 1 (respectively 0) represent satisfying (respectively unsatisfying) assignments.

Figure 2.5b presents a BDD for the feature model and formulas in Figure 2.3(a) The BDD contains 9 internal nodes, 2 external nodes, and 5 satisfying paths, each



(a) $A < B$



(b) $C < A < D < F < E < B < R$

Figure 2.5: (a) A simple BDD and (b) a BDD for the model in Figure 2.3(a), and the corresponding order of their variables. Both BDDs are reduced and ordered.

representing one or more solutions. For instance, path $\{C{=}0,\ A{=}0,\ D{=}0,\ F{=}1,\ E{=}1,\ B{=}1,\ R{=}1\}$ is a satisfying solution for the BDD and therefore a legal configuration for the feature model in Figure 2.3(a). There is a total of 10 distinct solutions for the BDD as shown below:

1. $(C{=}0,A{=}0,D{=}0,F{=}0,E{=}1,B{=}1,R{=}1)$
2. $(C{=}0,A{=}0,D{=}0,F{=}1,E{=}0,B{=}1,R{=}1)$
3. $(C{=}0,A{=}0,D{=}0,F{=}1,E{=}1,B{=}1,R{=}1)$
4. $(C{=}1,A{=}1,D{=}0,F{=}0,E{=}1,B{=}1,R{=}1)$
5. $(C{=}1,A{=}1,D{=}0,F{=}1,E{=}0,B{=}1,R{=}1)$
6. $(C{=}1,A{=}1,D{=}0,F{=}1,E{=}1,B{=}1,R{=}1)$
7. $(C{=}1,A{=}1,D{=}1,F{=}0,E{=}0,B{=}1,R{=}1)$

8. (C=1,A=1,D=1,F=1,E=0,B=1,R=1)
9. (C=1,A=1,D=1,F=0,E=1,B=1,R=1)
10. (C=1,A=1,D=1,F=1,E=1,B=1,R=1)

A BDD is *ordered* if every top-down path in the DAG visits the variables in the same order. In a *reduced* BDD any two nodes differ either by labels or at least by one of their children (*uniqueness*), and no node has both edges pointing to the same child (*non-redundancy*). Notice that the BDDs in Figure 2.5 are both reduced and ordered. In particular, the variable order of the BDD in Figure 2.5a is $A < B$, i.e., variable A always precedes variable B in any top-down traversal of the BDD. Reduced-ordered BDDs (RO-BDDs) are the most used in practice and we shall use the term BDD to refer to them from now on.

The advantage of BDDs over SAT solvers is the superior performance of some BDD algorithms once the BDD structure is built. For instance, while a state-of-the-art SAT solver will almost certainly struggle to count the number of solutions in a SAT problem, a BDD can perform this operation very efficiently. In addition, equivalence checks of Boolean formulas can be performed in constant time using BDDs while this is generally NP-hard for SAT solvers. Finally, there are efficient BDD algorithms for calculating *valid domains*[2] [45], i.e., giving an assignment of some of the variables in the BDD it is possible to compute efficiently the available valid choices for each unassigned variable. This is especially important in interactive configuration in which while users make configuration decisions the configuration system updates in realtime the available configuration options. In other words, the system guides the user backtrack-free towards a valid configuration. Meanwhile, SAT propagation algorithms only enforce some degree of "local" consistency which does not guarantee backtrack-freeness. Instead, a SAT solver would have to perform several rounds of satisfiability checks to achieve the same results as BDDs yet with no guarantees of realtime responsiveness.

However, the high performance of BDD algorithms comes at a price. That is, the BDD structure represents a compilation of the entire combinatorial space of a SAT problem which causes building and maintaining such structure to be very costly and even impossible in some cases. For instance, in worst-case scenarios the size of the BDD is exponentially larger than the number of variables in the corresponding Boolean formula which translates to BDDs containing several millions of nodes.

[2]also referred to as *minimal domains*

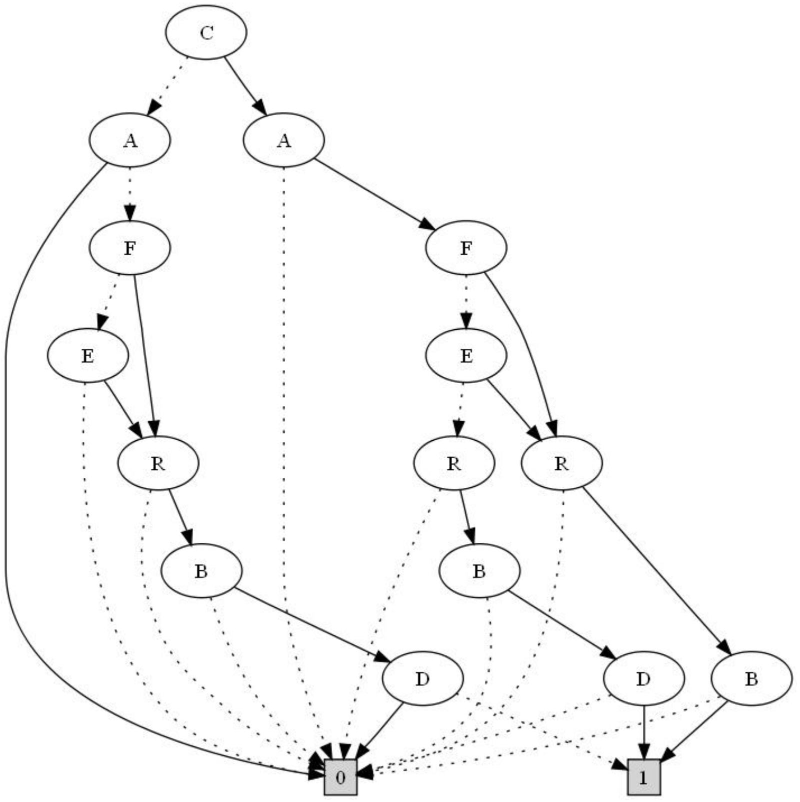


Figure 2.6: Another possible BDD for the feature model in Figure 2.3(a) with variable order ($C < A < F < E < R < B < D$). The BDD is 66% larger than the structure in Figure 2.5b.

Building such large BDDs can consistently cause "memory overflow" errors even in modern workstations. In this context, techniques to reduce the size of BDDs are critical not only as a means to improve the performance of BDD algorithms but also to allow the construction of such large BDDs using typical computer systems.

It is known that the size of a BDD can vary dramatically depending on the order specified for its variables. For an illustration of this problem, consider the BDDs in Figure 2.5b and in Figure 2.6 representing the same Boolean formula, i.e., the feature model in Figure 2.3(a), but with different variable orders. While the former BDD (Figure 2.5b) has only 9 nodes, the latter contains as many as 15 nodes, i.e., 66% more! Thus, it is of crucial importance to care about variable ordering when applying BDDs in practice.

A good variable order is one that keeps the size of the BDD as compact as possible, i.e., ideally close to the optimal. Unfortunately, finding an optimal variable order is an NP-hard problem [19, 62]. In the worst case, all possible variable combinations would have to be checked which has an exponential cost. For this reason the BDD variable ordering problem has been typically approached by heuristic algorithms. Heuristics exploit specifics of the problem domain in order to compute good orders efficiently. Many research communities applying BDDs developed such heuristics for their domain. In the next section, we discuss some of the most renowned heuristics developed so far.

Another important property of BDDs is *canonicity*. For any Boolean formula there is exactly one BDD b with a given variable order v. That is, the BDD structure is canonical for a formula given a fixed variable order. Canonicity is what makes the use of BDDs so attractive for checking the equivalence of formulas. If two formulas f_1 and f_2 produce the same BDD for the same variable order the formulas are equivalent. In many BDD libraries this can be performed by simply checking if two variables refer to the same BDD object in memory.

2.4.1 A Survey of BDD Variable Ordering Heuristics

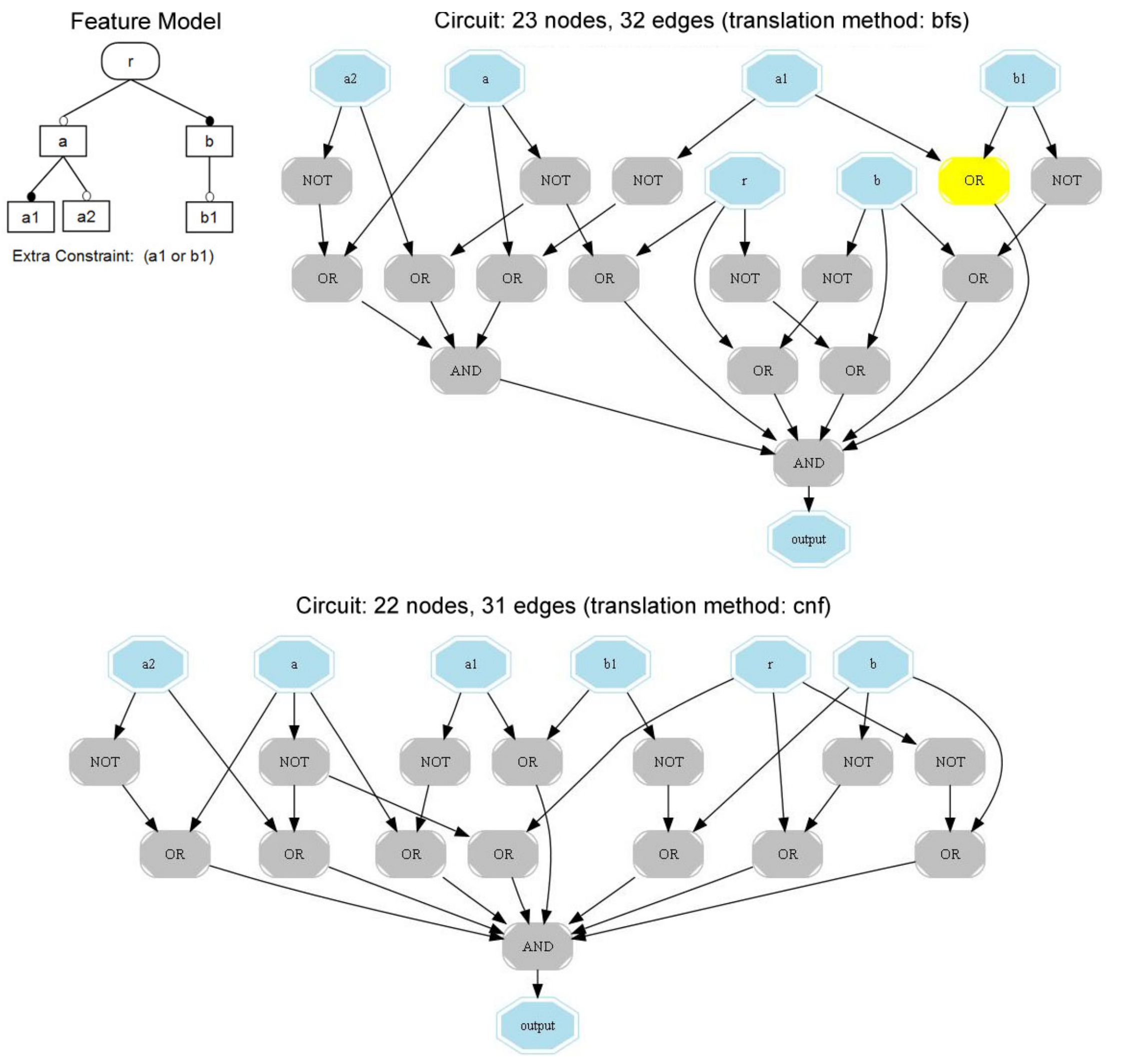
A pervasive goal of all the ordering heuristics is to place connected variables, i.e., variables that appear together in one or more Boolean formulas, close to each other in the ordering. This task is nontrivial. Dependencies between variables often interfere, i.e., optimizing the placement of a variable with respect to one dependency often decreases the quality of the ordering with respect to the others.

Variable ordering heuristics can be categorized into *dynamic* and *static*. Dynamic heuristics reorder the variables on-the-fly during the construction and manipulation of a BDD, usually exploiting garbage collection cycles of the underlying BDD system. Static heuristics compute a variable order off-line, which is then applied once to construct and analyze the BDD.

Static Heuristics

BDDs have been very successful in synthesis and analysis of digital circuits. Similar to a BDD, a circuit represents a Boolean function, and there exist direct translations between circuits and BDDs in either direction. Since the efficiency of verification strongly depends on the size of the BDD used, it is not surprising that the variable ordering problem has been deeply studied for the circuit domain.

Feature models can be easily translated to Boolean circuits which enables the use of existing ordering heuristics from that domain. However, since both feature models and circuits are Boolean formulas there exist many possible translations that



Feature Model
r
a
b
a1
a2
b1
Extra Constraint: (a1 or b1)
Circuit: 23 nodes, 32 edges (translation method: bfs)
a2
a
a1
b1
NOT
OR
r
AND
output
Circuit: 22 nodes, 31 edges (translation method: cnf)

Figure 2.7: A feature model and two equivalent Boolean circuits representing the model

produce different yet equivalent circuits for the same feature model. For instance, a particular translation approach here called "bfs" might consider generating a circuit based on a breadth-first traversal of the feature tree. That is, first a node labeled output is added to the circuit. Next, the features in the feature tree are visited in breadth-first traversal during which inputs and gates are generated. Gate structures for parent-child relations are constructed and recursively grouped into a hierarchy of AND gates. Finally, the extra constraints are taken into account to connect existing input nodes through new gates. Notice that if we change the traversal algorithm applied to the feature model from breadth-first to depth-first a different circuit is generated. Figure 2.7 shows a feature model in the left-top corner and a circuit graph for the model in the right top-corner assuming the breadth-first translation approach.

Yet, a possible translation might consider converting a feature model to a CNF formula and then generating the circuit from the formula. Let us call this translation method “cnf”. In this case, the circuit generated would typically exhibit a 5-layer topology, i.e., the inputs, the NOT gates (for negative literals, if applicable), the OR gates (one for each CNF clause), a single AND gate (to join the clauses), and the output node. NOT gates can be reused for inputs whenever necessary. A graph circuit for this translation approach is illustrated in the bottom of Figure 2.7. Notice that the two circuits have a distinct number of nodes and edges and the input nodes are ordered differently from left to right. We are not concerned with generating a functional circuit but rather a graph that can be processed by circuit heuristics.

We chose the “bfs” approach to generate the circuits addressed in this thesis as the approach seems to reveal more of the feature model hierarchy than its counterpart “cnf” that always produces a very flat circuit graph. In general, both approaches perform linearly for all feature model elements except for exclusive-OR groups, for which the number of gates is quadratic in the size of the group. In our evaluations, the translations produce circuits 3 to 10 times larger than the corresponding feature model.

Fujita’s Heuristic. Fujita-DFS [39] is a heuristic that traverses the circuit from the output to the inputs (which correspond to variables) in a depth-first search (DFS) order. During the traversal, inputs connected to two or more gates are placed first in the generated variable ordering in the hope that the remaining nodes in the circuit will form a tree-like structure for which a standard DFS produces good variable orderings. Since a circuit is a directed-acyclic graph (DAG) with a single output, nodes connected to many other nodes are removed from such a rooted DAG, the remaining structure approximates a tree. Fujita-DFS proved to generate good orderings for some circuit benchmarks, e.g. ICAS-85 [21].

Level Heuristic. The *level* heuristic [57] assigns the *depth level* to each circuit node, which is the length of the longest path from that node to the output. Subsequently, the inputs are sorted in decreasing order of levels to produce the final order. The level heuristic performs particularly well for multi-level circuits in which the outputs of a sub-network serve as inputs to the next subnetwork in the chain.

FORCE Heuristic. FORCE [2] is a domain-independent static heuristic for variable ordering. The heuristic is applied to a CNF formula and uses a measure called *span* to assess quality of placement for related variables. Given a pair of variables its span is defined to be their distance in a given variable ordering. The span of a clause is the maximum span of all pairs of variables occurring in the clause. Finally, the span of a CNF formula is the sum of spans of all its clauses.

FORCE begins with a random variable ordering and through successive steps attempts to minimize the formula span by moving variables near each other. At each iteration a new order is produced, which serves as input for the next iteration. It stops when the span value no longer decreases.

In order to apply FORCE, we implemented a simple CNF translation algorithm that traverses the feature tree in DFS and generates CNF clauses for each parent-child and feature group relation. For simplicity, we assumed the conjunction of the extra constraints was already in CNF form. The translation has the advantage that it places features related in a feature tree together in the same clauses, which makes FORCE try to put them close to each other in the ordering.

Sifting

Sifting [77, 62] is a popular domain-independent *dynamic* heuristic implemented in most BDD libraries. Unlike a static heuristic, sifting operates dynamically by trying to reduce the size of an already *existing* BDD on demand or on-the-fly; for example during garbage collection cycles. The main advantage of sifting is that it can enable the construction of BDDs that cannot be built with static heuristics.

Sifting is a local search algorithm. It swaps variables in the BDD if this leads to an improvement of the BDD size. Despite its merits, sifting has a serious drawback. The heuristic can be extremely slow in practice. In fact, we observed running times of over an hour for tasks that could be performed in a few minutes by good static heuristics. This is primarily caused by the fact that unlike FORCE a swap in a variable ordering requires a modification of the existing BDD to obey the new ordering.

Despite the advances made in BDD minimization research, space explosion of BDD structures remains an issue for certain kinds of large formulas. This is one of the topics addressed in this thesis. In particular, we explore several structural properties of Boolean formulas derived from feature models and how these properties can be used to develop efficient domain-specific variable ordering heuristics for reducing the size of BDDs for feature models. These properties and the proposed heuristics are discussed in detail in Chapter 4.

2.5 Summary

In this chapter, we introduced feature models and showed how they can be used to build valid system specifications in software product lines. In addition, we presented rules for translating feature models into propositional formulas and showed how this translation enabled the configuration of feature models to be addressed as a constraint satisfaction problem. Finally, we discussed two mature techniques used for several decades to reason on many practical combinatorial problems, i.e., constraint solvers and binary decision diagrams, and argued that these techniques can also be applied effectively in the feature modeling domain.

Chapter 3

Related Work

In this chapter, we extend the discussion on feature models by examining and classifying several feature model reasoning activities discussed in the literature into four major areas based on the context where these activities have been studied. Furthermore, we survey the state-of-the-art on automated techniques for feature model reasoning and discuss how proposed configuration systems and reasoning techniques relate to and benefit from our research.

3.1 Feature Model Reasoning

Several activities related to the manipulation of feature models in software product lines have been examined by previous research. For instance, after construction feature models might contain errors and thus need to be debugged. As well, feature models might need to be adjusted to follow the evolution of the corresponding product line. In addition, feature models must be correctly specialized and/or configured to derive consistent product specifications. Finally, several metrics can be applied to feature models to measure different aspects of the corresponding product line. In the next sections, we examine in detail these four major areas where feature model reasoning has been studied: debugging, refactoring, configuring, and measuring feature models.

3.1.1 Debugging Feature Models

We refer to *feature model debugging* as the process of verifying the correctness of a feature model, i.e., that the model accurately describes all product line features

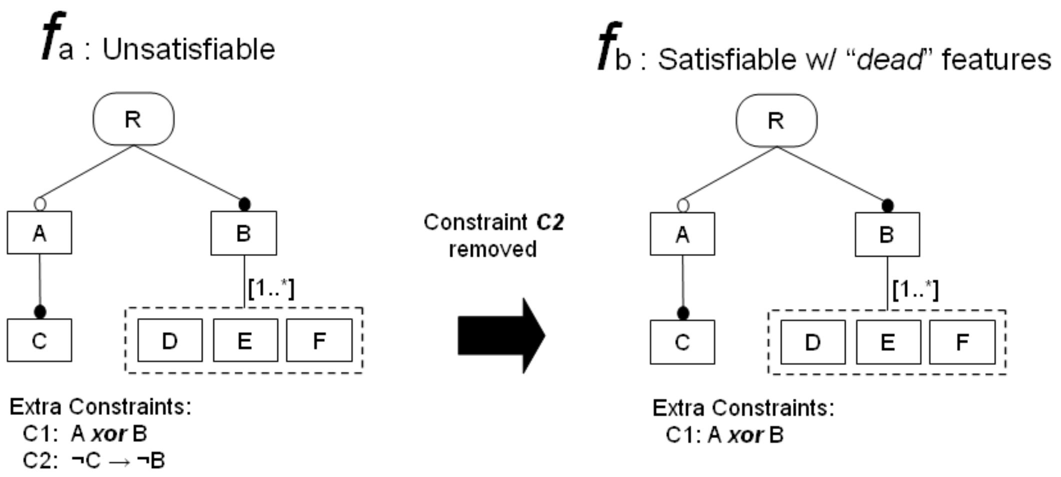


Figure 3.1: An unsatisfiable feature model f_a becomes satisfiable (f_b) after the removal of constraint C2.

and relations. At least two properties have been related to feature model debugging in the literature: satisfiability [7, 86, 58, 94] and the presence of "dead" features [10, 32].

A feature model is *satisfiable* if at least one valid product specification can be derived from the model, otherwise the model is *unsatisfiable* and thus incorrect. In practice, an unsatisfiable model corresponds to a product line from which no systems can be derived which is obviously a contradiction. Unsatisfiable models are usually a result of careless analysis or mistakes made by feature model designers especially when dealing with large models.

Figure 3.1 shows an unsatisfiable feature model f_a. The model contains 7 features and 2 extra constraints. Feature R roots the model and thus is always *true*. Hence, feature B must also always be *true* as it is mandatory and directly connected to the root. That is, features R and B must be *true* in all valid configurations of feature model f_a. Moreover, in order to satisfy constraint C1 (A xor B) feature A (and consequently its child feature C) must be *false* since feature B is *true*. If feature C is *false* so must be feature B to satisfy constraint C2 ($\neg C \rightarrow \neg B$), which is obviously a contradiction since we just stated that feature B must always be *true* in any valid configuration of the model. Hence, we conclude that model f_a is unsatisfiable since there is no value of B that satisfies the constraints in the model.

One possibility to make model f_a satisfiable is to eliminate one of the constraints that is causing the contradiction. For instance, feature model f_b in Figure 3.1, that is obtained by eliminating constraint C2 from model f_a, is satisfiable. For instance, a possible solution for f_b is the set of features $\{R, B, D\}$. This solution does not

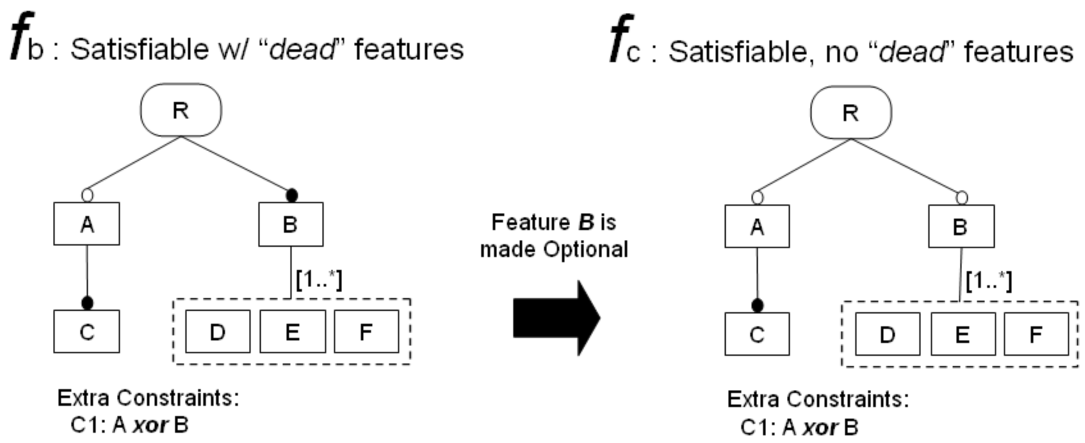

fb : Satisfiable w/ "dead" features
R
A
B
C
[1..*]
D
E
F
Extra Constraints:
C1: A xor B
Feature B is made Optional
fc : Satisfiable, no "dead" features
R
A
B
C
[1..*]
D
E
F
Extra Constraints:
C1: A xor B

Figure 3.2: Feature model f_b containing "dead" features A and C is fixed by making feature B optional in model f_c.

violate any of the constraints specified in model f_b giving that constraint C2 has been removed and thus feature B can now be safely assigned *true*.

A feature model is also incorrect when it contains "dead" features. A "dead" feature is a feature that can never be part of any legal configuration of the model. This corresponds to having a product line that provides features that can never be included in any derived system. Figure 3.2 shows feature model f_b again. As we know, this model is satisfiable, in fact, it has 7 valid configurations. However, in all these configurations features A and C are *false*, i.e., "dead" features. This is caused by constraint C1 that falsifies feature A (and thus feature C) whenever feature B is *true*, which is always the case. Once again, the model needs to be fixed by adjusting its relations. A possible alternative could be to make feature B optional as is shown in model f_c in Figure 3.2. This would allow features A and C to be *true* in some solutions of f_c. For instance, a valid configuration in f_c could be $\{R, A, C\}$.

3.1.2 Refactoring Feature Models

Software product lines evolve over time, for instance, to cope with market demands or to improve the corresponding software architecture. In one way or another, it is highly desirable that such changes do not compromise the compatibility of the product line with its former products otherwise the costs involved in maintaining different versions of the product line could quickly overcome the benefits. In this context, *feature model refactoring* [3, 40] is referred to as the process of performing semantic-preserving changes to the configurability of a product line. That is, a

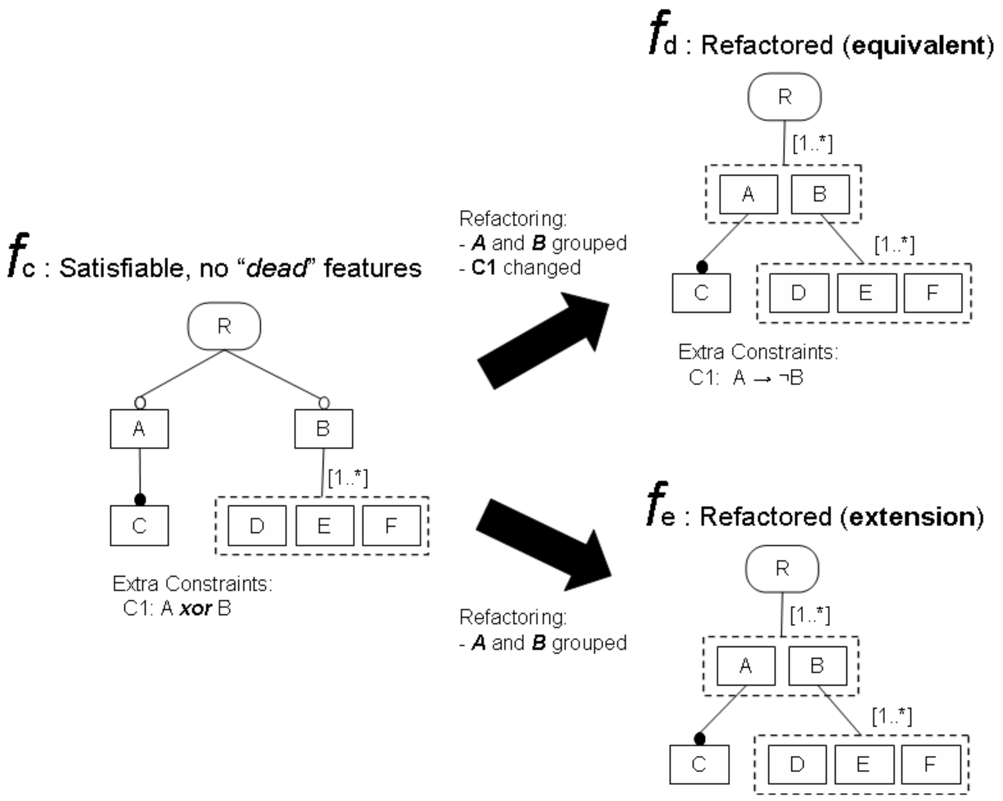


Figure 3.3: Feature model f_c is refactored yielding models f_d and f_e. While models f_e and f_c share the exact same configurations (equivalence), model f_e admits additional configurations not valid in f_c (extension).

refactoring maintains or increases the configurability of the product line.

When applied to a feature model a refactoring produces a new model such that all valid configurations in the original model are equally valid in the new model. In addition, the new model can have extra valid configurations. When two feature models f_1 and f_2 are such that they have exactly the same configurations, we say that f_1 is equivalent to f_2 or $f_1 \equiv f_2$. If otherwise, f_2 is a refactoring of f_1 but contains extra valid configurations, we say that f_2 is an extension of f_1 or $f_1 \subset f_2$. Notice that in this case (extension) the refactoring is not bi-directional, i.e., f_2 cannot be refactored back to f_1.

Figure 3.3 shows a satisfiable feature model f_c and two refactored models f_d and f_e. In f_c, features A and B are optional while in f_d and f_e they appear together as part of an inclusive-OR group. In addition, the *xor* constraint C1 in model f_c was modified to an implication in f_d. Since models f_c and f_d have exactly the same 8 valid configurations the refactoring produced an equivalent model ($f_c \equiv f_d$). Instead, constraint C1 was removed in f_e causing the resulting model to be an

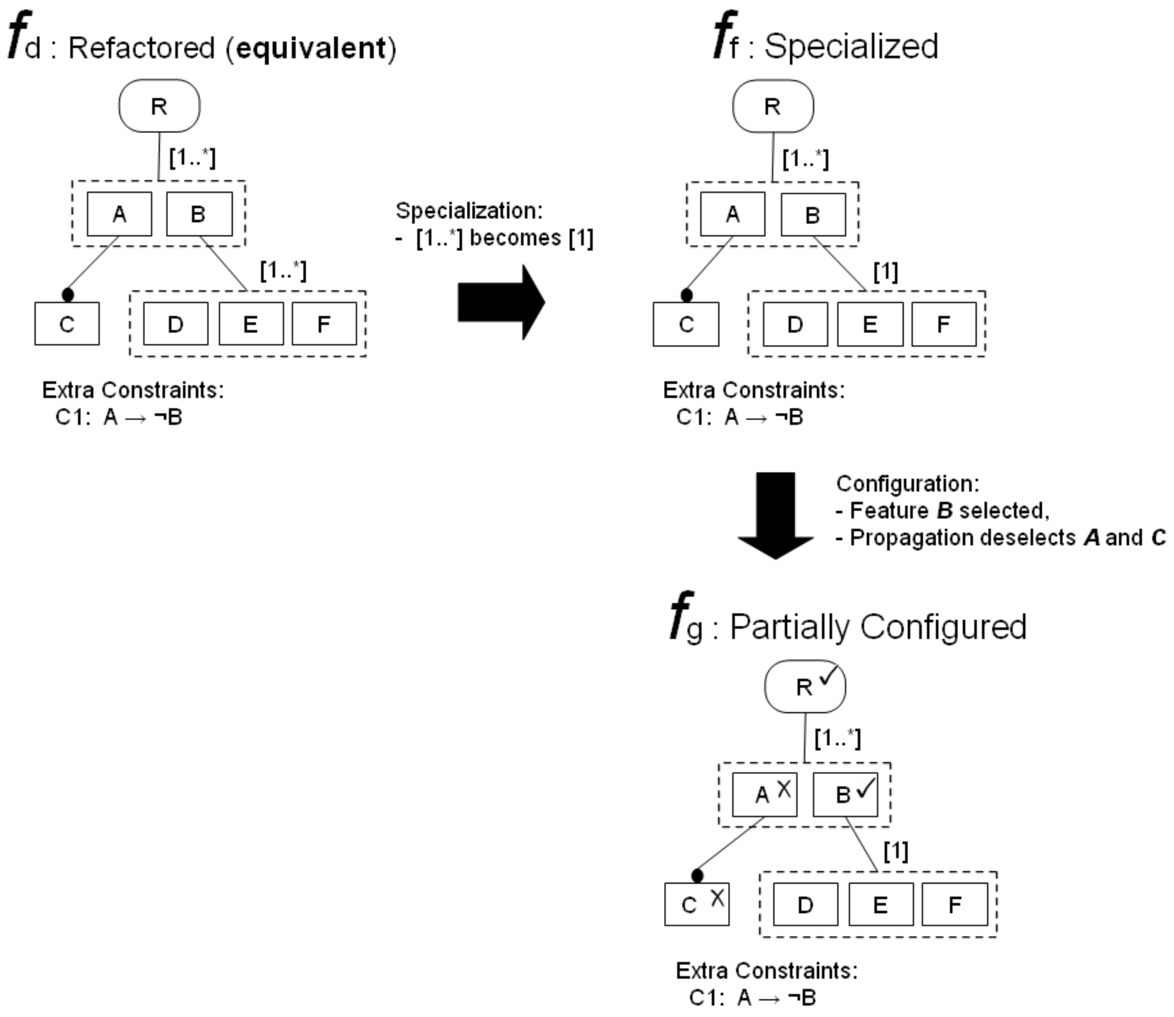


Figure 3.4: Feature model f_d is modified by replacing one of its inclusive-OR groups by an exclusive-OR relation yielding a more restrictive model f_f. Model f_f is further configured by having some of its features selected (e.g. features R and B) and deselected (e.g. features A and C) in model f_g.

extension of the original ($f_c \subset f_e$). For instance, the set $\{R, A, B, C, D\}$ is a solution in f_e but is invalid in f_c in which features A and B are mutually exclusive. In fact, model f_e contains as twice as many solutions as f_c (16 and 8, respectively).

3.1.3 Configuring Feature Models

As discussed in the previous chapter, *product configuration* is the process of selecting features in the feature model in order to build a valid specification for a product line system. In practice, configuration can be performed in different ways. In the following, we discuss some approaches to configuration mentioned in the literature.

In staged configuration [28] a feature model can be refined (or specialized) by continuously eliminating configuration choices in stages by either adding constraints to the feature model or by selecting/deselecting features. Each stage takes a feature model as input and yields an implicant feature model, i.e., a model that contains a subset of the configurations of the original model. Therefore, specialization can be viewed as the inverse process of feature model extension discussed in the previous section, i.e., if feature model f_2 is an extension of feature model f_1 ($f_1 \subset f_2$), then f_1 is a specialization of f_2. Figure 3.4 shows a feature model f_d that comprises 8 possible configurations. Feature model f_f represents a specialization of f_d in which the cardinality of the feature group containing features D, E and F is modified from an inclusive-Or ([1..*]) to an exclusive-Or ([1]) relation. This change makes model f_f more restrictive. In fact, f_f contains only 4 valid configurations that represent a subset of the valid configurations in f_d (configurations 1, 2, 4, and 8 below).

1. (R=1,A=0,C=0,B=1,D=0,E=0,F=1) - solution for f_d and f_f
2. (R=1,A=0,C=0,B=1,D=0,E=1,F=0) - solution for f_d and f_f
3. (R=1,A=0,C=0,B=1,D=0,E=1,F=1) - solution for f_d
4. (R=1,A=0,C=0,B=1,D=1,E=0,F=0) - solution for f_d and f_f
5. (R=1,A=0,C=0,B=1,D=1,E=1,F=0) - solution for f_d
6. (R=1,A=0,C=0,B=1,D=1,E=0,F=1) - solution for f_d
7. (R=1,A=0,C=0,B=1,D=1,E=1,F=1) - solution for f_d
8. (R=1,A=1,C=1,B=0,D=0,E=0,F=0) - solution for f_d and f_f

Figure 3.4 shows a feature model f_g that partially configures the specialized model f_f by selecting feature B. Feature R is also (and always) selected as it roots the model. However, notice that the selection of feature B was propagated in the model causing features A and C to be deselected (a check mark indicates selected features and an "X" indicates deselected features). Propagation was caused by constraint C1. Model f_g is only partially configured as one decision still remains in the model regarding which one of the features D, E or F is to be selected. Therefore, 3 possible configurations are still represented in the model. Notice that because f_g is a configured version of f_f then the relation ($f_g \subset f_f$) holds and by transitivity relation ($f_g \subset f_d$) also holds. In fact, an initial feature model f_1 can be refined in n stages by combining specialization and configuration techniques such that the resulting model f_n represents a product specification containing no more decisions. In this case, the configuration process can be represented by n successive refinement steps ($f_1 \supset f_2 \supset \ldots \supset f_n$).

A variation of feature model specialization is know as *interactive configuration*

[45]. Configuration systems supporting interactive configuration must enforce that any valid configuration is reachable (*completeness*), i.e., it must expose to its users all possible combinations of features. In addition, the users should never be required to review past decisions (*backtrack-freeness*). This is usually achieved by implementing algorithms to calculate *valid domains*[45]. These algorithms work by eliminating decisions in the feature model that are incompatible with the current set of decisions made by the users. Finally, the configuration system must respond to user requests in realtime (*realtime responsiveness*). This includes system requests to complete partial configurations or to list one or more valid configurations.

In collaborative configuration [64],[65], a feature model is configured concurrently by several users through multiple steps. A configuration plan is devised upfront to define the configuration spaces and the decisions each user will work on, the arrangement of the configuration steps (e.g. sequential or parallel), and the strategy to handle eventual decision conflicts. For instance, conflicts can be resolved by specifying user priorities. In this case, whenever a conflict occurs the decisions of the users with higher priorities will prevail. Alternatively, another strategy for conflict resolution could be to minimize the number of revisited decisions, i.e., the configuration system would find a valid configuration that best approximates the current set of decisions made [65, 92].

3.1.4 Measuring Feature Model Properties

Several metrics have been proposed to measure feature model properties. For instance, computing the *number of valid configurations* in the feature model allows reasoning on the flexibility and complexity of product lines [12, 86, 31]. In addition, this information can be used to measure how much the combinatorial space of configuration decisions has been reduced after several rounds of configurations steps in collaborative or interactive configuration. This is important as the target is to find a single valid configuration. Alternatively, this metric has been called *variation degree*[88] of feature models and has been related to issues that arise in developing, maintaining and evolving product lines.

The number of valid configuration has also been used as a parameter to compute other metrics on feature models. For instance, the *variability factor* of a feature model [12, 11] is a value between 0 and 1 computed as the ratio of the number of valid configurations to 2^n, where n is the number of features in the feature model. The smaller the ratio the more restrictive is the feature model and vice-versa. The

Table 3.1: Summary of feature model reasoning activities and operations

Summary of Feature Model Reasoning Activities and Operations	
Activity	**Operations (Technique)**
Debugging	- Checking satistiability of models (SAT) - Detecting if a given feature is "dead"(SAT) - Detecting "dead"features (SAT, BDD)
Refactoring	- Checking equivalence of feature models (SAT, BDD) - Checking extension of feature models (SAT, BDD)
Configuring	- Checking specialization of feature models (SAT, BDD) - Validating partial or full configurations (SAT) - Calculating valid domains (BDD) - Enumerating one or more valid configurations (SAT, BDD) - Resolving decision conflicts in collaborative configuration (SAT, BDD)
Measuring	- Counting valid configurations (BDD) - Computing variability factor (BDD) - Computing commonality of a feature (BDD)

relevance of computing the variability factor has been related to decision-making strategies for adopting the product line approach [12].

Finally, the *commonality of a feature* computes the number of valid configurations that includes a given feature f (f=*true*). This can be used to detect "dead" features or to plan the implementation order of features in the product line architecture [10]. This information can also be computed dynamically as decisions are made in interactive or collaborative configuration to support decision-making. For instance, a particular strategy for conflict resolution in collaborative configuration could alert decision makers about scenarios in which some "relevant" features would not be included in the product specification and make sure that this is indeed desirable.

3.1.5 BDDs, SAT Solvers and Feature Model Reasoning

Table 3.1 provides a summary of the feature model reasoning activities and respective operations discussed in the previous sections. There is no doubt that feature model designers and users must be properly assisted by automated tools in order to perform the activities listed in this table. For instance, debugging a feature model requires designers to check the satisfiability of the model and the presence of "dead" features which is virtually impossible without proper tool support, especially for large scale models. Also, interactive configuration strongly relies on the use of configuration systems capable of reacting to user decisions in a fraction of seconds in order to validate decisions and to prune the combinatorial space of configuration choices accordingly. In other cases, tool support is required to check the soundness of feature model transformations such as refactoring or specialization.

SAT solvers and BDDs can be very helpful in this context. Beside each operation description in Table 3.1 (enclosed in parentheses) we have indicated the technique that is most commonly suitable for performing the operation (SAT solver, BDD, or both) based on a literature review and on our own experience using these techniques. SAT solvers offer specialized algorithms to address the satisfiability problem and thus can handle efficiently operations that fit well in this context such as checking the satistiability of models, detecting if a given feature is "dead", and validating partial or full configurations. Although a BDD can also be used to perform these operations, building a BDD would require compiling the entire combinatorial space of a feature model which might delay the processing of operations that only require a partial analysis of the problem space such as those mentioned earlier. Instead, BDDs are typically suitable for handling computationally hard operations such as counting valid configurations, computing the variability factor of a feature model, computing the commonality of a feature, and calculating valid domains in interactive configurations. These operations usually require an exhaustive analysis of the problem combinatorial space. Yet in other cases, it is not clear which technique is most suitable as researchers have applied either technique successfully. For instance, past research has examined the use of SAT solvers [44, 18], BDDs [6] and even the combination of both techniques [71] for checking formula equivalence. Another important factor to consider for situations in which both techniques are applicable is the frequency in which the BDD structure will be used once it is built. Usually, if the BDD is used only once a SAT solver might be a better choice otherwise the cost of building the BDD structure might pay off. A typical example where a BDD is valuable is in interactive configuration in which the BDD is used several times for refining the configuration choices until a final configuration is reached.

The major issue with BDDs is not the efficiency of BDD algorithms for processing the operations just mentioned. In fact, this is the strength of this technique. Instead, the main issue is usually the inherent difficulty of building BDDs of tractable sizes for large scale models. Similarly, most of the operations for which SAT solvers are indicated in Table 3.1 rely on the efficiency of the solvers in performing satisfiability checks. Therefore, in our research we do not focus necessarily on particular operations provided by these systems but rather on space and/or time (in)tractability issues related to these systems. That is, we want to be able to build BDDs of tractable sizes for as large as possible feature models, and to examine the time (in)tractability of SAT solvers in handling SAT instances derived from feature models.

3.2 Reasoning Tools and Techniques

In this section, we survey the state-of-the-art in configuration systems and feature model reasoning techniques and show how research in this field relates to and benefits from our work.

3.2.1 Product Configurators

Feature Model Plug-in (FMP) [5] is a configuration system based on feature models. The tool provides users with a graphical interface for building and configuring feature models and is able to verify the correctness of generated product specifications. FMP also uses BDDs to keep track of the number of legal configurations in the feature model as configuration decisions are made. The XPath[1] language can be used to add additional relations to the feature model not contemplated in the feature tree. A novelty in the tool is its support for the staged configuration of feature models [30] as discussed in Section 3.1.3. In staged configuration a feature model is gradually specialized by adding constraints to the model or by selecting/deselecting features. FMP implements the specialization process by generating copies of the feature model each time a specialization step is performed. The tool enforces that the specialized models are indeed a specialization of the original models, i.e., the valid configurations in the specialized models are a subset of the valid configurations in the original models

XFeature[2] [75] is a feature modeling tool that relies heavily on the use of the XML technology to model product families and instantiate product specifications. Feature models are specified in XML and validated using XML schemas. While standard constraints such as *requires* and *excludes* can be written in XML, arbitrary constraints can be specified using XPath expressions. Configurations are also

[1]See http://http://www.w3.org/TR/xpath

[2]See http://www.pnp-software.com/XFeature/

described in XML and validated by automatically-generated XML schemas. XSLT, an XML transformation language, is used to check whether a given configuration conforms to the constraints in the model. The tool is mostly targeted to XML users and hence provides a solid embeddable architecture.

RequiLine[3] [89] is a requirements engineering tool that integrates requirements and feature modeling to produce product specifications. Feature models as well as standard relations such as *requires* and *excludes* can be constructed by using user interfaces. Products can be configured manually or by following wizards that guide the users through the configuration process. The tool offers a consistency checker to verify the correctness of feature models and the validity of configurations. Feature models are checked in a two-phase process. First, the feature graph is analyzed and validated. Following this step, each standard constraint is individually checked against the others. The strength of the tool is the integration between requirement analysis and feature modeling.

Gears[4] [50] is a commercial software product line engineering tool and framework that incorporates a product configurator. The rationale of the tool is to support the notion of *software mass customization*, i.e., the means of efficiently producing and maintaining a family of similar software products. Gears offers a development environment in which software assets (e.g. source code, UML models, script files) are developed and maintained. Moreover, the environment supports the construction of feature models and descriptions that map features to software assets. Configurations of the feature model, known as *feature profiles*, serves as input for the product configurator to generate products automatically. The tool provides a proprietary solution to describe constraints and validate feature models and configurations.

The Pure::Variant family[5] [74, 15] is a set of commercial tools that cover the entire life cycle of product line development, i.e., analysis, design, and implementation of the product line as well as production, test, use and maintenance of products. Feature models are used to capture and manage product variability and serve as a guide for product configuration. Family models describe the individual software components of the product line and their dependencies with features in the feature model. A configuration, called *feature selection*, serves as input to a generator that automates the generation of product line members. Pure::Variant uses Prolog to describe constraints and to check the consistency of feature models and validity of

[3]See http://www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline/

[4]See http://www.biglever.com/solution/product.html

[5]See http://www.pure-systems.com/

product configurations.

Other configuration tools supporting the notion of feature models are available including CaptainFeature [9], AmiEddi [59], DecisionKing [36] and VarMod [72].

There is no doubt that the development of configuration systems represent important research contributions in the field of software product lines. In this thesis we are concerned with the suitability of the techniques embedded in these tools for handling large feature models. Unfortunately, this is often missed in most research. That is, related publications do not provide enough details regarding the scalability of the techniques used as the size of the models increases, the amount of space required for building underlying structures, and the efficiency of the operations supported. Yet, these are crucial practical issues. For instance, the FMP tool uses BDDs to count the number of valid configurations in the feature model but it is not known how the tool orders BDD variables, a fundamental issue related to the BDD technique. As a consequence, we do not know how well the tool will scale given that BDDs for large models can become unfeasibly large. Moreover, it is known that the performance of XML tools such as XSLT and XPath degrades substantially for large XML models yet the XFeature tool relies heavily on such technologies. In other cases, configuration systems rely on SAT solvers to support the configuration of feature models yet it is known that satisfiability is an NP-complete problem. In this context, it is relevant to know whether feature model SAT instances can ever become intractable.

Ultimately, we argue that it is extremely hard to make any assumptions regarding the suitability of many current configuration systems in handling large models giving that critical issues related to the techniques used by these systems have not been addressed properly.

In our research, we provide a deep analysis of two powerful techniques for supporting feature-based configuration, i.e., BDDs and SAT solvers. We examine properties of the feature modeling domain to develop efficient heuristics for minimizing the size of BDDs and to build hybrid solutions that can improve the performance of SAT solvers for certain operations. Also, we evaluate empirically all the techniques and algorithms examined including their scalability. Current configuration systems benefit directly from our research. For instance, BDD-based feature model configuration tools can incorporate the new heuristics for BDD variable ordering and benefit from reduced BDD sizes which ultimately allows larger feature models to be processed. Also, these tools can embed some of the hybrid algorithms proposed in our research to improve the efficiency of some operations. In addition, empirical evaluations provided in our research increase the level of confidence of using SAT-based configuration systems giving that the performance of these systems have been empirically evaluated and limits were determined.

3.2.2 Reasoning Techniques

Researchers have explored several alternatives to represent feature models rigorously. This gave rise to a number of techniques that capitalized on the strengths of formal languages and related tools to improve automated support for reasoning on feature models. In the following, we survey some of the most relevant works in this subject.

The interest in the connection between feature model and logics has grown substantially over the recent years as demonstrated by the increasing number of research papers covering this subject [58, 8, 32, 12, 66]. As a result, rules were introduced for translating feature models to propositional formulas and vice-versa, and the advantages of such translation were discussed. In fact, by converting a feature model to an equivalent propositional formula it is possible to use efficient off-the-shelf tools such as SAT solvers and BDDs to reason on feature models as we discussed in Chapter 2. This is especially important given the maturity of SAT and BDD technologies and the successful application of these techniques to solve many practical combinatorial problems. In feature-based configuration, SAT solvers can be applied to check the satisfiability of a feature model, to verify whether a given feature is "dead", to enumerate one or more valid configurations, and to check the validity of configurations. Meanwhile, BDDs are very useful in counting the number of valid configurations, checking the equivalence of feature models, and computing valid domains for uninstantiated variables. The suitability of these two particular techniques for reasoning on large feature models is one of the major subjects of this thesis.

Another publication [12] considered feature models containing non-Boolean variables, sometimes called *extended feature models*, which required the use of more general solutions such as constraint satisfaction solvers (as opposed to specialized solutions such as SAT solvers) to support automated reasoning on feature models. The paper provided a notation for extended feature models along with definitions mapping feature model elements to the corresponding abstractions in the constraint satisfaction framework. Support for various reasoning operations were discussed such as counting the number of configurations, checking satisfiability of models, verifying the validity of configurations, and applying metrics for measuring the variability of models and the commonality of features. In our work, we do not address extended feature models but rather models consisting only of Boolean variables and complying to the meta-model described in Figure 2.2 in Section 2.1. At the same time, we go much deeper than previous work in examining SAT algorithms and how well these algorithms handle formulas derived from feature models. For instance, in Chapter 6 we draw a correlation between hardness and a phenomenon called phase transition that clarifies some facts about the hardness of feature model SAT instances.

The use of BDDs to reason on feature models has been tackled by some approaches such as the FMP configuration system [5]. As discussed earlier, the FMP uses BDDs to count the number of legal configurations in a feature model during product configuration. As configuration decisions are made the number of legal configurations decreases and is dynamically recomputed in the BDD. As we showed in Section 2.4 BDDs are very sensitive to the order of their variables and a bad order can lead to BDDs that are exponentially larger than the corresponding feature models. Unfortunately, finding an optimal order is NP-hard and the problem is typically addressed by using heuristics. While it is very important that efficient heuristics are developed for minimizing the size of BDDs representing feature models this is still a mostly unexplored area of research. In fact, the proposal of such kind of heuristics and their validation through empirical experiments represents one of the relevant contributions of this thesis.

FAMA [85, 33] is an approach to feature model reasoning that combines three logic-based reasoning techniques: SAT solvers, constraint solvers, and BDDs. The work is perhaps the first to combine different alternatives explicitly in the same suite. The approach is supported by an extensible tool called FAMA FW[6] that allows the production and analysis of extended feature models, i.e., feature models containing Boolean and non-Boolean variables. At the moment of this writing, FAMA FW supports four reasoning operations: satisfiability checks of feature models, counting and listing configurations, and calculating the "commonality of a feature", i.e., the number of legal configurations containing a particular feature. Based on user's requests the tool automatically chooses the most suitable technique to process the request. For instance, a BDD is selected whenever the user wants to known the number of legal configurations in the feature model. Unfortunately, the work does not elaborate much on the strategy applied for dynamically selecting the reasoning techniques. For instance, once a BDD is built it should clearly be the

[6]See http://www.isa.us.es/fama

preferred choice for handling all four reasoning operations available in the tool given the superior performance of BDD algorithms when compared to those of constraint solvers. Furthermore, the tool does not discuss other sensitive issues related to the techniques supported, for instance, how BDD variables are ordered and the quality of the orderings, or how variables/values are ordered for SAT solvers and what are the practical impacts of these optimizations. Instead, our research addresses these issues explicitly by examining the space and/or time (in)tractability of SAT solvers and BDDs for reasoning on feature models.

Recently, some techniques to diagnose and fix configuration errors in feature models have been discussed [92]. The techniques considered translating a feature configuration into a respective constraint satisfaction problem and using a constraint solver to suggest corrections to the original incorrect model. The constraint solver is given a set of rules (feature relations) and is asked to generate a series of possible fixes for the incorrect configuration. Optimal and bounded strategies are proposed to find the optimal and an approximate fix, respectively, out of the many possible fixes proposed by the solver. Results reported indicate that for optimal cases feature models with up to 2,000 feature can be fixed in an average time of 7.5 min. For the unbounded strategy, larger models with up to 5,000 features can be handled in about 1 min of processing. The work recognizes that finding real feature models of large sizes (hundreds or thousands of features) has been a challenge for researchers in the field and hence generated models are commonly used to support empirical experiments. A nice contribution of the work is the numbers provided regarding the size of the models used and the required processing times. This is certainly a major omission of most of the relevant works in the field today. Despite, the work does not comment much on how supporting models were generated and yet this can impact the results. First, it is very important that generated models embed as much as possible observable properties of real models, otherwise few assumptions can be made regarding the practical suitability of the studied techniques. Second, it is known that the hardness of constraint problems can be strongly influenced by factors such as the clause density, i.e., the ratio of the number of variables to the number of clauses (or formulas) in the problem. In fact, thresholds for certain class of formulas have been determined for which the problem becomes extremely hard (and ultimately infeasible) for constraint solvers. Therefore, we might expect that at a certain threshold, feature model formulas might derive complex or infeasible constraint problems. It is important to consider such factors when running and reporting results of empirical experiments. In our work, we conduct a careful examination of several real feature models and identify some relevant properties that can be borrowed to generate models of arbitrary sizes. In addition, we discuss the hardness of feature model formulas based on the identification of threshold values for which corresponding SAT problems should theoretically become hard to solve. Experiments are performed to relate problem hardness to the thresholds.

Feature model refactorings are transformations applied to feature models that preserve or improve the configurability of the models [3]. Currently, a catalog of sound refactorings exist [3]. The catalog encompasses a set of uni- and bi-directional refactorings that can help feature model designers to refactor feature models safely. Each entry in the catalog is formally verified using a theorem proving system called PVS[7]. The actual refactoring process is implemented using a template-matching technique in which the elements of a template representing the state of the feature model "before" the refactoring is matched to features in the feature model. Once the matching is performed the feature model can be safely refactored by using another template describing the output of the transformation "after" the refactoring. In another paper [41] the authors proposed a safe procedure to extend the catalog of refactorings based on algebraic laws. The laws are useful in constructing a formal proof of the soundness of new refactorings proposed in the catalog. That is, by successively applying the laws in a certain order it is possible to prove that a given output model is indeed a derivation of an initial input model. No additional knowledge on theorem proving systems is required. The PVS system is used to prove a series of theorems on algebraic laws. The techniques exploited in this thesis can be used directly to verify the soundness of feature model refactorings. For instance, as discussed in Section 2.4 BDDs can compute equivalence checking in constant time. As well, SAT solvers have proven useful for performing equivalence checks of formulas in certain domains [43].

Alloy[8] is a formal language based on first-order logic that has been considered as a lightweight alternative for complex theorem proving system for feature model reasoning [40]. Alloy comprises a specification language and an analyzer that supports automated reasoning on those specifications. The analyzer usually relies on external tools such as SAT solvers for improved performance. For instance, version 4 of the analyzer uses the SAT4J solver[9] by default (the same solver used in our experiments in Chapter Section 6). Different theories for feature models in Alloy are available. For instance, in [40] two theories were proposed to address the cases of general (G-theory) and specific (R-theory) feature model reasoning scenarios. This

[7]http://pvs.csl.sri.com/

[8]See http://alloy.mit.edu

[9]See http://www.sat4j.org/

enables the use of the Alloy analyzer to automate various feature model reasoning activities such as verifying the validity of configurations and the satisfiability of feature models, counting and enumerating configurations, and verifying the equivalence of feature models. The work also provides some performance numbers for the Alloy analyzer. It was reported running times of up to 9 minutes (about 4 minutes for the best case) to validate some refactorings for a feature model containing 300 features using version 3 of the Alloy analyzer. Using another more-efficient Alloy encoding and version 4 of the Alloy analyzer the authors have reported improved performance results (seconds instead of minutes for performing analysis) for models containing up to 10,000 features. We believe that the insights provided in this thesis can significantly increase the level of confidence regarding the use of Alloy to reason on large feature models considering that Alloy relies on SAT solvers, one of the techniques studied in our research, to perform automated analysis.

Research has also explored the use of alternative languages to reason on feature models including OCL [81], Prolog [17], OWL-DL [90], and Z [83].

3.3 Summary

In this chapter, we provided a compilation of the most relevant feature model reasoning activities reported in the literature and classified these operations into four major areas: debugging, refactoring, configuring and measuring feature models. In addition, we discussed the needs of improved automated support for feature model reasoning and surveyed existing tools and techniques aiming at this direction. We also commented on how current research relates to and benefit from our research.

Chapter 4

Reasoning with Binary Decision Diagrams

As we argued in Chapter 2 (Section 2.4), binary decision diagrams (BDDs) are a powerful mature technique that has been used successfully for several decades to reason on many practical combinatorial problems. In our work, we are interested in the application of the BDD technique to improve automated support for reasoning on feature models and product configuration. By compiling the entire combinatorial space of the configuration problem, BDDs can be very effective in counting the number of valid configurations, checking the equivalence of feature models, and supporting interactive configuration. Some research in the field of software product lines already tackled BDDs by proposing the incorporation of this technique into configuration systems [5] and reasoning tools for feature models [33].

However, it is well known that the size of the BDD structure can grow exponentially in the size of the input, i.e., the size of the feature model in our case, depending on the order specified for its variables. A bad ordering can lead to very large BDDs that cannot be built using a typical computer system. Finding an optimal order is an NP-hard problem [19, 62]. For this reason the BDD variable ordering problem has been typically approached by heuristics. Yet, the study of those heuristics for the feature modeling domain is still highly unexplored.

In this chapter, we expand our initial investigations [66] in the development of heuristics for ordering BDD variables in the feature modeling domain. We explore several structural properties of feature models and discuss how they can be used to produce high quality orders, i.e., orders that reduce as much as possible the size of the BDD. The novel heuristics developed are further evaluated by contrasting them with existing general and domain-specific heuristics through empirical experiments in Chapter 6.

4.1 Preliminaries

The following definitions are used throughout this and the next chapters.

Definition 4.1.1 *The extra constraints representativeness (ECR) of a feature model is the ratio of the number of variables in the extra constraints (repeated variables counted once) to the number of variables (features) in the feature tree.*

The ECR for the feature model in Figure 2.1 equals $\frac{4}{14} \simeq 0.28$.

Definition 4.1.2 *For features f_1, ..., f_n their* lowest common ancestor*, written $LCA(f_1, \ldots, f_n)$, is their shared ancestor that is located farthest from the root (where a feature is an ascendant of itself).*

For instance, LCA(*html*, *video*) $=$ *doc-type* and LCA(*jpg*, *spanish*) $=$ *search-engine-PL* (see Figure 2.1).

Definition 4.1.3 *Given $f = LCA(f_1, \ldots, f_n)$, the* roots *of features f_1, ...,f_n, written Roots($f_1, \ldots, f_n$), is either set $\{f\}$, if f is parent of f_1, ...,f_n, or the set containing the children of f that are ancestors of f_1, ..., f_n, otherwise.*

For instance, Roots(*jpg*, *spanish*) $=$ {*doc-type*, *search-by-language*}, since features *doc-type* and *search-by-language* are children of LCA(*jpg*, *spanish*) and root the subtrees containing features *jpg* and *spanish*, respectively (see Figure 2.1).

4.2 Exploring Structural Properties of Feature Models for Improved BDD Minimization

Many heuristics adopt the rationale of identifying and shortening the distance of dependent variables as a means to produce good variable orders. For instance, in the Level heuristic connected variables share the same level in the circuit. Fujita's heuristic uses a DFS traversal to identify connected variables in a circuit. As we mentioned before, span is the measure used by FORCE to approximate connected variables in a CNF formula. Based on this observation, we characterize the problem of ordering BDD variables in our domain as the problem of identifying related variables in feature models and producing variable orders that minimize the relative distance of such variables. What makes the problem particularly challenging is the fact that the relations in the extra constraints usually connect independent branches in the feature tree. This causes good orders for the feature tree to be extremely inefficient for the extra constraints, and vice-versa. In addition, the larger the ECR (see Definition 4.1.1) of a feature model the harder it is to find a good order that suits both the feature tree and the extra constraints.

One way of obtaining an ordering heuristic is to compile a feature model into an intermediate representation such as a CNF formula or a circuit and use available heuristics to process the ordering. However, this approach would completely ignore the domain knowledge. For instance, the variables in the feature tree are arranged hierarchically in a tree, for which simple traversals produce good orders. At the same time, as will be seen later, such arrangements are obscured in a CNF or circuit representation, which prevents the respective heuristics from exploiting them.

In the following, we consider factors that influence the development of new heuristics for variable ordering in the feature modeling domain. These considerations are then exploited in the next section when we propose such heuristics.

4.2.1 Good Orderings For The Feature Tree Are Usually Effective For The Feature Model

The feature tree defines the variables in the feature model and specifies most of its relations. From our experience in examining several feature models in the literature, we noticed that feature trees are frequently orders of magnitude larger than the extra constraints in terms of number of relations. This suggests that ordering heuristics should primarily focus on the relations in the feature tree to produce orderings. The observation is that good orderings for the feature tree are usually effective for the entire feature model especially for models with low ECR.

4.2.2 Mandatory Features Disturb The Analysis

Feature models allow the specification of mandatory features which might improve system family documentation but play no role in variability analysis. That is,

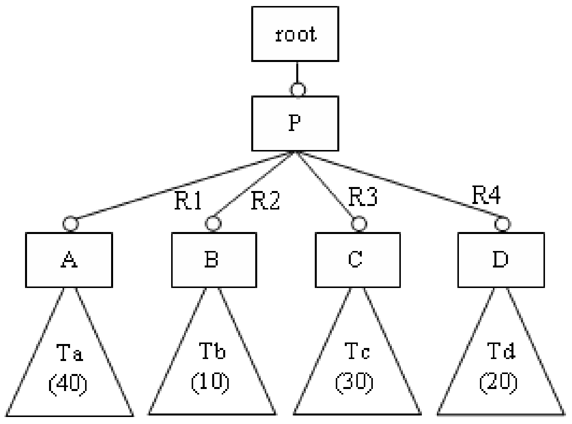
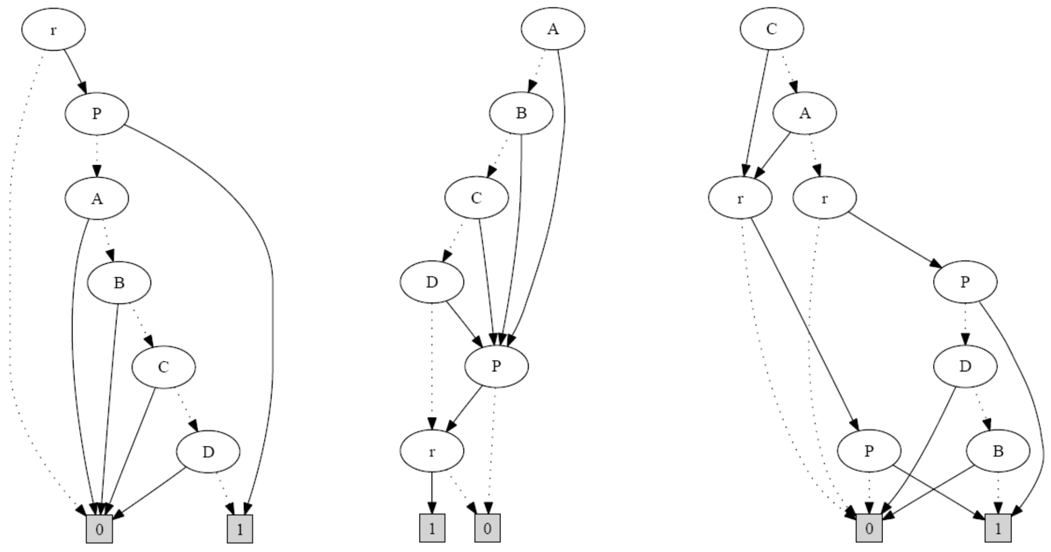


Figure 4.1: A feature tree highlighting a parent feature P and its children A, B, C, and D

mandatory features represent parent-child binary bi-implications and hence can be automatically inferred from their parent features (or another ancestor if the parent is a mandatory feature as well). For instance, consider an optional parent feature p and its mandatory child feature c. Hence, relation ($p \leftrightarrow c$) must hold which requires c to assume the same truth value as p in all valid configurations of the model. Therefore, c can be eliminated from analysis and inferred from p's assignment. A simplification algorithm safely removes mandatory features from the feature tree and updates all references to such features both in the feature tree and in the extra constraints, while preserving the core semantics of the model. The reduction of the number of features in a feature model can significantly reduce the size of BDDs since each feature potentially corresponds to multiple BDD nodes. We refer to feature models for which mandatory features were safely removed as *simplified* models.

4.2.3 Parent-Child Relations Define The Connected Variables

Feature tree constraints are expressed in terms of ancestral relations and groups. Our experiments have revealed that minimizing the distance between sibling features in groups does not improve BDD sizes. Therefore, we only consider parent-child relations to identify connected variables. Figure 4.1 shows an example of four parent-child relationships involving a feature P and its children A, B, C, and D. Since all five features are optional, relations $R1$, $R2$, $R3$ and $R4$ represent binary implications ($child \rightarrow parent$). The goal of a good heuristic for the feature tree



(a)Pre-Order (b)Post-Order (c)Average-Order

Figure 4.2: BDDs for various traversals of the feature tree

should be to minimize the relative distance between P and each of its children in the variable order produced. Excessive minimization in one branch of the tree might cause poor minimization in others. For instance, one might decide to order variables P, A, B, C, and D in a straight sequence. However, by doing so features B, C and D are placed in between A and its children increasing their relative distance. In fact, if this strategy is applied recursively in the feature tree, a BFS traversal of the feature tree is implemented, which is an extremely poor ordering.

4.2.4 Depth-First Traversals Produce Good BDD Patterns

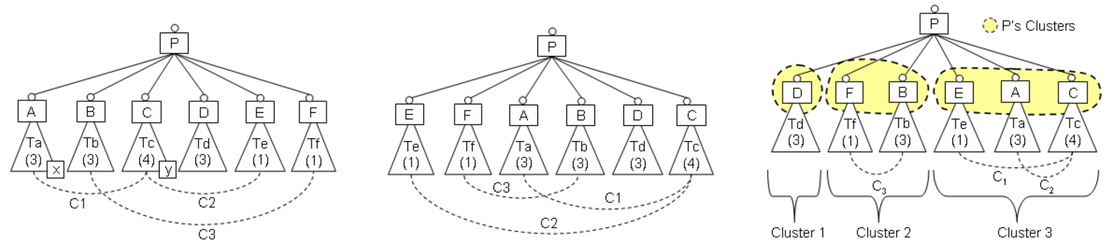
Depth-first traversals of the feature tree produce orders in which parent nodes are placed either prior to (pre-order) or following (post-order) their child nodes. This is far from ideal for reducing distances between variables. For instance, a better approach would be to place the parent node in between its children. This would clearly improve (shorten) the average distance between the parent node and its children. However, quite surprisingly this strategy produces BDDs with chaotic structures that in many cases are larger than one expects. We observed that the placement of parents prior or after their children often produced compact BDD structures. Figure 4.2 shows three BDDs for features *Root*, P, A, B, C and D from Figure 4.1. A variable order for a pre-order traversal of the feature tree is shown in Figure 4.2a (R indicates the root feature). A BDD of size 6 is shown and a very compact structure is observed for pre-order, e.g., if P is *true* the BDD evaluates to *true* no matter the values of its children. Conversely, if P is *false*, whenever A, B, C, or D are *true*, the BDD evaluates to *false*. Post-order also produces a compact pattern (Figure 4.2b). However, if P is placed between its children and R is placed near P (referred to as *average-order* in Figure 4.2c) the size of the BDD increases to 8 nodes despite the fact that the average distance of P and its children is reduced. Therefore, considering that pre- and post-order produce comparable good quality BDD patterns we arbitrarily choose pre-order as the reference variable ordering implementation from now on. We refer to this ordering simply as *natural pre-order*.

4.2.5 Sorting Decreases Parent-Child Distances

Considering that natural pre-order is able to produce compact BDD patterns the goal now becomes to minimize variables distances while enforcing those patterns. In this context, a drawback of the natural pre-order ordering is that it relies on the *natural* placement of nodes in the feature tree which, despite the good BDD patterns produced, is not necessarily good from the point of view of variable distance minimization. Consider again the feature model in Figure 4.1, showing four subtrees Ta, Tb, Tc, and Td containing 40, 10, 30, and 20 features, respectively. Natural pre-order would produce the order: $P < A < [Ta] < B < [Tb] < C < [Tc] < D < [Td]$, where $[Tn]$ replaces the set of features in subtree Tn. Hence, the total distance between feature P and its children is 180, i.e., 1 (A to P) + 42 (B to P) + 53 (C to P) + 84 (D to P). However, if the subtrees rooted by A, B, C and D are sorted in ascending order of their size the new order would be: $P < B < [Tb] < D < [Td] < C < [Tc] < A < [Ta]$ and the total distance of P and its children is reduced to 110. Note that sorting still preserves pre-order (and so the compact BDD patterns), only the relative order in which child features are visited has changed. We refer to this ordering as *sorted pre-order*.

4.2.6 Grouping Dependent Subtrees Minimizes Variable Distances In The Extra Constraints

So far we have focused primarily on the feature tree relations to order BDD variables. However, in practice feature models usually contain extra constraints attached to them that complement the relations in the feature tree. A large number



(a) Natural Pre-Order (b) Sorted Pre-Order (c) Clustered Pre-Order

Figure 4.3: Three different arrangements for P's children: A, B, C, D, E, and F

Table 4.1: Variable distances for pre-order-based traversals of the feature tree

		Feature Tree (FT) and Extra Constraint (EC) Variable Distances									
Feature Tree Traversals	Variable Order	FT Var.	EC Shortest Distance				EC Longest Distance				EC Average
		Distance	C1	C2	C3	Total	C1	C2	C3	Total	Var.
Natural Pre-Order	P<A<B<C<D<E<F	67	5	5	12	22	12	10	16	38	30
Sorted Pre-Order	P<E<F<A<B<D<C	48	9	15	5	29	16	20	9	45	37
Clustered Pre-Order	P<D<F<B<E<A<C	54	5	1	1	7	10	8	5	23	15

of relations in the extra constraints can significantly affect the size of the BDD. One way to take the extra constraints into account would be to group the children of a node together based on identified dependencies among their subtrees, instead of purely sorting nodes by subtree size. Figure 4.3a shows a parent feature P, its children A, B, C, D, E, and F, and subtrees Ta, Tb, Tc, Td, Te, and Tf rooted by each of P's children. Three extra binary constrains are shown: $C1$, $C2$ and $C3$. These constraints indicate that some of the subtrees of P's children have dependencies: Ta and Tc for $C1$, Tc and Te for $C2$, and Tb and Tf for $C3$. Different node arrangements are shown representing the visitinqg order of different pre-order-based traversals: natural pre-order (a), sorted pre-order (b), and clustered pre-order (c), where the latter will be explained shortly.

Table 4.1 shows the variable orders and the relative variable distances for the three different traversals depicted in Figure 4.3. The first row shows the distances for the natural pre-order traversal. The total distance between P and each of its children is 67 (column *FT Var. Distance*). Columns *EC Shortest Distance* and *EC Longest Distance* indicate the shortest and longest possible distances for extra constraint variables for each traversal as well as the average parent-child distance, i.e., the mean of the shortest and longest distances (column *EC Average Var. Distance*). For natural pre-order, the shortest (respectively longest) distance between variables in the constraint $C1$ is 5 (respectively 12). In the shortest-distance case, $C1$ variables correspond to features X (see bottom-right feature on subtree Ta in Fig. Figure 4.3a) and C. In the worst case, they correspond to features A and Y (see bottom-right feature on subtree Tc in Fig. Figure 4.3a). The average total distance of all variables occurring in the extra constraints is 30 (column *EC Average Var. Distance*).

The sorted pre-order traversal (second row in Table 4.1) considers sorting child nodes in ascending order of the size of their subtrees. The distance between P and its children is reduced to 48. However, since this traversal does not take the extra constraint into account a bad average distance of 37 is observed. Figure 4.2b shows the new arrangement of P's children for sorted pre-order.

The third traversal, clustered pre-order, considers using extra constraint relations to decide which nodes should be visited first. Note that in Figure 4.3c nodes A, B, C, D, E and F were rearranged based on the dependencies of their subtrees. Features E, A, and C were grouped together into *clusters* since constraints $C1$ and $C2$ connect their subtrees. The same is observed for features F and B because of constraint $C3$. Feature D is isolated as none of its descendants is referenced in the extra constraints. Three clusters are shown in Figure 4.3c: Cluster 1, Cluster 2, and Cluster 3. Note that the clusters have been sorted according to their size from left to right so that larger clusters are in the rightmost positions. The size of a cluster is the total number of nodes of its contained trees. The combination of these two techniques, sorting and clustering, can considerably improve the quality of orders produced by clustered pre-order traversals. In fact, while clustering enforces distance minimization of extra constraint variables, sorting aims at parent-child distance minimization in the feature tree. A slightly higher distance for parent-child variables is observed for the clustered pre-order when compared to sorted pre-order (54 against 48, respectively), but still much better than natural pre-order (67). Yet, a significant improvement on distance minimization for extra constraint variables is achieved (15 against 37 for sorted pre-order and 30 for natural pre-order).

4.3 New Heuristics for Feature Models

In this section, we propose two novel heuristics to order BDD variables for feature models taking the considerations made in the previous section into account. The new heuristics are based on pre-order traversals of the feature tree and rely heavily on the sorting and clustering techniques discussed earlier. Both heuristics assume that a pre-processing stage has been performed in the feature tree in order to

Algorithm 1 Clustering algorithm for the feature tree

```
Function FT-create-clusters()
1:  $EC$ = Extra constraints in CNF
2:  for (each clause $C$ in $EC$) do
3:      for (each non-repeated pair of variables $(v_i, v_j) \in C, v_i \neq v_j$) do
4:          $A$ = LCA$(v_i, v_j)$
5:          if (clusters set of $A$ is empty) then
6:              $CS$ = create-initial-clusters-set($A$)
7:          else
8:              $CS$ = clusters set attached to $A$
9:          end if
10:     end for
11:     $R$ = Roots$(v_i, v_j)$
12:     $MC$ = merge-clusters-sharing-elements($CS, R$)
13:     add relation $R$ to merged cluster $MC$
14: end for
15: for (each feature $F$ without a clusters set attached) do
16:     $CS$ = create-initial-clusters-set($F$)
17: end for
```

decorate the tree with clusters. Next, we discuss the clustering algorithm that will further support the proposed ordering heuristics.

4.3.1 Clustering Procedure

A cluster C consists of two sets: a set F that defines the features in the cluster, and a set of relations R that describe how the features in F are related to each other. Clusters can be used to group the child nodes of a given feature to indicate that their subtrees have dependencies caused by the extra constraints attached to the feature model. For instance, consider a feature f, its child features a, b, c, d and e, and the subtrees T_a, T_b, T_c, T_d, and T_e rooted by each of those child features, respectively. Clusters C_1 and C_2 below split the child features into two groups, i.e., $\{a, b, d\}$ and $\{c, e\}$ (see set F in each cluster). In addition, cluster C_1 indicates that subtrees T_a and T_d as well as T_a, T_b and T_d have dependencies (see set R in cluster C_1). Notice that the same subtrees may appear in more than one relation indicating that multiple extra constraint relations connect those subtrees. Similarly, cluster C_2 indicates that subtrees T_c and T_e have dependencies. Notice that even when the relations contain the exact same features they are considered different (see set R containing two similar relations $\{c, e\}$ in cluster C_2). Notice that clusters are always associated with a single parent feature. In the example below, clusters C_1 and C_2 are associated with feature f.

$C_1 = \{ \quad F = \{a, b, d\} \quad R = \{\{a, d\}, \{a, b, d\}\} \quad \}$

$C_2 = \{ \quad F = \{c, e\} \quad R = \{\{c, e\}, \{c, e\}\} \quad \}$

Operation *FT-create-clusters* illustrated in Algorithm 1 creates clusters in the feature tree based on the extra constraint relations. The extra constraints are initially converted to CNF (line 1). Next, the algorithm visits each clause in the CNF formula individually (line 2). For each clause, all combinations of pairs of variables are examined (line 3). For a given pair (v_i, v_j) the lowest common ancestor A=LCA(v_i, v_j) is found (line 4). A represents the parent feature for which clusters will be created because of the dependency between variables (features) v_i and v_j in the CNF clause being currently processed. If clusters have never been created for A (line 5), a new cluster set CS is created using operation *create-initial-clusters-set* (see Algorithm 31 in page 170 in the appendix). This operation simply creates one cluster for each child feature C of A containing no relations. Now, each of A's child belong to an individual cluster (line 6). Instead, if A already has clusters associated with it, set CS is assigned the current clusters set (line 8). In line 11, A's children that root the distinct subtrees containing v_i and v_j are retrieved and stored in set R. These two child nodes will form a single cluster since their subtrees have dependencies. R is used as an input to operation *merge-clusters-sharing-elements* in order to merge all of A's clusters that share features in R (see Algorithm 32 on page 170 in the appendix). The operation returns a new cluster MC that represents the combination of the features and relations of all merged clusters (line 12). Relation R is then added to cluster MC to indicate the dependency between the subtrees rooted by the features in the relation (line 13). For completion, the algorithm examines all parent features in the feature tree for which clusters have not been created. For each of those, operation *create-initial-clusters-set* is called in order to create a single cluster for each child feature (lines 15-17). This indicates that the child nodes have no dependencies.

4.3.2 Variable Ordering Heuristics

We introduce two new heuristics to order BDD variables for feature models: *Pre-CL-Size* and *Pre-CL-MinSpan*. As we mentioned, both heuristics assume that

Algorithm 2 *Pre-CL* parameterized recursive algorithm for heuristics *Pre-CL-Size* and *Pre-CL-MinSpan*

```
O: variable order set (non-repeated elements)
N: feature being visited in the feature model
S: constant that indicates how clusters' internal nodes are sorted

Function Pre-CL(O: feature{}, N: feature, S: constant) : { }
 1: add N to the end of set O
 2: CS = clusters set attached to N
 3: Sort clusters in CS in ascending order of size
 4: for (each cluster CL in the now sorted set CS) do
 5:     if (S = SIZE) then
 6:         Sort CS's internal nodes in ascending order of subtree size
 7:     else if (S = MIN_SPAN) then
 8:         Use FORCE to sort CS's internal nodes:
            - FORCE initial order is the current order of the nodes
            - FORCE relations are the relations in the cluster
 9:     end if
10:     for (each node F in the now sorted cluster CL) do
11:         Pre-CL-Rec(O,F,S)
12:     end for
13: end for
14: return O
```

children nodes have been clustered in the feature tree. Moreover, since the heuristics share many implementation aspects, a single parameterized algorithm is provided. In fact, we refer to both heuristics as part of the *Pre-CL* family of heuristics as we hope that the family will gain new members in the future.

Operation *Pre-CL* implements a recursive parameterized algorithm for the *Pre-CL* heuristics (see Algorithm 2). The algorithm takes as input an initially-empty variable set O, a feature N, and a constant S. O is updated by the algorithm to store the final variable order. N represents the feature being currently visited by the algorithm. In the first call to operation *Pre-CL* N is assigned the root of the feature tree. S is a constant that indicates the strategy to be used to sort the internal features of each processed cluster. Strategy *SIZE* is used by heuristic *Pre-CL-Size* while strategy *MIN_SPAN* applies to heuristic *PRE-CL-MinSpan*. The algorithm starts by adding node N to the end of set O (line 1). Next, the clusters associated with N are retrieved and stored in variable CS. CS represents a set of one or more clusters (line 2). The clusters in CS are sorted left-to-right in ascending order of their size. Recall that the size of a cluster is the sum of the sizes of each subtree rooted by a feature in the cluster. The sorting procedure will move the larger clusters to the rightmost positions in CS (line 3). The next step involves another sorting procedure to sort the internal nodes of a cluster. Each cluster CL in the set CS is examined left-to-right (line 4). If strategy S is constant *SIZE* the features in cluster CL are sorted in ascending order of the size of their subtrees. That is, the features rooting the larger subtrees are in placed in the rightmost positions in the cluster (lines 5-6). Notice that the sorting does not affect the relations in the cluster. Instead, if the strategy to be applied is *MIN_SPAN*, the FORCE algorithm is used to sort the cluster's internal nodes. FORCE is passed to as input the features and the relations in cluster CL (lines 7-8). As opposed to sorting by size as in the SIZE strategy, FORCE will attempt to move the root features of highly-connected subtrees to center positions in order to minimize their relative distance. Once cluster CL is sorted, its features are visited left-to-right and a recursive call is made with parameters O, feature F representing the next feature to visit in the now sorted cluster, and strategy S (lines 10-12). Line 14, returns set O containing the order in which the features in the feature tree have been visited.

4.3.3 Clustering and Heuristic Illustration

Figure 4.4 illustrates the clustering procedure and how the *Pre-CL* heuristics prune the subtrees in the feature tree during the pre-order traversal. A feature model is depicted on the top of the figure. Features are simply represented by circles with internal labels. Feature r roots the model. Three extra constraints C_1, C_2, and C_3 are attached to the model. For simplicity, we assume that each of those constraints represent a CNF clause and thus only the variables are shown. Constraint C_1, C_2 and C_3 represent relations involving features a and m, g and d, and a, k, and n, respectively.

The clustering procedure starts by examining constraint C_1. The lowest common ancestor of features a and m is found, i.e., the root feature r. Since r does not have clusters created, the algorithm will create one cluster for each of r's children as shown next:

cluster-r$1 = \{F = \{a\}, R = \{\}\}$

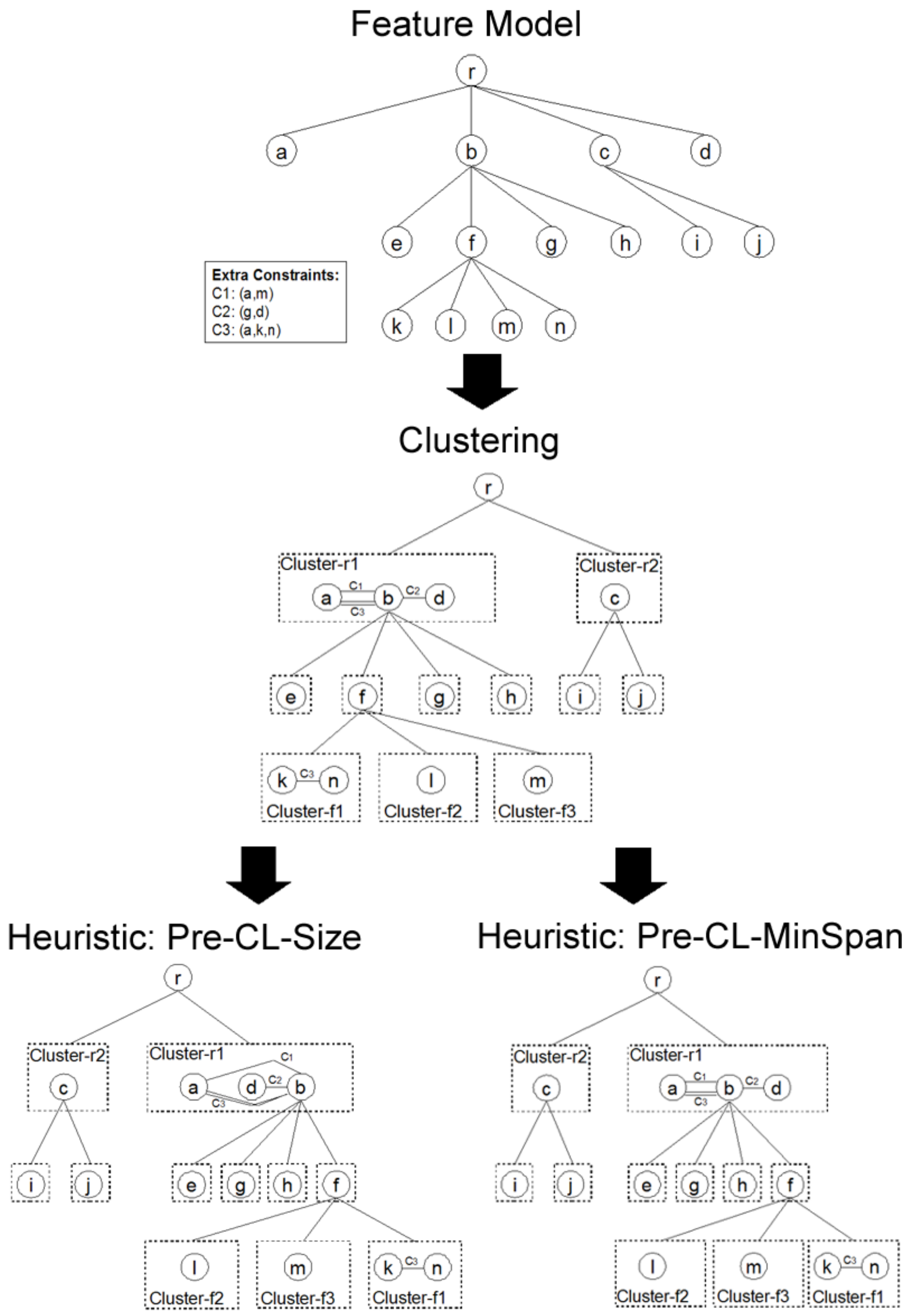


Figure 4.4: A feature model is clustered and rearranged by heuristics Pre-CL-Size and Pre-CL-MinSpan

$\textit{cluster-}r2 = \{F = \{b\}, R = \{\}\}$
$\textit{cluster-}r3 = \{F = \{c\}, R = \{\}\}$
$\textit{cluster-}r4 = \{F = \{d\}, R = \{\}\}$

Next, function *Roots*(a,m) is applied to merge r's clusters. This function returns the set R=$\{a$,$b\}$. Therefore, all clusters containing features a or b need to be merged. Set R is also added as a relation in the merge cluster. This causes r's clusters to be rearranged as follows:

$\textit{cluster-}r1 = \{F = \{a, b\}, R = \{\{a, b\}\}\}$
$\textit{cluster-}r3 = \{F = \{c\}, R = \{\}\}$
$\textit{cluster-}r4 = \{F = \{d\}, R = \{\}\}$

Next, constraint C_2 is examined. Again, root feature r is the LCA(g,d). Roots(g,d) returns set $\{b$,$d\}$ which causes another rearrangement of r's clusters after the merging procedure. Clusters *cluster-*$r1$ and *cluster-*$r4$ are merged and we renamed *cluster-*$r3$ to *cluster-*$r2$ as follows:

$\textit{cluster-}r1 = \{F = \{a, b, d\}, R = \{\{a, b\}, \{b, d\}\}\}$
$\textit{cluster-}r2 = \{F = \{c\}, R = \{\}\}$

Unlike constraints C_1 and C_2, constraint C_3 is not binary and therefore variable pairs (a,k), (a,n), and (k,n) need to be considered. Pairs (a,k) and (a,n) will once again find the root node as the lowest common ancestor and nodes a and b as their roots. As clusters exist for r, this will only cause two new relations $\{a$,$b\}$ to be added to cluster *cluster-*$r1$. On the other hand, pair (k,n) does not find the root node as its LCA but node f. Since f does not have clusters, four initial clusters are created, one for each of f's child. Since features k and n are children of f these nodes are also their own roots. Therefore, relation (k,n) is used to merge f's clusters. r and f's clusters now appear as follows (also as illustrated in the feature model in the center of Figure 4.4):

$\textit{cluster-}r1 = \{F = \{a, b, d\}, R = \{\{a, b\}, \{b, d\}, \{a, b\}, \{a, b\}\}\}$
$\textit{cluster-}r2 = \{F = \{c\}, R = \{\}\}$
$\textit{cluster-}f1 = \{F = \{k, n\}, R = \{\{k, n\}\}\}$
$\textit{cluster-}f2 = \{F = \{l\}, R = \{\}\}$
$\textit{cluster-}f3 = \{F = \{m\}, R = \{\}\}$

Notice that in Figure 4.4 the clusters' relations are illustrated as lines connecting the clusters' internal features. In addition, the constraint labels (C_1, C_2, and C_3) are placed near those lines to indicate which constraint built the relation. For instance, in cluster *cluster*-$r1$ a line connecting features a and b has been labeled C_1 since this constraint has caused the dependency between those features.

The two feature models on the bottom of Figure 4.4 show how heuristics *Pre-CL-Size* and *Pre-CL-MinSpan* prune the feature tree during the pre-order traversal to order variables. Notice, though, that the heuristics do not actually move the nodes in the tree but rather guide the traversal procedure through paths that minimize the distance of variables in the extra constraints. One of the first steps performed by both heuristics is to sort all clusters in the feature tree in ascending order of size (line 3 of Algorithm 2). This measure intends to shorten the distance between a parent node and its children while respecting the children's dependencies represented in the clusters. Therefore, notice that r and f's clusters have been sorted in the same way in both feature models. *cluster*-$r2$ of size 3 has been placed prior to *cluster*-$r1$ that contains 11 nodes in its subtrees. Similarly, clusters *cluster*-$f2$ and *cluster*-$f3$ with 1 node each have been placed prior to cluster *cluster*-$f3$ that contains two nodes. The same cluster sorting strategy has been applied throughout the feature tree.

What distinguishes the *Pre-CL* heuristics is the way the heuristics sort cluster's internal nodes. Heuristic *Pre-CL-Size* gives priority to parent-child relations in the feature tree and thus sort clusters based on their size. For instance, in the feature model on the bottom-left of Figure 4.4 the features in cluster *cluster*-$r1$ have been sorted in ascending order of the size of their subtree's. Hence, nodes a and d with size 1 have been placed prior to node b with size 9. Instead, heuristic *Pre-CL-MinSpan* (bottom-right) sorts cluster's features in such a way that features with a high number of connections are placed between the others in an attempt to minimize the distance of extra constraint variables for pre-order traversals. Therefore, feature b in cluster *cluster*-$r1$ has been placed between features a and d since b has 3 connections with a and 1 connection with d. The order produced by both heuristics is generated by traversing the feature models illustrated in pre-order as shown below:

Pre-CL-Size variable order: $\{r,c,i,j,a,d,b,e,g,h,f,l,m,k,n\}$
Pre-CL-MinSpan variable order: $\{r,c,i,j,a,b,e,g,h,f,l,m,k,n,d\}$

Heuristic *Pre-CL-Size* seems to be more effective when the number of dependencies between cluster's features is low in which case prioritizing parent-child relations in the feature tree can be more advantageous. On the other hand, heuristic *Pre-CL-MinSpan* can be more effective when there are a fairly large number of relations in the cluster concentrated on a few features in which case it is worth placing those features in between their dependent features.

4.4 Summary

In this chapter, we explored several structural properties of feature models to propose two novel heuristics for ordering BDD variables for the feature modeling domain. The heuristics are based on pre-order traversals of the feature tree. However, rather than following the "natural" arrangement of the nodes in the feature tree, the traversal is guided by a clustering algorithm that identifies and connects dependent sub-trees. As a result, the overall distance of the BDD variables in the orders produced is reduced while compact structural patterns are enforced. In Chapter 6, the proposed heuristics are further evaluated by contrasting them with existing general and domain-specific heuristics through empirical experiments.

Chapter 5

Reasoning with Domain-Specific and SAT-Based Systems

In Chapter 2, we discussed how the translation of feature models to propositional formulas enables the use of constraint systems to reason on feature models and product configuration. For instance, a SAT solver can assist feature model designers in debugging feature models by checking the satisfiability of the models or the presence of "dead" features. In addition, a SAT solver can be used to check the soundness of transformations applied to feature models such as refactorings or to complete partial system specifications on behalf of users in product configuration. Currently, many configuration systems and reasoning approaches rely on the use of constraint solvers as discussed in Chapter 3.

However, as we argued before, current research exploring the use of constraint solvers to reason on feature models have adopted a "black-box" approach. That is, the techniques proposed have mostly focused on translating a feature model into a corresponding Boolean formula and using an off-the-shelf constraint system to reason on the formula. While "black-box" approaches are convenient as they refrain from delving into the intricacies of constraint systems they also neglect some important sensitive issues related to these systems. For instance, satisfiability is a well-known NP-complete problem which means that a SAT solver may take an infeasibly long time to check the satisfiability of an "intractable" Boolean formula. Yet, we do not know whether SAT instances derived from realistic feature models can lead to intractability problems. In order to address this question properly, it is necessary to understand the mechanics of SAT solvers and examine carefully relevant properties of feature models that can provide some evidence about the tractability of SAT instances derived from feature models. Another great advantage of examining such properties is that they open many opportunities for developing efficient domain-specific algorithms for feature models as will be discussed next.

In this chapter, we explore several properties of the feature modeling domain to propose efficient domain-specific and hybrid reasoning algorithms for feature models. In addition, we examine properties that can help better understanding the hardness of SAT instances derived from feature models.

5.1 FTRS: A Reasoning System for Feature Trees

As previously discussed, a feature model can be converted into a corresponding propositional formula by applying the translation rules depicted in Table 2.1 to its feature tree and conjoining the resulting formula with the extra constraints. As a consequence, a SAT solver can be conveniently used to reason on the resulting formula that represents the model.

In this section, we argue that there are advantages in not converting the feature tree to a corresponding propositional formula but rather to use a tree structure that conforms to the meta-model depicted in Figure 2.2 (*FeatureTree* element) as basis for encoding feature relations. In fact, we show that there are interesting properties in formulas represented in such a tree that support the development of efficient reasoning algorithms. In the following, we explore some of these properties to develop a domain-specific reasoning system for feature trees called *FTRS* (Feature Tree Reasoning System).

The benefits of the *FTRS* are three-fold. First, it provides an efficient set of reasoning operations for feature trees. Second, it provides insights on how to take advantage of properties of feature trees to build efficient domain-specific reasoning algorithms. As will be shown in Section 5.2, we can further integrate the FTRS with a constraint system to form a hybrid reasoning system for the entire feature model. That is, the *FTRS* cannot be used standalone to reason on feature models having extra constraints attached. Third, the properties examined for the construction of the *FTRS* can be used to better understand the hardness of feature model SAT instances as will be discussed in Section 5.3.

In the following, we discuss the *FTRS* system and introduce its supporting operations as listed in Table 5.1. Notice that the operations are named with prefix "FT" as a reference to feature trees.

Table 5.1: FTRS operations

Operation	**Description**
FT-assign(f,v)	Assign the truth value v to feature f; f becomes instantiated
FT-save-state(*id*)	Save the current state of the feature tree and associate it to identifier *id*
FT-restore-state(*id*)	Restore the feature tree rooted to a given state *id*
FT-propagate(f,v)	Propagate the assignment of v to f throughout the feature tree
FT-is-satisfiable(f)	Returns *true* if the feature tree rooted by f is satisfiable or $false$, otherwise
FT-count-sol(f)	Returns the number of solutions in the feature tree rooted by f
FT-create-sol-iterator(f)	Returns an iterator object that can be used to enumerate the solutions in the feature tree rooted by feature f
FT-has-next-sol(*iterator*)	Returns *true* if the solution iterator object *iterator* still has a solution to enumerate or *false* otherwise
FT-next-sol(*iterator*)	Returns a list of features representing the next solution of the solution iterator object *iterator*

5.1.1 Assigning values

Operation *FT-assign* assigns a truth value to a feature in the feature tree. This operation can be triggered, for instance, by a user performing configuration actions on a feature tree such as selecting (*true* assignment) or deselecting (*false* assignment) features. Algorithm 3 implements the *FT-assign* operation. A feature f and a truth value v are passed as input parameters. If feature f is uninstantiated it is assigned the truth value v (lines 1-2) and the function returns *true* (line 6). Otherwise, if the current value of f is different from v a conflict has been found and the function returns *false* to indicate this fact (lines 3-5).

5.1.2 Saving and restoring states

The *FT-save-state* (Algorithm 4) and *FT-restore-state* (Algorithm 5) operations save and restore the state of the feature tree, respectively. For each instantiated

Algorithm 3 Assign feature f the truth value v

Inputs:
f: feature to be assigned a value
v: truth value to be assigned to f
Output:
true if assignment succeeds or *false*, otherwise

Function **FT-assign**(f : $feature$, v : $Boolean$)

```
1: if ($f$ is uninstantiated) then
2:     $f = v$
3: else if (FT-get-value($f$) $\neq v$) then
4:     return *false*Conflict!
5: end if
6: return *true*
```

feature in the feature tree its name and truth value are saved and associated with a unique identifier. The identifier can be used to restore the feature to a particular saved state. These operations are particularly important in the integration of the *FTRS* to a general-purpose constraint solver as will be shown in Section 5.2.

Algorithm 4 Save the current state of the feature tree and associate it to identifier *id*

Inputs:
id: state identifier

Function **FT-save-state**(*id*:string)

```
1: $state = \{\}$
2: for (each feature $p$ in the feature tree) do
3:     add tuple$\langle p$, FT-get-value($p$)$\rangle$ to *state*
4: end for
5: associate *state* to identifier *id*
6: add *id* to the list of state identifiers
```

Algorithm 5 Restore the feature tree rooted to a given state *id*

```
Inputs:
 $id$: state identifier

Function FT-restore-state($id$:string)
1: $state$ = state associated with identifier $id$
2: for (each tuple $\langle p, v\rangle$ in $state$) do
3:     FT-assign($p$, $v$)
4: end for
5: remove $id$ from the list of state identifiers
```

5.1.3 Propagating value assignments

Constraint propagation [76] is a very important mechanism used by constraint solvers to enforce local consistency and optimize the search process as discussed in Section 2.3 (page 17). For instance, some propagation techniques such as forward checking operate by eliminating values from variable domains that can not be part of any solution during the search procedure. In practice, this can improve the efficiency of a constraint solver tremendously as it prevents the solver from examining numerous unproductive branches in the search tree. Propagation is particularly effective in the Boolean domain (e.g., unit propagation) as eliminating a value from the domain of a Boolean variable (say *false*) is equivalent to instantiating the variable to the other value (*true*).

Instead of relying on a SAT solver propagation algorithm that would cause a dependency between the *FTRS* and a proprietary SAT infra-structure we developed a domain-specific propagation algorithm for feature trees. This make the *FTRS* fully independent of any particular SAT implementation and additionally provides insights on how to build recursive algorithms for feature trees.

The propagation algorithm works on feature trees and has the same effect as known algorithms such as unit propagation and forward checking. That is, the algorithm examines contexts containing a single uninstantiated variable and for which a consistent assignment for the variable can be computed. Before delving into the details of the propagation algorithm we first discuss the various scenarios in which propagation can be applied in a feature tree.

Figure 5.1 compiles all possible scenarios involving a given feature P and the impact an assignment to P may cause to adjacent features, i.e., P's parent, P's

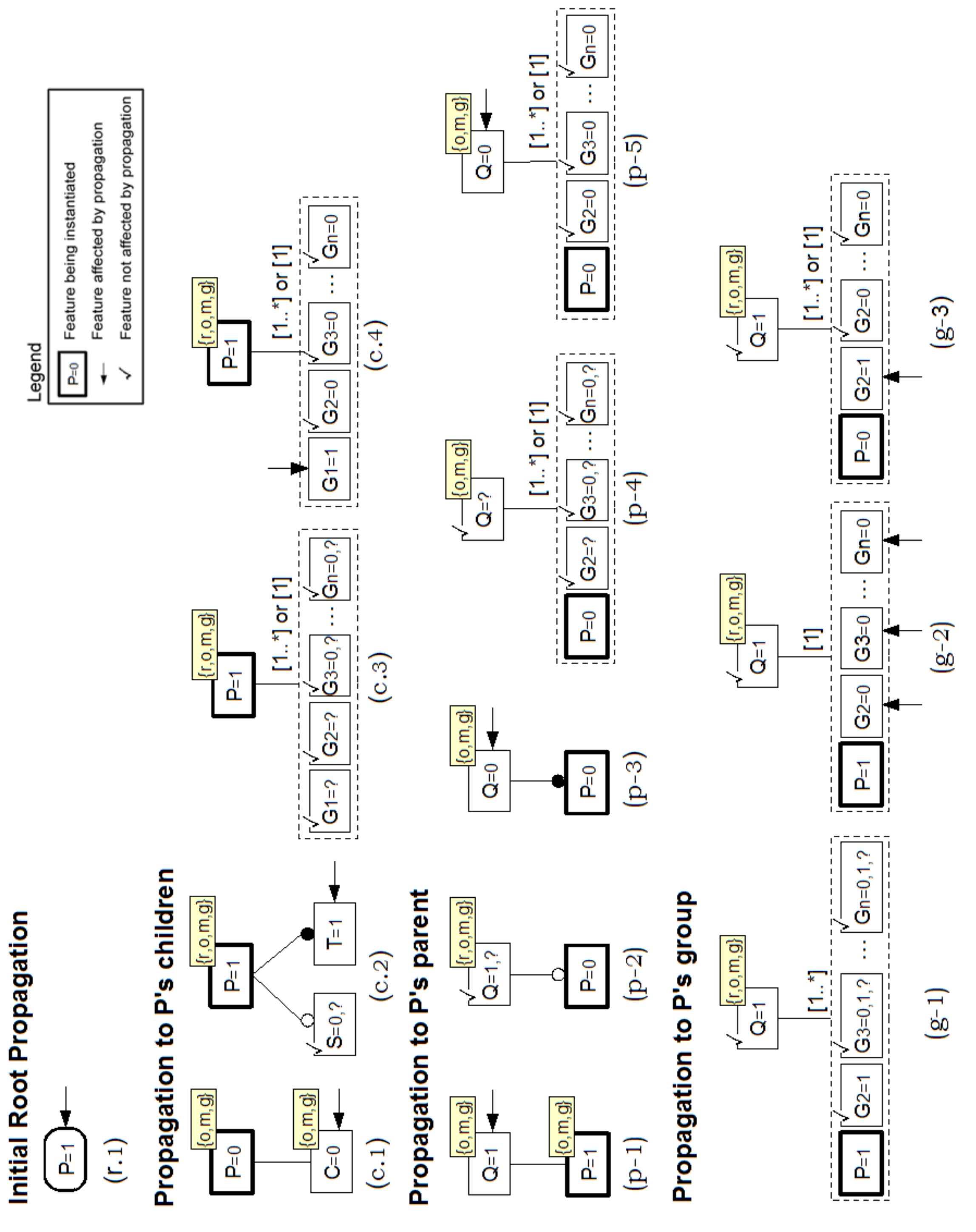


Figure 5.1: All possible propagation scenarios for a given feature P

children and, if P is a grouped feature, P's sibling features in the group. Root, mandatory, optional and grouped features are represented using the corresponding notation described in Figure 2.1. In addition, whenever a scenario can be applied to multiple types of features, the feature object is drawn as a rectangle decorated with a curly-braced set in the top-right corner. The letters in the set "r" (root), "o" (optional), "m" (mandatory), and "g" (grouped feature) indicate the kinds of features that can assume the role of the feature object. For instance, in scenario *c.1* parent feature P represents any of the three types listed in the set {o,m,g}, i.e., an optional, mandatory, or grouped feature. In the figure, P=0 and P=1 represent a *false* and *true* assignment to P, respectively. A question mark (?) indicates that the feature is uninstantiated. Notice that because each scenario might describe multiple situations a feature can be assigned more than one truth value and a question mark. This indicates that the feature can assume any of these values in the scenario. For instance, scenario *c.2* represents the cases in which a parent feature P and its mandatory child feature T are *true* while the optional child feature S is either uninstantiated or *false*. Similarly, feature groups are sometimes shown containing more than one cardinality relation to represent both inclusive-OR and exclusive-OR groups. A check mark symbol in the top-left corner of a feature indicates the state of the feature prior to assigning a value to feature P. An arrow indicates the features that have been instantiated as a result of propagating the assignment on P. For example, in scenario *c.2* feature S is initially uninstantiated or *false*. Upon assigning *true* to P propagation assigns *true* to T as indicated by the arrow.

In the following, we discuss each propagation scenario in Figure 5.1 in detail.

Table 5.2: All possible propagation scenarios for feature P as shown in Figure 5.1

Initial Root Propagation

Scenario	Propagation Effect
r.1	The first propagation step performed in a feature tree is triggered by the unary formula (P) that requires the root of the feature tree to be always *true* (see first row in Table 2.1). This assigns *true* to the root feature P as shown in scenario *r.1*. Notice that this might trigger further rounds of assignments and propagations in the tree as shown next.

Propagation to *P*'s Children

Scenario	Propagation Effect
c.1	Consider a parent feature P of type optional, mandatory or grouped and its child feature C of type optional, mandatory, or grouped. C is uninstantiated. P is assigned *false*. Formula f_c=($C \rightarrow P$) must hold (see second row in Table 2.1). If P is assigned *false* C must be *false* in order to satisfy formula f_c. Therefore, scenario *c.1* uses an arrow pointing to feature C to indicate that propagation has set this feature to *false*.
c.2	Consider a parent feature P of type root, optional, mandatory or grouped and its optional and mandatory features S and T, respectively. S is *false* or uninstantiated while T is uninstantiated. P is assigned *true*. Formulas f_o=($S \rightarrow P$) and f_m=($T \leftrightarrow P$) must hold (see second and third rows in Table 2.1). If S is *false* formula f_o is already satisfied. If S is uninstantiated assignment $P = true$ satisfies f_o and allows S to assume any truth value. Therefore, propagation does not impact S. On the other hand, propagation assigns *true* to T in order to satisfy formula f_m that requires P and T to necessarily assume the same truth value. Scenario *c.2* uses an arrow pointing to feature T to indicate that propagation has set this feature to *true*.

c.3	Consider a parent feature P of type root, optional, mandatory or grouped and its n child grouped features $G_1, G_2, \ldots, G_n$ part of an inclusive-OR ([1..*]) or exclusive-OR ([1]) feature group. At least two grouped features G_1 and G_2 remain uninstantiated in the group while all others are either *false* or uninstantiated. P is assigned *true*. Assuming the case of an inclusive-OR group, formula f_{ig}=($P \leftrightarrow (G_1 \vee \ldots \vee G_n)$) must hold (see fourth row in Table 2.1). Therefore, assignment P = *true* requires at least one of the uninstantiated features to be *true*. However, because there are at least two uninstantiated features that can be potentially set to *true*, propagation does not instantiate any features. The same applies to exclusive-OR groups in which formula f_{eg}=($P \leftrightarrow (G_1$ **xor** $\ldots$ **xor** $G_n)$) must hold (see fifth row in Table 2.1). If P is *true*, one and only one grouped feature must be *true*. However, once again because at least two features remain uninstantiated in the group, propagation will not affect any grouped features. Notice, however, that in both cases formulas f_{eg} and f_{ig} are satisfiable. For instance, if G_1 is assigned *true* and all other uninstantiated features are set to *false* both formulas are satisfied.
c.4	Consider the same features as in scenario *c.3* but now with all grouped features assigned *false* but feature G_1. If P is assigned *true* propagation must assign *true* to G_1 in order to satisfy formulas f_{eg} and f_{ig}. Therefore, scenario c.4 shows an arrow pointing to feature G_1 to indicate that propagation has set this feature to *true*.
Propagation to *P*'s Parent	
Scenario	Propagation Effect
p.1	Consider a child feature P of type optional, mandatory or grouped and its parent feature Q of type optional, mandatory, or grouped. Q is uninstantiated. P is assigned *true*. Formula f_c=($P \rightarrow Q$) must hold (see second row of Table 2.1). Therefore, Q must be *true* to satisfy f_c. Scenario p.1 shows an arrow pointing to feature Q to indicate that propagation has set this feature to *true*.

p.2 Consider an optional child feature P and its parent feature Q of type root, optional, mandatory, or grouped. Q is initially *true* or uninstantiated. P is assigned *false*. Formula f_c=($P \rightarrow Q$) must hold (see second row of Table 2.1). If Q is *true* formula f_c is already satisfied. If Q is uninstantiated the assignment $P =$ *false* will satisfy f_c and allow Q to be either *true* or *false*. Therefore, propagation does not affect Q.

p.3 Consider a mandatory child feature P and its parent feature Q of type optional, mandatory, or grouped. Q is initially uninstantiated. P is assigned *false*. Formula f_c=($P \leftrightarrow Q$) must hold (see third row of Table 2.1). Therefore, feature Q is assigned *true* to satisfy f_c. Scenario p.3 shows an arrow pointing to feature Q to indicate that propagation has set this feature to *false*.

p.4 Consider a grouped feature P, its $n-1$ siblings $G_2, \ldots, G_n$ part of an inclusive-OR ([1..*]) or exclusive-OR ([1]) feature group, and its parent feature Q of type optional, mandatory , or grouped. At least one of P's siblings is uninstantiated while all others are either *false* or uninstantiated. Q is uninstantiated. P is assigned *true*. Assuming the case of an inclusive-OR group, formula f_{ig}=($Q \leftrightarrow (P \vee G_2 \vee \ldots \vee G_n)$) must hold (see fourth row in Table 2.1). The assignment $P =$ *false* does not instantiate Q as this feature can still assume a *true* or *false* value considering that at least one grouped feature remains uninstantiated. Hence, propagation does not affect Q. The same applies to exclusive-OR groups in which formula f_{eg}=($Q \leftrightarrow (P \textbf{ xor } G_2 \textbf{ xor } \ldots \textbf{ xor } G_n)$) must hold (see fifth row in Table 2.1). If P is *false*, Q can still assume any truth value since at least one grouped feature remains uninstantiated. Notice, however, that in both cases formulas f_{eg} and f_{ig} are satisfiable. For instance, if Q and G_2 are assigned *true* and all other uninstantiated features are set to *false* both formulas are satisfied.

p.5 Consider the same features as in scenario *p.4* but now with all P's siblings set to *false*. If P is assigned *false* propagation must assign *false* to Q in order to satisfy formulas f_{eg} and f_{ig}. Therefore, scenario p.5 in Figure 5.1 shows an arrow pointing to feature Q to indicate that propagation has set this feature to *false*.

Propagation to *P*'s Group

Scenario	Propagation Effect
g.1	Consider a grouped feature P, its $n-1$ siblings in the group $G_2, \ldots, G_n$ part of an inclusive-OR ([1..*]) feature group, and its parent feature Q of type root, optional, mandatory or grouped. At least one of P's siblings is *true* while all others are unknown (uninstantiated, *true*or *false*). Q is *true*. P is assigned *true*. Formula f_{ig}=($Q \leftrightarrow (P \vee G_2 \vee \ldots \vee G_n)$) must hold (see fourth row in Table 2.1). Since one of the grouped features is *true* formula f_{ig} is already satisfied. Hence, propagation does not affect any of P's siblings.
g.2	Consider a grouped feature P, its $n-1$ siblings in the group $G_2, \ldots, G_n$ part of an exclusive-OR ([1]) feature group, and its parent feature Q of type root, optional, mandatory or grouped. All P's siblings are uninstantiated. Q is *true*. P is assigned *true*. Formula f_{eg}=($Q \leftrightarrow (P$ **xor** G_2 **xor** $\ldots$ **xor** $G_n)$) must hold (see fifth row in Table 2.1). If P is assigned *true* all other grouped features must be *false* to satisfy f_{eg}. Scenario g.2 shows an arrow pointing to each of P's siblings to indicate that propagation has set them to *false*.
g.3	Consider a grouped feature P, its $n-1$ siblings in the group $G_2, \ldots, G_n$ part of an inclusive-OR ([1..*]) or an exclusive-OR ([1]) feature group, and its parent feature Q of type root, optional, mandatory or grouped. Exactly one of P's siblings G_2 is uninstantiated while all others are *false*. Assuming the case of an inclusive-OR group, formula f_{ig}=($Q \leftrightarrow (P \vee G_2 \vee \ldots \vee G_n)$) must hold (see fourth row in Table 2.1). Therefore, assignment P = *false* requires G_2 to be *true* in order to satisfy formula f_{ig}. The same applies to exclusive-OR groups in which formula f_{eg}=($Q \leftrightarrow (P$ **xor** G_2 **xor** $\ldots$ **xor** $G_n)$) must hold (see fifth row in Table 2.1). If P is *false* G_2 is set to *true* in order to satisfy formula f_{eg}. Scenario g.2 shows an arrow pointing to G_2 to indicate that propagation has set this feature to *true*.

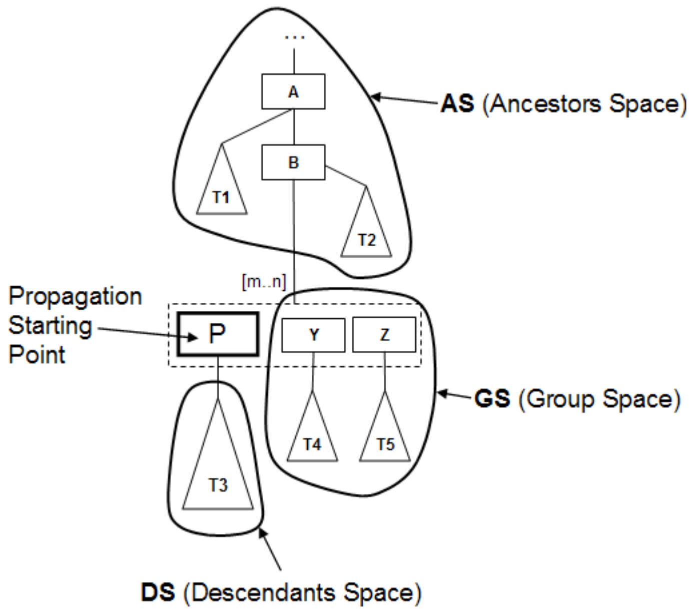


Figure 5.2: Propagation spaces for feature P

For all scenarios depicted in Table 5.2, if P is instantiated and assigned the same truth value no propagations are performed. Instead, if P is assigned a different truth value a conflict error is raised. For instance, in scenario *c.1* if P is initially *true* an error is raised since P can not be assigned *false* as illustrated in the scenario.

Based on the propagation scenarios depicted in Figure 5.1 we developed a propagation algorithm for feature trees. The algorithm attempts to identify contexts in the feature tree that match scenarios in Figure 5.1. Whenever a match is found the corresponding propagation steps described in the scenario are performed. If a conflict is found the algorithm raises an error to indicate this fact. The algorithm is naturally recursive since propagating a context can give rise to new contexts that can be matched to scenarios in Figure 5.1. For instance, the initial propagation shown in scenario *r.1* sets the root feature to *true*. This causes scenario *c.2* to be matched since a parent feature (in this particular case, the "root feature") has been set to *true*. As a result, scenario *c.2* sets all root's mandatory child features to *true*. Once again, if those mandatory features have children scenario *c.2* is matched and propagation continues until no more mandatory child features are found.

The propagation algorithm named *FT-propagate* is shown on page 76 (see Algorithm 6). It takes a feature f and a Boolean value v as parameters and returns a list of tuples $< f_p, v_p >$, where f_p is a feature and v_p is a Boolean value assigned to f_p, representing the features that have been instantiated as a result of propagating the assignment $f = v$. If a conflict is found the function returns an error. Operation *FT-propagate* examines three disjoint "propagation spaces" in the feature tree relative to feature f. Figure 5.2 shows these propagation spaces for a given feature P ($f = P$) named AS, DS and GS corresponding, respectively, to the anscestor of P and their children (excluding P), the descendants of P, and the siblings of P within the feature group and their descendants. That is, $AS = \{A, B\} \cup T1 \cup T2$, $DS = T3$, and $GS = \{Y, Z\} \cup T4 \cup T5$. If feature P is not a grouped feature GS is the empty set.

The algorithm starts by checking if feature f is instantiated to a value different than v. If that is the case a conflict has been found and an error is raised (lines 1-3). Next, the current state of the feature tree is saved (line 4). In line 5, the list of instantiated features in the feature tree is retrieved by function *FT-get-instantiated-features* and stored in variable I. If the value to propagate is *true* auxiliary functions *FT-prop-trueAS*, *FT-prop-trueDS*, *FT-prop-trueGS* are called to propagate the assignment on spaces AS, DS and GS, respectively (lines 6-9). Otherwise, functions *FT-prop-falseAS*, *FT-prop-falseDS*, and *FT-prop-falseGS* are called to address propagations for the assignment $P =$ *false* (lines 10-13). For the complete implementation of the FT-prop* auxiliary functions please refer to Appendix A. If a conflict is found during the propagation process by any of the *FT-prop* auxiliary functions the feature tree is restored to its initial state and an error is returned (lines 15-18). Instead, if propagation succeeds without errors variable I is updated to contain only the features that have been instantiated during propagation (line 19). For each feature f_p in I a corresponding tuple $\langle f_p, v_p \rangle$ is created, where v_p is the value assigned to feature f_p during propagation, and added to the initially-empty set of tuples T (lines 20-23). The list of tuples is then returned (lines 24).

Functions *FT-assign* and *FT-propagate* should be used in combination. While the former assigns a value to a feature the latter propagates the assignment in the feature tree to enforce consistency. The following code fragment illustrates the case in which a feature f_1 is assigned *true*. If the assignment does not cause any conflicts it is further propagated in the tree. The "assign and propagate" strategy is useful to keep the feature tree in a consistent satisfiable state as will be discussed next.

```
...
if (FT-assign($f_1$,true)) then
  FT-propagate($f_1$,true)
else
  Error: assignment conflicts!
end if
```

Algorithm 6 Propagates a variable assignment throughout the feature tree

Inputs:
f: propagation starting point; f is a feature assigned v
v: truth value assigned to f that starts the propagation
Output:
The list of features instantiated by propagation and their respective truth values

```
Function FT-propagate($f$:feature,$v$:Boolean)
:$\langle feature, Boolean\rangle\{\}$
 1: if ($f$ is instantiated AND FT-get-value($f$) $\neq v$ )) then
 2:    raises a conflict error!
 3: end if
 4: FT-save-state($ft$-$state$)
 5: $I$ = FT-get-instantiated-features()
 6: if ($v = true$) then
 7:    FT-prop-trueAS($f$)
 8:    FT-prop-trueDS($f$)
 9:    FT-prop-trueGS($f$)
10: else
11:    FT-prop-falseAS($f$)
12:    FT-prop-falseDS($f$)
13:    FT-prop-falseGS($f$)
14: end if
15: if (a conflict has been found in any of the FT-prop* functions) then
16:    FT-restore-state($ft$-$state$)
17:    Error: assignment conflicts during propagation!
18: end if
19: $I$ = FT-get-instantiated-features() $- I$
20: $T = \{\}$
21: for (each feature $f_p$ in $I$) do
22:    add $\langle f_p$, FT-get-value($f_p$$\rangle$ to $T$
23: end for
24: return $T$
```

5.1.4 Checking satisfiability

Some properties that support checking the satisfiability of feature trees are discussed next. We assume that referred feature trees conform to the meta-model in Figure 2.2.

Property 5.1.1 *Let f be a feature in the feature tree. Whenever f is assigned false so must be all its descendants to satisfy the relations in the feature tree.*

Proof Every child feature c holds an implication relation with its parent feature p ($c \rightarrow p$) as depicted in Table 2.1. Therefore, the equivalent relation ($\neg p \rightarrow \neg c$) also holds which means that if p is *false* so must be c and all other p's child features. If this relation is enforced recursively for each of p's children it is straightforward to conclude that whenever p is *false* all its descendants must also be *false.* ▮

Property 5.1.2 *Feature trees conforming to the meta-model in Figure 2.2 (page 13) are always satisfiable.*

Proof The following procedure always finds a solution for an uninstantiated feature tree. Start at the root node r in pre-order traversal. Set r to *true* to satisfy the formula that requires the root of the feature tree to be always *true* (first row in Table 2.1). Visit each child c of the root node. If c is optional, the node is skipped and the pre-order traversal backtracks to examine the remaining child features of r. When c is skipped it is set to *false* along with all its decendants. This satisfies parent-child relation ($c \rightarrow r$) since c is *false* and r is *true.* In addition, it satisfies property 5.1.1. If c is mandatory, set c to *true* and apply the procedure recursively to each of c's child features. This satisfies relation ($c \leftrightarrow r$) since c and r are *true.* Finally, if c is an inclusive-OR ([1..*]) or exclusive-OR ([1]) feature group set one arbitrary feature g in the group to *true* and all others to *false* and repeat the procedure recursively to each of g's child features. This satisfies the group cardinality relation. The procedure stops when there are no more child features to examine and the root feature has been reached again. Notice that the procedure adds the minimum number of features to the solution. ▮

We developed a recursive algorithm called *FT-min-conf* (see Algorithm 7) that follows the procedure described previously to find the minimum valid configuration for a feature tree. The root of the feature tree f and a variable named $conf$ are passed as parameters to the algorithm. Variable $conf$ is initially an empty set that

Algorithm 7 Find the minimum configuration for the feature tree rooted by f

Inputs:
f: root of the feature model
$conf$: solution containing only *true*features

Function **FT-min-conf**($f : feature, conf : feature\{\}$)

```
1: if ($f$ is the root feature) then
2:     add $f$ to $conf$
3: end if
4: for (each child $c$ of $f$ ) do
5:     if ($c$ is optional) then
6:         skip it...
7:     else if ($c$ is mandatory) then
8:         add $c$ to $conf$
9:         FT-min-conf($c$, $conf$)
10:    else if ($c$ is a feature group) then
11:        $g = 1^{st}$ child of $c$
12:        add $g$ to $conf$
13:        FT-min-conf($g$, $conf$)
14:    end if
15: end for
```

will store the features added to the configuration. If f is the root feature, it is added to conf ($r = true$). Next, each child feature c of f is visited in pre-order (line 4). Optional child features are skipped and not added to $conf$. Mandatory child features are automatically added to $conf$ (lines 7-8) and a recursive call is made to examine each of their children (line 9). If c is a feature group the first grouped feature g is added to $conf$ and the others are skipped (lines 10-12). A recursive call is made to examine each of g's children (line 13). All skipped features are assumed *false* and thus not added to $conf$.

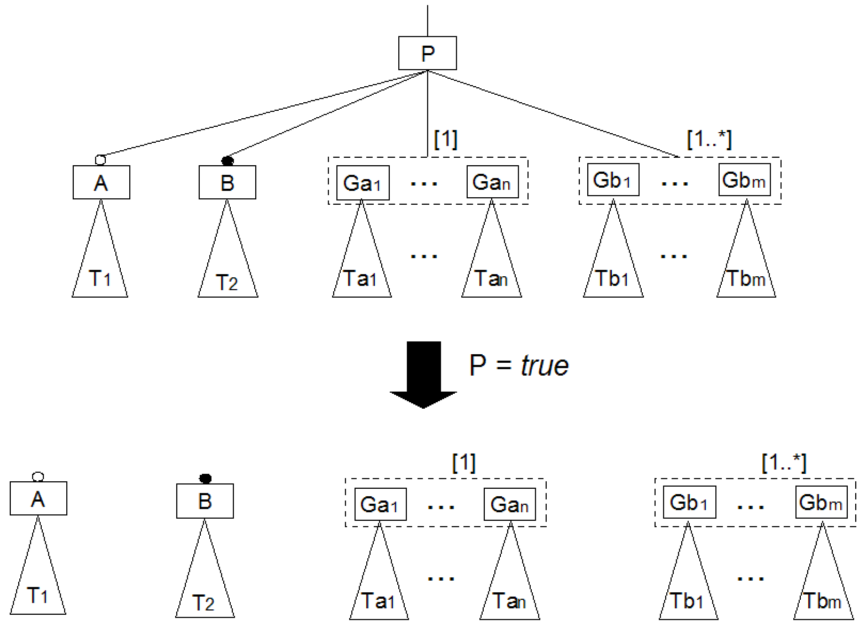


Figure 5.3: Subtree tree rooted by P is split into four independent formulas when P is assigned *true*

Property 5.1.3 *Let f be a parent feature in the feature tree. Whenever f is assigned true the formulas in the tree rooted by f are split into independent sets of formulas that can be processed separately.*

Proof Consider the feature tree fragment shown in the top of Figure 5.3 in which a parent feature P, its children A, B, an exclusive-OR group containing n features, and an inclusive-OR group containing m features are depicted. The semantics of P's tree can be expressed by the following formula f:

$f =$
$\quad \mathrm{T}_1 \wedge T_2 \wedge$
$\quad \mathrm{Ta}_1 \wedge \ldots \wedge Ta_n \wedge$
$\quad \mathrm{Tb}_1 \wedge \ldots \wedge Tb_m \wedge$
$\quad (\mathrm{A} \rightarrow P) \wedge$
$\quad (\mathrm{B} \leftrightarrow P) \wedge$
$\quad (\mathrm{P} \leftrightarrow (Ga_1\ xor\ \ \ldots\ \ xor\ Ga_n)) \wedge$
$\quad (\mathrm{P} \leftrightarrow (Gb_1 \vee \ldots \vee Gb_n))$

Once P is assigned *true* formula f is split into four formulas f_1, f_2, f_3, and f_4 that share no variables and thus can be processed separately as shown next:

$f_1 = T_1$

$f_2 = T_2 \wedge (B)$

$f_3 = Ta_1 \wedge \ldots \wedge Ta_n \wedge (Ga_1 \ xor \ \ldots \ xor \ Ga_n)$

$f_4 = Tb_1 \wedge \ldots \wedge Tb_n \wedge (Gb_1 \vee \ldots \vee Gb_m)$

Formula f is now a conjunction of independent formulas, i.e., $f = f_1 \wedge f_2 \wedge f_3 \wedge f_4$ as shown in the bottom of Figure 5.3. ∎

Notice that in the case of mandatory child features such as B in Figure 5.3 the splitting process continues to be applied lower in the tree as B must be *true* to satisfy formula f_2. This causes B's subtree to break up into another set of independent formulas again. Yet, in the case of feature groups the subtrees rooted by grouped features (e.g., $Ta_1, \ldots, Ta_n, Tb_1, \ldots, Tb_m$) are still connected by the cardinality relation in the group. However, it is possible to process each of those subtree formulas separately and combine the results obtained in accordance with the cardinality relation. Property 5.1.3 can be used to improve the performance of some reasoning algorithms for feature trees, e.g., to count the solutions in the feature tree as will be shown in Section 5.1.5.

Property 5.1.4 *A satisfiable feature tree remains satisfiable if any truth value is assigned to one of its uninstantiated features and propagated. In addition, propagation always succeeds, i.e., a conflict is never raised.*

(*) For a graphical illustration of a scenario of the proof below please refer to Figure 5.4 on page 84.

Proof Let p be an uninstantiated feature of a satisfiable feature tree other than the root node. We will show that when p is assigned any truth value and the assignment is propagated in the feature tree the formulas in the tree are partitioned into independent sets of satisfiable formulas, i.e., formulas that do not share any variables. Hence, the conjunction of these sets of formulas yields a satisfiable formula that corresponds to the feature tree after p's assignment.

Assumption 1: $(p = \boldsymbol{true})$

First, let us assume that p is assigned *true* and that the assignment is propagated in the feature tree.

Assumption 1.1: (p is a parent feature) Now, let us assume that p is a parent feature and that c is one of p's child features. We know that if p is assigned *true* all p's parent-child relations, including the relation between p and c, are satisfied. In particular, the tree rooted by p is split into a set of independent formulas (property 5.1.3). In assumptions 1.1.* below, we show that these independent formulas are satisfiable after p is assigned *true*. As a result, their conjunction is also satisfiable.

Assumption 1.1.1: (c is an optional child feature of p)

Let us assume that c is an optional child feature of p. This corresponds to scenario *c.2* illustrated in Figure 5.1 ($p = P$ and $c = S$). When P is assigned *true* the parent-child relation between P and S (i.e. $S \rightarrow P$) is satisfied. As a result, the subtree rooted by S is disconnected from the feature tree, i.e., becomes an independent branch (formula). None of the features in this subtree, including S, are *true* otherwise propagation would have already set feature P to *true* which violates the initial assumption that p is initially uninstantiated. Therefore, one way to satisfy the relations in this subtree would be to set all its features to *false*.

Assumption 1.1.2: (c is a mandatory child feature of p)

Let us assume that c is a mandatory child feature of p. This corresponds to scenario *c.2* illustrated in Figure 5.1 ($p = P$ and $c = T$). We know that T must be uninstantiated otherwise propagation would have already set P to the same truth value as T which violates the initial assumption that p is initially uninstantiated. When P is assigned *true* T is propagated to *true* in order to satisfy the parent-child relation between P and T (i.e. $T \leftrightarrow P$). As a result, the subtree rooted by T is disconnected from the feature tree, i.e., becomes an independent branch (formula). Hence, assumptions 1.1.* are applied recursively to T, i.e., $p = T$, in order to verify the satisfiability of the T's subtree. In this context, the purpose of mandatory child features is to propagate the *true* assignment of their parent features down in the feature tree and thus split the tree into several independent sets of formulas.

Assumption 1.1.3: (c is a grouped child feature of p)

Let us assume that c is a grouped child feature of p. This corresponds to scenarios *c.3* and *c.4* illustrated in Figure 5.1 ($p = P$ and $c = G_1, G_2, \ldots, G_n$) addressing both group cardinality cases. In scenario *c.3* at least two grouped features are uninstantiated and none are *true*. In this case, if P is *true* the group relation can be satisfied by falsifying all grouped features but one, say G_i. Hence, we can assign *true* to G_i and apply assumptions 1.1.* to G_i recursively, i.e., $p = G_i$, to prove the satisfiability of the feature tree. In scenario *c.4*, all grouped features are *false* but one, say G_1. Hence, if we assign *true* to G_1 the group relation is satisfied and we can apply assumptions 1.1.* to G_1 recursively, i.e., $p = G_1$, to prove the satisfiability of the feature tree.

Assumption 1.2: (p is a grouped feature)

Let us assume that p is a grouped feature. This corresponds to scenarios *g.1* and *g.2* illustrated in Figure 5.1 ($p = P$) addressing both group cardinality cases. If P is *true* the group cardinality is satisfied immediately in both scenarios. In addition, in scenario *g.2* containing an exclusive-OR group, propagation will cause all grouped features and their corresponding subtrees to be *false*.

Assumption 1.3: (p is a child feature)

Let us assume that p is a child feature. This corresponds to scenario *p.1* illustrated in Figure 5.1 ($p = P$). If p is assigned *true* propagation will cause p's parent, i.e., feature Q, to be *true* to satisfy relation $P \rightarrow Q$. Notice that Q cannot be *false* otherwise P would have been already set to *false* by propagation which violates the initial assumption that p is initially uninstantiated. Since Q is assigned *true*, assumptions 1.1.*, 1.2 and 1.3 need to be applied to Q recursively, i.e., $p = Q$, to prove the satisfiability of the feature tree.

Assumption 2: ($p =$ *false*)

Now, let us assume that p is assigned *false* and that the assignment is propagated in the feature tree.

Assumption 2.1: (p is a parent feature) Now, let us assume that p is a parent feature and that c is one of p's child features. This corresponds to scenario *c.1* illustrated in Figure 5.1 ($p = P$). We know that when P is assigned *false* propagation will set all of P's descendants to *false* in order to satisfy the relations in the subtree rooted by P (property 5.1.1).

Assumption 2.2: (p is a child feature)

Let us assume that p is a child feature.

Assumption 2.2.1: (p is an optional feature)

Let us assume that p is an optional child feature. This corresponds to scenario *p.2* illustrated in Figure 5.1 ($p = P$). We know that P's parent, i.e., feature Q, is either *true* or uninstantiated otherwise propagation would have already set P to *false* which violates the initial assumption that p is initially uninstantiated. If we assign *false* to P the relation between P and Q (i.e. $P \rightarrow Q$) is satisfied.

Assumption 2.2.2: (p is a mandatory feature)

Let us assume that p is a mandatory child feature. This corresponds to scenario *p.3* illustrated in Figure 5.1 ($p = P$). We know that P's parent, i.e., feature Q, is uninstantiated otherwise propagation would have set P to the same truth value as Q which violates the initial assumption that p is initially uninstantiated. If we assign *false* to P propagation will assign *false* to Q to satisfy relation $P \leftrightarrow Q$. In addition, assumptions 2.1.*, 2.2 and 2.3 need to be applied to Q recursively, i.e., $p = Q$, to prove the satisfiability of the feature tree.

Assumption 2.2.3: (p is a grouped feature and p's parent is uninstantiated)

Let us assume that p is a grouped feature and p's parent is uninstantiated. This corresponds to scenarios *p.4* and *p.5* illustrated in Figure 5.1 ($p = P$) addressing both group cardinality cases. In scenario *p.4*, Q is not affect by P's assignment as at least one grouped feature remains uninstantiated and all others are either uninstantiated or *false*. Hence, the parent-child relation between P and Q is satisfied. In scenario *p.5*, all grouped features are previously *false* what causes the assignment to P to propagate up in the tree and set Q to *false* in order to satisfy the relation $P \rightarrow Q$. In addition, assumptions 2.1, 2.2.*, and 2.3 need to be applied to Q recursively, i.e., $p = Q$, to prove the satisfiability of the feature tree.

Assumption 2.3: (p is a grouped feature and p's parent is *true*)

Let us assume that p is a grouped feature and p's parent is *true*. This corresponds to scenario *g.3* illustrated in Figure 5.1 ($p = P$) addressing both group cardinality cases. Since at least one grouped feature is already *true* in the group, P's assignment will cause no impact in the group relations which remains satisfiable.

Considering that whenever p is assigned any truth value the feature tree formulas are split into independent satisfiable sets of formulas, we can conclude that the conjunction of these formulas is satisfiable. In addition, we have seen that propagation is either unnecessary or prunes the values of uninstantiated features. Hence, the feature tree remains satisfiable after p's assignment and propagation always succeeds. $\blacksquare$

Figure 5.4 illustrates a possible scenario of the proof above. A feature tree is shown at the top of the figure rooted by feature R that has value *true* (R=1). The other features remain uninstantiated as indicated by the symbol "?". An assignment is made in the tree to instantiate feature F to *true* (F=1). This causes the splitting of the subtrees in the tree as shown at the bottom of the figure. The

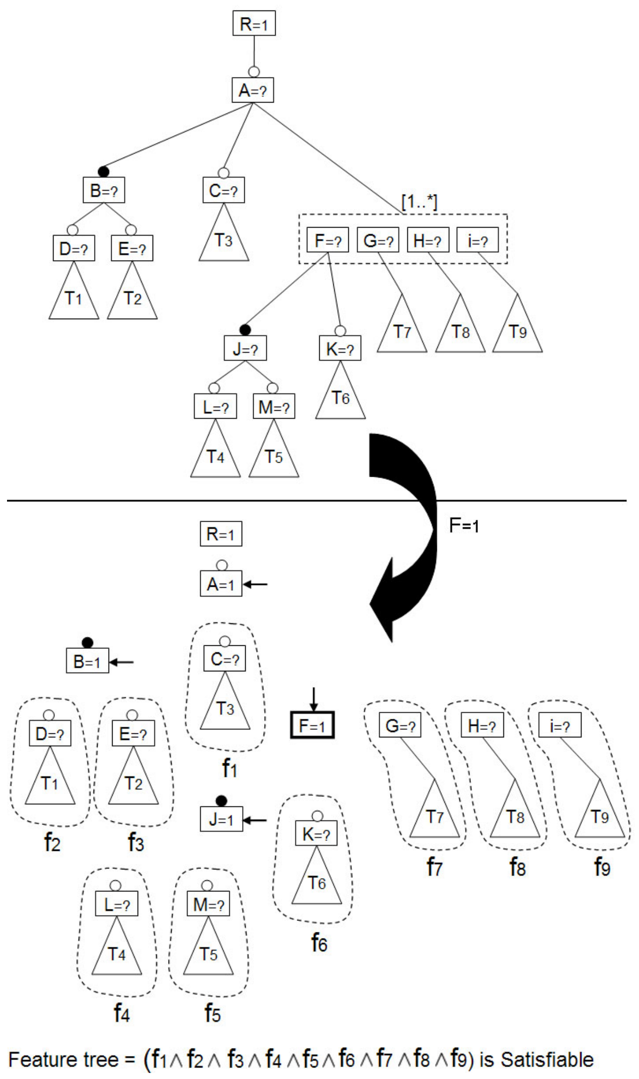


Figure 5.4: The feature tree (top) is split into a set of nine independent satisfiable formulas (bottom) after feature F is assigned *true*.

assignment to F causes the relations between this feature and its children (features J and K) to be satisfied thus detaching the subtrees rooted by J and K from the tree. In addition, feature J is assigned *true* via propagation as it is a mandatory child feature and the splitting proceeds to J's subtrees. Hence, three subtrees are disconnected from the original tree as illustrated by the dashed lines in f_4, f_5, and f_6. Similarly, the assignment to F satisfies the inclusive-OR group relation ([1,*]) that requires at least one grouped feature to be *true*. This causes each of the subtrees rooted by a grouped feature to be disconnected from the tree as shown in f_7, f_8, and f_9. Finally, F is propagated up in the tree which causes parent feature A to be assigned *true*. Similarly, A's assignment is propagated which instantiates B

to *true* and causes a splitting of some of A's subtrees as shown in f_1, f_2, and f_3. As a result, the original feature tree (top) has now four new instantiated features (F, A, J, and B as illustrated by arrows in the figure) and nine independent subtrees (or formulas). In fact, the subtrees share no variables.

Notice that each independent subtree in Figure 5.4 is rooted by an uninstantiated feature. For instance, subtrees f_1 and f_2 are rooted by features C and D, respectively. The fact that the root of each subtree is uninstantiated shows that none of the features in these subtrees are *true* otherwise the root would have already been assigned *true* through propagation. Therefore, a possible solution to satisfy each subtree is to falsify all of its features. We can then conclude that the conjunction ($f_1 \wedge f_2 \wedge f_3 \wedge f_4 \wedge f_5 \wedge f_6 \wedge f_7 \wedge f_8 \wedge f_9$) that represents the feature tree after the assignment F=*true* is satisfiable since it is a conjunction of independent yet satisfiable formulas. Although the figure illustrates just one possible scenario of the proof previously discussed, it helps vizualizing the splitting of the formulas in a feature tree when one of its features is instantiated.

Property 5.1.4 can be applied in at least three practical contexts. First, we can conclude that the propagation procedure for feature trees (Algorithm 6) enforces global (as opposed to local) consistency since the feature tree remains satisfiable after each propagation step. Therefore, support for interactive configuration of feature trees is straightforward. That is, the feature tree can be configured backtrack-free by simply selecting (*true*) or deselecting (*false*) uninstantiated features and propagating these decisions one at a time. Recall that for general Boolean formulas more robust solutions based on BDDs are required to support interactive configuration. Second, property 5.1.4 supports the construction of proofs for the hardness of SAT instances derived from feature trees. In particular, it can be shown that SAT solvers take only linear time in the size of the feature tree to check the satisfiability of the Boolean formula derived from the tree (see details in Section 5.3). Third, property 5.1.4 can be used to build an interplay between the *FTRS* and a constraint solver system to support the development of efficient hybrid algorithms to reason on feature models. This topic is discussed in the next section when we introduce a reasoning system for feature models.

Operation *FT-is-satisfiable* that checks whether a feature tree is satisfiable is described below (Algorithm 8). The algorithm simply returns *true* as it relies on properties 5.1.2 and 5.1.4. That is, an initially satisfiable feature tree remains satisfiable whenever a feature assignment is successfully propagated. Also, note that for the cases in which propagation fails the feature tree is restored to a consistent satisfiable state.

Algorithm 8 Checks the satisfiability of the feature tree rooted by f

Inputs:
f: root of the feature tree
Output:
always *true* as the satisfiability of the tree is enforced by other functions

```
Function FT-is-satisfiable($f$ : $feature$)
1: return true
```

5.1.5 Counting solutions

Section 3.1.4 discussed the importance of computing the number of valid configurations in a feature model to support other metrics such as the variability of a product line. However, counting the solutions of a constraint problem can be an extremely time-consuming task for SAT solvers as it usually requires the solver to iterate over all the solutions in the problem. In practice, this can make the use of SAT solvers prohibitive in this context. Fortunately, feature trees exhibit properties that allow for a quick counting of their solutions without requiring an exhaustive enumeration of those as discussed next.

We developed an algorithm called *FT-count-sol* (see Algorithm 9) as part of the *FTRS* system to count the number of available solutions in uninstantiated or partially-instantiated feature trees. The algorithm is recursive and takes advantage of some of the feature tree properties described earlier (properties 5.1.1 and 5.1.3) as will be shown next. The algorithm starts by visiting a given feature f passed as an input parameter and continues by traversing the feature tree in depth-first order. Feature f usually corresponds to the root of the feature tree when the algorithm is first called (non-recursive call). If feature f is *false* (line 1) then, according to property 5.1.1, there is only one possible configuration for f in which all its descendants are *false* (line 2). Otherwise, if f is not *false* and f is an exclusive-OR feature group (line 5), each of f's children have their configurations counted recursively and the results obtained are added (lines 7-9). The reason for adding the configurations is related to the fact that only one grouped feature can be *true* in the group while all others are *false* as per the group cardinality [1]. If f is not an exclusive-OR feature group but has children (line 11), then f child's configurations are recursively computed and multiplied (lines 12-14). Multiplication is applied in this context since we need to account for all possible combinations of valid

Algorithm 9 Count the number of solutions of the (possibly partially instantiated) feature tree rooted by f

Inputs:
f: root of the feature tree
Output:
number of valid configurations in the feature tree rooted by f

```
Function FT-count-sol($f$ : $feature$) : integer
 1: if ($f = false$ ) then
 2:     return 1
 3: end if
 4: $count\_conf = 1$
 5: if ( $f$ is an Exclusive-OR Feature Group) then
 6:     $count\_conf = 0$
 7:     for (each child $c$ of $f$) do
 8:         $count\_conf = count\_conf$ + FT-count-sol($c$) - 1
 9:     end for
10: else
11:     if ($f$ has children) then
12:         for (each child $c$ of $f$) do
13:             $count\_conf = count\_conf \times$ FT-count-sol($c$)
14:         end for
15:         if ($f$ is Optional or Grouped and $f \neq true$) then
16:             $count\_conf = count\_conf + 1$
17:         else if ($f$ is an Inclusive-OR Feature Group) then
18:             $count\_conf = count\_conf$ - 1
19:         end if
20:     else
21:         if ($f$ is Optional or Grouped and $f \neq true$) then
22:             $count\_conf = count\_conf + 1$
23:         end if
24:     end if
25: end if
26: return $count\_conf$
```

configurations represented in each of f's subtrees. In addition, if f is an optional or a grouped feature and is uninstantiated (in this case we just check condition $f \neq true$ since we have already checked that $f \neq$ *false*)(line 15), we add one more configuration to variable $count\_conf$ that stores the number of valid configurations for feature f (lines 15-16). This accounts for the case where f and its decendants are all *false*. Instead, if f is an inclusive-OR feature group we deduct one configuration for the opposite reason, i.e., there is no such case where all grouped features are *false* when their parent feature is *true* (line 17-19). Notice that in all the cases where f is *true* the algorithm has taken advantage of property 5.1.3 to compute the valid configurations for f by recursively combining the valid configurations of each of f's *independent* subtrees. In the case of feature groups, group cardinalities had to be considered as well. Finally, if f does not have children but is optional or grouped feature and uninstantiated (line 21), we add one to variable $count\_conf$ to account for the case where f is *false* (lines 21-23). This will cause $count\_conf$ to evaluate to two to represent the two possible assignments to f, i.e., *false* and *true*. For all other cases, f can only be *true* and thus $count\_conf$ remains one. Line 24 returns the total number of valid configurations for f.

In the worst-case, when none of the features in the feature tree are instantiated, the algorithm *FT-count-sol* visits each feature at most once in depth-first search. As a result, the worst-case complexity of the algorithm is linear in the number of nodes n in the feature tree ($O(n)$). Yet, a constraint solver would perform an exponential number of steps to perform the same operation ($O(2^n)$) as the solver would have to find each solution.

5.1.6 Enumerating solutions

Enumerating the solutions of a feature tree corresponds to identifying the existing *valid* product configurations. Here the term *enumerating* replaces the usual term *searching* used by constraint solvers based on backtracking-search. Searching usually entails several cycles of constraint propagation and backtracking in order to find solutions. On the contrary, enumerating solutions does not require propagation or backtracking but rather combining intermediate results into complete solutions. For that reason, *enumerating* typically involves less algorithmic steps potentially translates to faster algorithms.

It is possible to enumerate the solutions of a feature tree efficiently by exploring properties 5.1.1 and 5.1.3. That is, since the subtrees of a given feature f are independent when f is *true* (5.1.3), the solutions of each subtree can be recursively

Algorithm 10 Returns an iterator object that can be used to enumerate the solutions of the feature tree rooted by feature f

```
Inputs:
 $f$: root of the feature tree for which solutions will be enumerated
Output:
 $f$: solution iterator object for the feature tree rooted by $f$

Function FT-create-sol-iterator($f$ : $feature$)
: FT-sol-iterator
1: $iterator$ = FT-create-iterator-object($f$)
2: for (each child $c$ of $f$) do
3:    FT-add-child-iterator($iterator$,FT-create-sol-iterator($c$))
4: end for
5: if ($f$ is a feature group) then
6:    FT-set-iterator-combination($iterator$,FT-min-cardinality($f$),FT-max-
      cardinality($f$))
7: end if
8: return $iterator$
```

(bottom-up) combined until the root node is reached. In addition, an extra solution is considered for the case where f is optional and assigned *false* (5.1.1). Feature group solutions are enumerated simply by combining the solutions of each subtree rooted by their grouped nodes. Unlike other parent feature nodes, feature groups have the solutions computed by combining their subtrees' solutions according to the cardinality relations specified in the group.

Even though an algorithm to exhaustively enumerate all possible solutions in a feature tree is straightforward to implement it is often inefficient and useless in practice. A better approach is to consider algorithms that enumerate one solution at a time. In the following, we provide an implementation of the latter using iterators to navigate through available solutions in the feature tree. Since the corresponding implementations are lengthy we focus on the main algorithms.

Algorithm 10 takes a feature f as input and creates a solution iterator for the feature tree rooted by f. Line 1 of the algorithm creates an iterator object for f. In addition, if f has children an iterator is created for each of f's child feature and associated with f's iterator (lines 2-4). This allows computing the solutions in f's tree incrementally by invoking the child iterators. Moreover, in case f is a feature group (line 5) operation *FT-set-iterator-combination* is called to indicate that f' subtrees need to be combined in several different ways according to the cardinality of the group (line 6). Function *FT-min-cardinality(f)* and *FT-max-cardinality(f)* return the cardinality lower and upper bounds. For instance, consider an inclusive-OR group ([1..*]) containing features f, g, and h. In order to enumerate the solutions in the feature group the following subtree combination sets need to be considered: $\{f\}$, $\{g\}$, $\{h\}$, $\{f,g\}$, $\{f,h\}$, $\{g,h\}$, and $\{f,g,h\}$. For each set, the solutions of the subtrees rooted by its elements are combined. Line 8 of Algorithm 10 returns a reference to the solution iterator object created for f.

Once the iterator object is created it can be used to iterate over f's solutions. Function *FT-has-next-sol* checks whether there are still solutions that can be enumerated for a given subtree in the feature tree. It can be used to iterate over a set of solutions safely until no more solutions are available. Algorithm 11 implements this function. It takes a solution iterator created by function *FT-create-sol-iterator* as input and returns *true* if there are still solutions to enumerate or *false* otherwise. In line 1, the feature f associated with the iterator object is retrieved. If f is an optional uninstantiated feature for which *false* is yet to be enumerated the function returns *true* indicating that there is a solution to enumerate (lines 2-3). Instead, if f is any kind of leaf feature (no children) and value *true* has not been enumerated the function also returns *true* (lines 4-6). Yet, if none of the previous cases are true f's children need to be examined (line 7). If f has no children, then all its values (*false* and *true*) have already been already enumerated and the function returns *false* (line 18). Otherwise, if f is not a leaf node, function *FT-prepare-child-iterators* (listed in appendix A) takes the list of child iterators of a given iterator and prepares those for the next enumeration cycle. The function returns *true* whenever there are combinations of child iterator elements to be processed (and thus solutions available). The child iterators are processed last-to-first and reset whenever they have been fully processed. When the first child iterator is processed and reset the operation returns *false* indicating that the elements of the child iterators have been fully combined. Therefore, if feature f in *FT-has-next-sol* is not a feature group (line 8) its child iterators are retrieved (function *FT-get-child-iterators*) and processed by function *FT-prepare-child-iterators*. The result of the function is returned. However, if f is a feature group other combinations of child iterators have to be considered. In line 11, function *FT-get-current-iterator-combination* returns the current child iterator combination set and stores in the *child-iterators* list. The list of child iterators is processed by function *FT-prepare-child-iterators* until either a combination is found containing yet-to-be-enumerated solutions or the conclusion

Algorithm 11 Returns *true* if the feature iterator object still has a solution to enumerate or *false* otherwise

Inputs:
iterator: the solution iterator object for feature trees
Output:
returns *true* if the feature iterator object still has a solution to enumerate or *false* otherwise

Function **FT-has-next-sol**(*iterator*:FT-sol-iterator) : Boolean

```
 1: f = FT-get-iterator-feature(iterator)
 2: if (f is uninstantiated optional AND false has not been enumerated) then
 3:     return true
 4: else if (f does not have children AND true has not been enumerated) then
 5:     return true
 6: end if
 7: if (FT-has-child-iterators(iterator)) then
 8:     if (f is NOT a feature group ) then
 9:         return FT-prepare-child-iterators(FT-get-child-iterators(iterator))
10:     else
11:         child-iterators = FT-get-current-iterator-combination(iterator)
12:         while (child-iterators <> NIL AND
            NOT FT-prepare-child-iterators(child-iterators)) do
13:             child-iterators = FT-get-next-iterator-combination(iterator)
14:         end while
15:         return (child-iterators <> NIL)
16:     end if
17: end if
18: return false
```

that all combinations have been processed and enumerated (lines 12-14). In the former case, the function returns *true*, otherwise it returns *false* (line 15).

Finally, operation *FT-next-sol(iterator)* retrieves the individual solutions of the iterator object associated with a given feature in the feature tree. It is implemented by Algorithm 12. In the algorithm, variable *sol* is created to store features that are part of a solution in the iterator. Therefore, *sol* only stores the features that are assigned *true* in the solution. The variable is initially set to *NIL* (line 1). If there are no more solutions available according to the *FT-has-next* function (line 2) *NIL* is returned (line 19). Otherwise, feature f associated with the *iterator* object is retrieved (line 3). If f is any kind of feature that has been assigned *false* and *false*has not been enumerated for this feature yet, then *sol* is assigned the empty set ($sol = \{\}$) to indicate that whenever f is *false* so all its children are *false* (property 5.1.1) (lines 5-6). If *false* has already been enumerated *NIL* is returned as no more solutions are available for the subtree considering that f is *false*. If f is rather optional and uninstantiated, and *false* has not been enumerated *sol* is also assigned an empty set ($sol = \{\}$) for the same previous reasons (lines 7-8). However, because in this case f is uninstantiated there is still a need to check the case whether f is *true*.

In fact, for all cases in which a feature is assigned *true* or needs to be enumerated for this value the feature is included in the solution (line 10) and, if that is the case, its children are examined (lines 11-19) (property 5.1.3). If f is not a feature group its child iterators are retrieved by auxiliary function *FT-get-child-iterators*, otherwise another auxiliary function *FT-get-current-iterator-combination* has to be used to recover the current child iterator combination set to be stored in *child-iterators*. Once the right combination set of child iterators is known, a unique solution needs to be extracted from the set. Function *FT-get-child-iterators-sol* (see appendix A) is used for that purpose (line 15). It extracts the next solution in a list of child iterators by traversing the iterators list in first-to-last order. For all child iterators the current solution is retrieved (function *FT-current-sol*) except for the last iterator in which the next solution (function *FT-next-sol*) is considered. Remember that it is always guaranteed that a solution will be found by *FT-get-child-iterators-sol* since function *FT-has-next-sol* has returned *true* (line 2 of Algorithm 12). The solution returned by *FT-get-child-iterators-sol* is then incorporated (union set) into *sol* (line 15). The unique solution set *sol* is then returned (line 19).

Functions *FT-create-sol-iterator*, *FT-has-next-sol*, and *FT-next-sol* should be used together to enumerate the solutions of a feature tree iteratively. The following code fragment illustrates how these functions can be used together. First, a solution

Algorithm 12 Returns a list of features representing the next solution of the solution iterator object

Inputs:
iterator: the solution iterator object
Output:
returns a list of features representing the next solution of the solution iterator object

Function **FT-next-sol**(*iterator*:FT-sol-iterator) : feature{}

```
1: $sol = NIL$
2: if (FT-has-next-sol($iterator$)) then
3:    $f$ = FT-get-iterator-feature($iterator$)
4:    if ($f$ is instantiated to $false$) then
5:       if ($false$ has not been enumerated) then
            sol = {}
6:       end if
7:    else if ($f$ is an optional uninstantiated feature AND $false$ has not been
      enumerated) then
8:       sol = {}
9:    else
10:      sol = {f}
11:      if (FT-has-child-iterators($iterator$)) then
12:         if ($f$ is NOT a feature group) then
13:            $child\text{-}iterators$ = FT-get-child-iterators($iterator$)
14:         else
15:            $child\text{-}iterators$ = FT-get-current-iterator-combination($iterator$)
16:         end if
17:         $sol = sol \cup$ FT-get-child-iterators-sol($child\text{-}iterators$)
18:      end if
19:   end if
20: end if
21: return $sol$
```

iterator *iterator* is created to enumerate the solutions of the subtree rooted by a given feature f. Next, a loop is introduced to iterate over each solution in the subtree while one can be found. Inside the loop, the solution *sol* is retrieved from the *iterator* object and printed out. The solution sets contain only features assigned *true*, i.e., all non-listed features are assigned *false*.

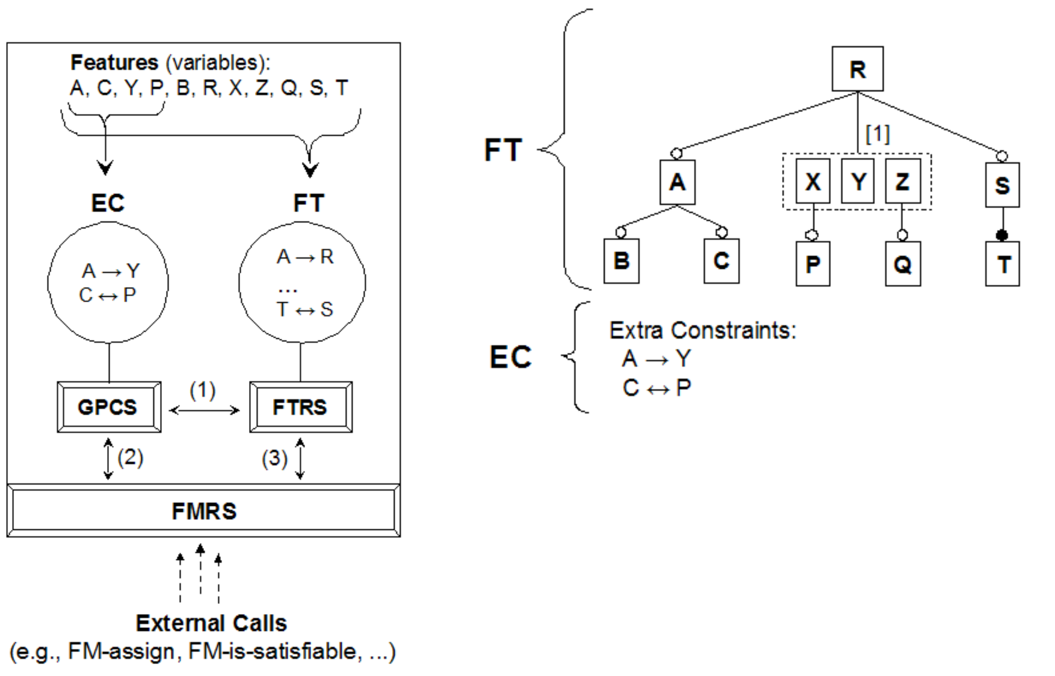
```
...
$iterator$ = FT-create-sol-iterator($f$)
while ( FT-has-next-sol($iterator$) ) do
    $sol$ = FT-next-sol($iterator$)
    print-out($sol$)
end while
...
```

5.2 FMRS: A Hybrid Reasoning System for Feature Models

Up to this point, we have explored several properties of feature trees to develop a reasoning system called *FTRS* that provides efficient operations for reasoning on feature trees. For instance, the satisfiability of feature trees can be checked in constant time and the solutions in the tree can be computed in linear time in the number of features in the tree. However, since most of these properties examined previously do not hold for feature models containing extra constraints but rather to feature trees the *FTRS* cannot be used to reason on feature models. For instance, while it could be proved that feature trees are always satisfiable (property 5.1.2) there is no guarantee that a solution exists for a feature model.

Despite this fact, in this section we show that the properties examined for the *FTRS* can still be advantageous for building a reasoning system for the entire feature model. In the following, we introduce a hybrid reasoning system for feature models called *FMRS* (Feature Model Reasoning System). The system relies on the *FTRS* to handle feature trees and a constraint system to address the extra constraints, and explores an advantageous interplay between these systems. The rationale of the *FMRS* is that it is possible to improve the overall performance of reasoning operations for feature models by taking advantage of domain knowledge through the *FTRS* system.

Next, we depict the archicture of the *FMRS*.



(a) (b)

Figure 5.5: (a) The architecture of the FMRS and (b) the feature model illustrated in the architecture

5.2.1 FMRS Architecture

The *FMRS* is a reasoning system for feature models that provides the same operations shown in Table 5.1 but now applied to the entire feature model. We use prefix "FM" to name the operations in the *FMRS*.

The architecture of the *FMRS* is shown in Figure 5.5(a) along with an illustrative feature model in Figure 5.5(b). Inside the larger rectangle, the two circles represent the formulas in the feature model, i.e., the extra constraints (EC) and the feature tree (FT). Notice that only a subset of the eleven problem variables are referenced by EC formulas, i.e., variables A, C, Y and P. Instead, all variables are referenced in the feature tree. The *GPCS* rectangle represents a general-purpose constraint system (GPCS), for instance, a SAT or CSP solver, that addresses the EC formulas. The other corresponding rectangle represents the *FTRS* system discussed in the previous section that can be used to reason on feature trees. The *FMRS* rectangle in the bottom of the figure is the interface of the *FMRS* system. It represents a facade that hides from external systems the fact that two internal solvers support the major functionalities of the *FMRS* system. Therefore, external calls to the *FMRS* are always made through the facade interface. The combination of two distinguished techniques to address feature model formulas characterizes the *FMRS* as a hybrid system. Notice that the *FMRS* and its internal components communicate through well-defined interfaces represented by labels (2) and (3). In addition, there is an extra communication channel labeled (1) between the two internal solvers. These communication channels play a key role in enforcing the consistency of the solvers as will be shown later. The rationale as well as the strengths of the *FMRS* architecture are explained next as the operations of the system are introduced.

Algorithm 13 Assign feature f the truth value v

```
Inputs:
 $f$: feature to be assigned a value
 $v$: truth value to be assigned to $f$

Function FM-assign($f : feature, v : Boolean$)
 1: FT-assign($f$,$v$)
 2: GP-assign($f$,$v$)
```

5.2.2 Assigning and propagating values

Even though the GPCS and the *FTRS* share some variables, each system keeps its own local variables and a unique key is used to relate shared variables. This is important to enforce the independence of these systems and to centralize their integration in the *FMRS*. In this context, a value assignment to a variable in the *FMRS* is implemented as forward calls to its two internal systems. Algorithm 13 implements the *FM-assign* operation for the *FMRS* in which value v is assigned to feature f. The algorithm simply forwards the value assignment to the internal solvers by calling operations *FT-assign* and *GP-assign* (lines 1-2). These calls are represented in Figure 5.5(a) by communication channels (2) and (3) and are important to enforce the consistency between the *FTRS* and the GPCS. Note that GPCS operations are prefixed by "GP" and represent operations available in most current SAT solver implementations (most likely with different names).

Similarly, propagation calls to the *FMRS* are simply forwarded to the internal solvers. However, unlike value assignments several rounds of updates might be necessary to keep the solvers consistent with each other. That is, the results of

Algorithm 14 Propagates a variable assignment throughout the feature model

Inputs:
f: feature
v: truth value assigned to f that starts the propagation

Function **FM-propagate**($f : feature, v : Boolean$)

```
 1: FT-save-state("ft-state-before-assignment")
 2: GP-save-state("gp-state-before-assignment")
 3: $FT\_tuples = \{\langle f, v \rangle\}$
 4: $GP\_tuples = \{\langle f, v \rangle\}$
 5: $tmp\_tuples = nil$
 6: $solvers\_consistent = false$
 7: while ($solvers\_consistent$ is $false$) do
 8:   $tmp\_tuples = FT\_tuples$
 9:   for (each tuple $\langle f, v \rangle$ in $GP\_tuples$) do
10:     FT-assign($f$,$v$)
11:     $FT\_tuples$ = FT-propagate($f$,$v$)
12:     Eliminate tuples in $FT$ that refer to variables not present in the GPCS
13:     if (assignment conflicts is FTRS) then
14:       FT-restore-state("ft-state-before-assignment")
15:       GP-restore-state("gp-state-before-assignment")
16:       {Error: Assignment Conflict!}
17:     end if
18:   end for
19:   for (each tuple $\langle f, v \rangle$ in $tmp\_tuples$) do
20:     GP-assign($f$,$v$)
21:     $GP\_tuples$ = GP-propagate($f$,$v$)
22:     if (assignment conflicts in GPCS) then
23:       FT-restore-state("ft-state-before-assignment")
24:       GP-restore-state("gp-state-before-assignment")
25:       {Error: Assignment Conflict!}
26:     end if
27:   end for
28:   if ($FT\_tuples = \{\}$ and $GP\_tuples = \{\}$) then
29:     $solvers\_consistent = true$
30:   end if
31: end while
```

every propagation in one of the solvers need to be updated in the other and vice-versa.

Operation *FM-propagate* propagates the assignment of value v to feature f in the *FMRS* by updating the state of its internal solvers repeatedly until either a conflict error is raised or there are no more propagations to carry out. The implementation of *FM-propagate* is shown in Algorithm 14. The algorithm starts by saving the state of the two internal solvers (lines 1-2) prior to any propagations. This allows the solvers to be restored to a consistent state in case of errors. Following, three sets *FT_tuples*, *GP_tuples* and *tmp_tuples* are defined to store propagations values (lines 3-5). The propagation tuples in the *FTRS* (GPCS) are recorded in the *FT_tuples* (*GP_tuples*). The *GT_tuples* set is traversed and each of its tuples that represent propagations performed in the GPCS are used to assign values to variables in the *FTRS* (lines 9-17). Similarly, propagations in the *FTRS* stored in the *FT_tuples* set are forwarded to the GPCS (lines 19-27). Notice that the *FT_tuples* tuples containing references to variables not found in the GPCS are removed from the set (line 12). The propagation loop (line 7) continues until either a conflict arises in one of the internal solvers (lines 13 and 22) or a consistent state is reached (lines 28-30). The *FMRS* is consistent when the propagation sets *FT_tuples* and *GP_tuples* are empty, i.e., there are no more propagations to carry out. If a conflict error is raised the solvers are restored to their original states (lines 14-16 and 23-25). Once again, the propagation calls forwarded to both solvers are represented in Figure 5.5(a) by labels (2) and (3).

5.2.3 Checking satisfiability

Property 5.2.1 *Let FM be a feature model formula obtained by the conjunction of a feature tree FT and an arbitrary propositional formula EC, i.e., $FM = FT \wedge EC$. The formula FM is satisfiable iff there is a solution S in EC that can be successfully propagated in FT.*

Proof First, let us prove the first part of the implication, i.e., whenever a feature model is satisfiable there is a solution in EC that can be successfully propagated in FT. If the feature model is satisfiable there is a solution S that satisfies the model. In addition, S also satisfies EC (considering only EC variables) and FT since S is a solution in FM. Therefore, S represents a solution in EC that can be propagated successfully in FT since it is a solution in FM. Now, lets us prove that if a solution in EC can be successfully propagated in FT the feature model is satisfiable. Let R be a solution in EC and let FT be a satisfiable feature tree. If R can be successfully propagated in FT the remaining formulas represented in FT are satisfiable (property 5.1.4) and so is the conjunction $FM = FT \wedge EC$. Hence, it can be concluded that the feature model is satisfiable and R is part of at least one of its solutions. ▮

As was discussed in Chapter 3, existing techniques for reasoning on feature models consider translating feature models to a specific encoding and using a constraint solver to check the satisfiability of the models. In this approach, the constraint solver takes into account all feature model variables to make decisions during the search procedure. For instance, the solver might decide to assign a given variable v value *true* and propagate. If propagation fails, the solver backtracks and eventually attempts assigning *false* to v. The higher the number of decisions made by the solver the higher the time required to process the operation. This explains why constraint solvers are usually inefficient in counting problem solutions as this operation usually requires dealing with a combinatorial number of decisions.

In this context, property 5.2.1 is interesting as it allows us to develop an alternative SAT procedure for feature models that performs a reduced number of decisions in comparison to constraint solvers. That is, the satisfiability of the feature model can be checked by examining whether a solution in EC can be propagated successfully in FT. In other words, a constraint solver needs only to make decisions on EC variables and attempt to propagate these decisions in FT. Hence, only a subset of the feature model variables need to be examined. In practice, this means that a solution can be found for the feature model and yet several variables in the model remain uninstantiated as the satisfiability of the formula is guaranteed. In fact, some of these variable are never reachable to the solver through propagation which means that they will always remain uninstantiated in the feature model.

This fact can be explored to build an interesting interplay between the *FTRS* and a constraint solver. That is, the solver searches for solutions in EC and attempts to propagate these in FT. Once propagation succeeds, the feature tree is now "pruned" and guaranteed to be in a consistent state. At this time, *FTRS* algorithms can be applied to the pruned tree efficiently. Recall that *FTRS* algorithms always take advantage of feature tree properties to deliver improved efficiency. Ideally, these properties should also hold for pruned trees. For instance, algorithm *FT-count-sol* (Algorithm 9) can count the number of solutions in a partially-instantiated feature tree in linear time in the size of the tree. Therefore, the number of solutions in the feature model can be computed by simply applying this algorithm repeatedly for each solution in EC that can be propagated successfully in FT. In the following, we propose a SAT procedure for the $FMRS$ based on the ideas derived from 5.2.1.

A possible approach to develop the new SAT procedure could be to use a general constraint solver to find complete solutions in the extra constraints and subsequently use function *FT-propagate* in the *FTRS* to verify if the solution can be successfully propagated in the feature tree. In addition, the state of the feature tree could be saved and restored between verifications using functions *FT-save-state* and *FT-restore-state*. Unfortunately, such approach can be very inefficient as it requires a complete solution for the extra constraint formula to be found prior to any checks on the feature tree. In practice, this can lead to several unnecessary search cycles in which a large number of solutions in the EC are unproductively analyzed.

Instead, we propose an iterative search procedure in which partial solutions in the EC are checked against the feature tree relations. Such solutions are either continuously expanded if they do not violate any of the feature tree relations or eliminated otherwise. This approach is more efficient than the one mentioned previously as it eliminates a group of unproductive solutions at once. Yet, the approach can be implemented straightforwardly using a conventional constraint solver and the *FTRS* system. The strategy is to register the *FTRS* as a constraint of the constraint solver. This allows the *FTRS* to be notified about all variable instantiations performed by the constraint solver and to either accept the instantiation if the variable assignment can be properly propagated in the feature tree or otherwise reject the assignment and force the constraint solver to backtrack. The advantage of this approach is that the entire searching infra-structure of the constraint solver can be reused without modifications. The only requirement is that the *FTRS* conform to the constraint interface required by the solver. In addition, the *FTRS* has to be able to save and restore multiple states of the feature tree in order to synchronize assignments and propagations in the tree with the expanding and backtracking moves of the constraint solver.

Many modern constraint systems and SAT solvers (e.g. [23], [13]) provide well-defined event-based interfaces to notify about relevant changes made to the state of the system during the search process. In those systems, constraints are usually implemented as objects that register to system events and thus can directly influence the search process. We are interested in three particular events to build a constraint object for the *FTRS* system to support the SAT procedure described previously. The first two events are related to the expanding and contracting of the search tree. Let us name such events *GP-on-expanding* and *GP-on-contracting*,

Algorithm 15 Interface for handling search tree expanding events

```
Inputs:
 $v$: variable instantiated
 $b$: Boolean value assigned to $v$

Function GP-on-expanding($v : variable, b : Boolean$)
1: FT-save-state($v$)
```

Algorithm 16 Interface for handling search tree contracting events

```
Inputs:
 $v$: variable instantiated
 $b$: Boolean value assigned to $v$

Function GP-on-contracting($v : variable, b : Boolean$)
1: FT-restore-state($v$)
```

respectively. That is, each time an assignment is posted in or retracted from the tree, an event is generated. For instance, in Figure 2.3(b) the assignment $R = 0$ was posted in the search tree at level 1 and later retracted since a conflict was found which forbids the assignment. Constraints should also receive propagation events so that the value assignments performed by the solver can be properly propagated. Let us name this event *GP-on-propagating.* Propagation events allow constraints either to check the validity of a partial solution and eventually instantiate new variables or to find a conflict and raise an error.

Algorithms 15 and 16 represent handlers to the constraint solver events mentioned. These operations simply save and restore the state of the feature tree for each expansion and contraction of the search tree, respectively. Procedure *GP-on-expanding* is called whenever a variable v is instantiated to Boolean value b. Similarly, procedure *GP-on-contracting* is called whenever the assignment of b to v is retracted. As a result, the consistency between the *FTRS* and the constraint solver is enforced.

Operation *GP-on-propagating* is a little more sophisticated. It is an event handler operation triggered by the constraint solver right after the call to *GP-on-expanding* to allow constraints to perform propagation. Algorithm 17 implements the propagation event handler. It starts by assigning truth value b to the feature in

Algorithm 17 Event handler to address constraint solver propagation events

Inputs:
v: variable instantiated
b: truth valued assigned to v

Function **GP-on-propagating**($v : variable, b : Boolean$)

```
1: FT-assign($v$,$b$)
2: $P$ = FT-propagate($v$,$b$)
3: if (an assignment conflict is raised during FTRS propagation ) then
4:    raises a conflict error to force the constraint solver to backtrack
5: else
6:    for (each tuple $\langle variable, value \rangle$ in $P$, variable $\in$ EC) do
7:       GP-enqueue($variable$,$value$)
8:    end for
9: end if
```

the feature tree corresponding to variable v (line 1). Following, the assignment is propagated in the feature tree by calling the *FTRS FT-propagate* operation (line 2). The tuples representing the new instantiations caused by the propagation call in the feature tree are store in set variable P. If a conflict was found during propagation, an error is raised (lines 3-4), otherwise each tuple in P that contains a reference to a variable in the extra constraints is enqueued in the constraint solver for further propagation (lines 6-8). This procedure supports the inverse synchronization process, i.e., that assignments in the feature tree are correctly propagated in the constraint solver.

Algorithm 18 Check the satisfiability of the feature model rooted by f

Inputs:
f: root of the feature model tree
Output:
returns *true* if the feature model is satisfiable or *false* otherwise

Function **FM-is-satisfiable**($f : feature$)

```
1: return GP-is-satisfiable()
```

The constraint system discussed so far is represented by the *GPCS* component in the architecture of the *FMRS* in Figure 5.5(a). Once operations *GP-on-expanding*, *GP-on-contracting* and *GP-on-propagating* are implemented the satisfiability of feature models (operation *FM-is-satisfiable*) can be checked by simply forwarding the call to GPCS *GP-is-satisfiable* operation as shown in Algorithm 18. Notice that those three *GPCS* operations are represented in Figure 5.5(a) by communication channel (1). This channel allows a direct communication between the two internal solvers.

Algorithm 19 Count the number of solutions of the (possibly partially instantiated) feature model rooted by f

Inputs:
f: root of the feature model tree
Output:
returns the number of valid configurations in the feature model

Function **FM-count-sol**(f : $feature$)

```
1: $num\_solutions = 0$
2: for (each solution $S$ in the GPCS) do
3:     $num\_solutions = num\_solutions$ + FT-count-sol($f$)
4: end for
5: return $num\_solutions$
```

5.2.4 Counting solutions

As discussed earlier, the number of solutions in the feature model can be computed by applying operation *FT-count-sol* (Algorithm 9) to the feature tree repeatedly for each solution in *EC* that can be propagated successfully in the tree. This operation named *FM-count-sol* is implemented in Algorithm 19. In line 1, the counter variable *num_solutions* is initialized to zero. For each solution found by the GPCS in the extra constraints and propagated successfully in the feature tree, the available solutions in feature tree are counted and added to variable *num_solutions* (lines 2-4). Operation *FT-count-sol* is applied to the feature tree after the GPCS has pruned the tree properly by propagating assignments made to *EC* variables. Finally, the total number of solutions in *num_solutions* is returned (line 5).

While a constraint solver would take exponential time in the number n of features ($O(2^n)$) to count feature model solutions, the new procedure would take time

$O(n \cdot 2^k)$, where k is the number of variables in the extra constraints. That is, $O(2^k)$ for the number of decisions required to find all solutions and $O(n)$ for saving and restoring the state of the feature model prior to and after each decision and for applying *FT-count-sol* for each solution found. Considering that k is usually a fraction of n the new algorithm can be orders of magnitude faster than a constraint solver equivalent operation. For instance, for a feature model with 100 features ($n = 100$) and 20% ECR ($k = 20$), $2^n \approx 1.27{\times}10^{30}$ and $n \cdot 2^k \approx 1.05{\times}10^8$.

Algorithm 20 Returns an iterator object that can be used to enumerate the solutions of the feature model rooted by feature f

Inputs:
f: root of the feature model tree for which solutions will be enumerated
Output:
f: Returns an iterator object that can be used to enumerate the solutions of the feature model rooted by feature f

```
Function FM-create-sol-iterator($f$ : $feature$)
: FM-sol-iterator
1: ec-iterator = GP-create-iterator(FM-get-EC-formula())
2: return FM-create-iterator-object($f$, ec-iterator)
```

5.2.5 Enumerating solutions

The same rationale used to count solutions can be applied to enumerate solutions in the feature model. That is, each solution found in *EC* and propagated successfully in *FT* prunes the feature tree to a satisfiable state and yet several variables remain uninstantiated in the tree. This means that the tree encompasses several solutions. Iterators can be used to traverse the tree and enumerate each of these solutions.

Operation *FM-create-sol-iterator* creates an iterator object for the feature model rooted by feature f (see Algorithm 20). The iterator object for feature models consists of a fixed iterator for the extra constraints named *ec-iterator* and a varying iterator for feature trees named *ft-iterator*. That is, for each enumeration provided by the *ec-iterator* a new *ft-iterator* object is created to enumerate the solutions in the partially-instantiated feature tree. Line 1 in Algorithm 20 creates the *ec-iterator* using operation *GP-create-iterator* provided by the constraint solver. Notice that the function takes as argument the extra constraint formula retrieved

Algorithm 21 Returns *true* if the feature model solution iterator object still has a solution to enumerate or *false* otherwise

Inputs:
 fm-iterator: the feature model solution iterator object
Output:
 returns *true* if the feature model solution iterator object still has a solution to enumerate or *false* otherwise

```
Function FM-has-next-sol(fm-iterator:FM-sol-iterator)
: Boolean
 1: ft-iterator = FM-get-FT-iterator(fm-iterator)
 2: ec-iterator = FM-get-EC-iterator(fm-iterator)
 3: if (ft-iterator = NIL) OR (NOT FT-has-next-sol(ft-iterator)) then
 4:    if (GP-has-next-sol(ec-iterator)) then
 5:       GP-next-sol(ec-iterator)
 6:       f = FM-get-iterator-feature(fm-iterator)
 7:       ft-iterator = FT-create-iterator-object(f)
 8:       FM-set-FT-iterator(ft-iterator, fm-iterator)
 9:       return true
10:    else
11:       return false
12:    end if
13: end if
14: return true
```

by function *FM-get-EC-formula*. Next, an iterator object for the feature model named *fm-iterator* is created using function *FM-create-iterator-object*. The root of the feature model and the extra constraint iterator object are passed as parameters to the *fm-iterator*. Notice that the *ft-iterator* object is not created until the *ec-iterator* enumerates its first solution.

Algorithm 21 implements the *FM-has-next-sol* operation. This operation returns *true* if there are still solutions to be enumerated in the feature model for iterator *fm-iterator* or *false* otherwise. Lines 1 and 2 retrieve the two internal iterator objects *ft-iterator* and *ec-iterator* using auxiliary functions *FM-get-FT-iterator* and *FM-get-EC-iterator*, respectively. If the *ft-iterator* object is *NIL* or its enumeration has been completed (line 3) a new iterator needs to be created for the feature tree. In order to create the *ft-iterator* it is necessary to prune the feature tree by moving to the next solution in the extra constraints using operation *GP-next-sol(ec-iterator).* If there are no more solutions in the extra constraint iterator the function returns *false* (line 4 and 11), otherwise values are assigned and propagated in the feature tree to reflect the assignments that form the solution in the extra constraints (line 4-5). Once the feature tree is pruned the *ft-iterator* object can be created using function *FT-create-iterator-object* (lines 6-7). The iterator object is then associated with the feature model iterator (line 8) so this object can be retrieved later. Finally, the function returns *true* to indicate that a solution can be enumerated. Notice that the function returns *true* immediately whenever the *ft-iterator* object is not NIL and still has solutions to enumerate (lines 3 and 14).

Algorithm 22 Returns a list of features representing the next solution of the feature model solution iterator object

Inputs:
fm-iterator: the feature model solution iterator object
Output:
returns a list of features representing the next solution of the feature model solution iterator object

```
Function FM-next-sol(fm-iterator:FM-sol-iterator)
: feature{}
1: sol = NIL
2: if (FM-has-next-sol(fm-iterator)) then
3:     sol = FT-next-sol(get-FT-iterator(fm-iterator))
4: end if
5: return sol
```

Operation *FM-next-sol* shown in Algorithm 22 returns the actual solutions in the feature model for a given iterator object *fm-iterator*. Variable *sol* stores the features that are part of the solutions and is initially set to NIL. If a there is still a solution to be enumerated in the feature model (line 2) this means that object *ft-iterator* is not NIL and can be used to retrieve the solution. Operation *FT-next-sol* finds this solution in the feature tree and the result is stored in variable *sol* (line 3). Notice that auxiliary function *get-FT-iterator* has been used to retrieve the *ft-iterator* object associated with the feature model iterator object. Finally, the solution found is returned (line 5).

Functions *FM-create-sol-iterator*, *FM-has-next-sol*, and *FM-next-sol* should be used together to iteratively enumerate the solutions of a feature model. The following code fragment illustrates how these functions can be used together. First, a solution iterator *fm-iterator* is created to enumerate the solutions of the feature model rooted by feature f. Next, a loop is introduced to iterate over each solution in the subtree while one can be found. Inside the loop, the solution *sol* is retrieved from the *fm-iterator* object and printed out. The solution sets contain only features assigned *true*, i.e., all non-listed features are assigned *false*.

```
...
fm-iterator = FM-create-sol-iterator(f)
while ( FM-has-next-sol(fm-iterator) ) do
  sol = FM-next-sol(fm-iterator)
  print-out(sol)
end while
...
```

5.3 Hardness of Feature Model SAT Instances

In this section, we show that some of the properties discussed so far also allow us to understand better the hardness of SAT instances derived from feature tree and feature model formulas. In particular, the following property can be proved.

Property 5.3.1 *SAT instances derived from feature trees can always be solved in linear time by a SAT solver regardless of the variable/value order considered.*

Proof The unit propagation algorithm of the SAT solver will initially propagate the root feature as it represents a unary constraint. The propagation will succeed (property 5.1.2). Next, any unassigned variable is picked, assigned any truth value, and propagated. Propagation will succeed (property 5.1.4). The solver repeats this step several times until no more uninstantiated variables are left. The search algorithm never backtracks since all propagations succeed and the formula remains satisfiable after each propagation step. Providing that at most one decision is made for each variable the solver performs a linear number of steps to find a solution to the model. ∎

Since a feature model is the conjunction of the feature tree and the extra constraint formulas and considering that the feature tree can be solved in linear time by a SAT solver (property 5.3.1), we can conclude that it is the extra constraint formula that can potentially make a feature model SAT instance hard. The question is whether a formula that in practice usually consists of a mix of binary and ternary constraints and uses a fraction of the variables in the feature model can affect the hardness of the entire formula to an extent that the formula becomes intractable. We examine this issue in detail in Section 6.5 through empirical experiments.

5.4 Summary

In this chapter, we explored several properties of feature trees to propose an efficient reasoning system for feature trees called *FTRS*. The *FTRS* computes satisfiability checks and counts the number of solutions in the feature tree in constant and linear time in the size of the feature tree, respectively. The feature tree properties were further expanded into more general properties to support building a reasoning system for feature models called *FMRS*. The *FMRS* incorporates two reasoning systems, i.e., the *FTRS* and a standard constraint solver and is able to capitalize on the strengths of each system by defining an interesting interplay between them. That is, the constraint solver is used to find solutions in the extra constraints that propagate successfully in the feature tree, which prunes the tree after each solution is found. Once the tree is pruned, the algorithms in the *FTRS* can be applied efficiently on the pruned tree to extract some useful information. This approach has proven valuable to reduce the algorithmic complexity of certain operations. For instance, while a constraint solver requires exponential time on the number of features n in the feature model to compute the number of solutions in the model ($O(2^n)$), the *FMRS* takes time proportional to $O(n \cdot 2^k)$, where k is the number of distinct variables in the extra constraints. Ultimately, the goal of the *FMRS* is to provide insights and a ready infrastructure to encourage future research on enhancing the performance of reasoning algorithms for feature models by exploring specific properties of the feature modeling domain. Finally, the properties discussed were related to the hardness of feature model SAT instances. In particular, we showed that SAT instances derived from feature trees can be solved in linear time in the number of nodes in the tree by a typical SAT solver. This suggests that it is the extra constraint formula that can increase the hardness of feature model SAT instances. The question raised was whether this formula that in practice usually consists of a mix of binary and ternary constraints and uses a fraction of the variables in the feature model can affect the hardness of the entire feature model formula to an extent that this formula becomes intractable. This issue as well as the evaluation of the *FMRS* are addressed in Chapter 6 that reports on the results of empirical experiments.

Chapter 6

Evaluation

In this chapter we report on the results of several empirical experiments carried out to evaluate the ideas proposed in this thesis. In particular, our goal is to evaluate the quality of the orders produced by the two new BDD variable ordering heuristics discussed in Chapter 4 in terms of BDD size reduction and the increase on the size of feature models that can now be compiled to BDDs successfully. In addition, we compare the performance of pure SAT solutions against the hybrid algorithms discussed in Chapter 5 to verify the gains in terms of performance for certain operations. Finally, we examine the hardness of SAT instances derived from feature models to verify whether these instances can ever become intractable for realistic feature models.

Next, we provide details of the resources used in the experiments and report and discuss the observed results.

6.1 Hardware and Software

An AMD Turion system with a dual-core 1.6 GHz processor and 1 GB of RAM supported the experiments. Moreover, a testing tool was developed in Java (JRE 1.4.2) to run the various test case scenarios involving BDDs and SAT solvers. Publicly-available third-party Java libraries were embedded in the testing tool providing a convenient infrastructure for manipulating BDDs (JavaBDD [91]), constraint solvers (Choco [23]), and DPLL solvers (SAT4J [13]). The libraries sometimes required specific parameters to be configured. Such parameters and their configurations are discussed in the appropriate sections in this chapter.

Table 6.1: Feature models from literature

Feature model [reference]	**Features**	**ECR**	**Clauses**	**Arity**
1. e-Shop [51]	213	15%	21	2, 3
2. Model Transformation [27]	71	0%	0	-
3. Home Integration System [49]	67	12%	4	2
4. Documentation Generation [86]	44	29%	8	2
5. Thread domain [14, ch.6, p.130]	44	0%	0	-
6. Web Portal [64]	35	25%	6	2
7. Graph Manipulation [61]	30	23%	8	2, 3
8. Digital Video System [82]	26	23%	3	2
9. Key Word in Context [83]	25	16%	3	2
10. Insurance Product [84]	25	28%	4	2
11. Weather Station [16]	18	22%	2	2
12. Text Editor [29]	18	0%	0	-
13. Monitor Engine System [20]	17	11%	1	2
14. Graph Product Line [8]	16	81%	14	2
15. JPlug [79]	14	28%	2	2
16. James [11]	14	28%	2	2
17. Virtual Office of the Future [52]	14	0%	0	-
18. Search Engine (thesis page 11)	14	28%	2	2
19. Telecommunication System [37]	12	33%	2	2, 3
20. Cellphone [88]	11	36%	2	2

6.2 Benchmarks

6.2.1 Real Feature Models

We carefully examined a large body of research works in the field of software product lines and product configuration to build a collection of twenty feature models that served as basis for our experiments. Those models have been used in a variety of ways by their proposers and served mostly as a means to illustrate approaches and techniques applied to software product lines or alternatively as a convenient encoding for representing the common and variable aspects of a particular domain of interest. Table 6.1 shows the twenty feature models sorted by their size, i.e., the number of features in the model. Each row depicts a model. The first column in the table describes the domain of interest and provides a reference to the work where the model was introduced. The second and third columns ("features" and "ECR") depict the number of features and the ECR (see definition 4.1.1) for each model, respectively. Each model had its extra constraints converted to CNF. Column "clauses" shows the number of CNF clauses in the extra constraints of each model while the "arity" column shows the arity of those clauses. For instance, while the model in the third row (Home Integration System) contains 4 binary clauses, the model in the first row (e-Shop) contains 21 clauses with arities 2 and 3. Yet, other models did not contain any extra constraints (ECR=0%) such as the models in rows 2, 5, 12 and 17. In these cases we noticed that the models were used either as a taxonomy (model 2) or to describe small domains for which feature tree relations were sufficiently expressive.

Despite the quite good number of models available in the literature its was extremely challenging to find larger models, i.e., comprising tens of thousands of features. We are aware that such models exist and some have already been mentioned in the literature [7], however they are usually part of commercial projects that offer limited access to their resources. Therefore, we took advantage of our experience in assembling a collection of real feature models to develop a tool to generate feature model instances of arbitrary sizes. This is the topic of the next section.

6.2.2 Automatically-Generated Feature Models

We conducted a careful examination of several real feature models including those listed in Table 6.1 to better understand the similarities and differences among the models and to learn how to generate models that somehow mirror real models. Based on this experience we make some observations as discussed next.

Size and types of relations in the feature tree

Feature trees can vary significantly in size in practice. For instance, while the average size of feature trees in the models in Table 6.1 is 42, the smallest and largest models have sizes of 11 and 287, respectively. The size of the feature tree is usually related to aspects such as the complexity of the domain in terms of numbers of concepts and the depth of the variability analysis. Moreover, we noticed that most of the models examined make use of all types of features in their structure, i.e., mandatory, optional, and grouped features. For instance, only model #8 in Table 6.1 does not have grouped features. In addition, the frequency in which each type of feature appears in the models varies reasonably. For example, while 66% of the features in model #3 are mandatory only 26% of the features in model #1 are of this type. The same is with features of inclusive-OR and exclusive-OR groups. For instance, model #1 has 46% of grouped features while model #11 has no feature groups at all. This suggests that a generation tool has to be able to generate models of arbitrary sizes and allow different types of features to be represented. In addition, the frequency in which each feature type is expected to appear in the models should be parameterized.

Number of children per parent node and size of groups in the feature tree

The ratio of child nodes per parent can vary from model to model and even within the same model. Similarly, the number of features in a group is variable and hard to predict in practice. The smallest groups observed in the real models contained two features. In fact, groups are expected to have at least two features otherwise they could be easily represented as a mandatory feature relation. This suggests that a generation tool has to be able to produce models with a varying number of children per parent node, enforce a minimum of two nodes per feature group, and allow the parameterization of the maximum size of the groups.

ECR, and number and arity of clauses in the extra constraints

While some models in Table 6.1 have no extra constraints (e.g. model #2) others have a fairly large number of features as variables in the extra constraints, i.e., large ECRs (e.g. model #14). Also, most of the variables (94 or 74%) appear in only one clause and in 95% of the cases the clauses are binary. Despite, there are cases of models containing 3-ary clauses (e.g. models #1, #7, and #19) and having the same variables appearing in multiple clauses. Hence, generation tools should be able to produce models with different ECRs and eventually allow variables to repeat in some clauses. Optionally, the arity of the clauses can be parameterized within a range. Notice, however, that these parameters can interfere with each other. For instance, if the number of clauses specified is too large compared to the number of variables it is very likely that many variables will be repeated throughout many clauses especially if the average arity is low.

Extra constraints and horizontal relations in the feature tree

The total number of clauses in the extra constraints considering all feature models in Table 6.1 is 84. Only 6 of those clauses (7%) contain variables that also have an ancestral relation in the feature tree. That is, in 93% of the cases none of the variables in a clause is an ancestor of the others which implies that the variables belong to different subtrees in the model. This is a strong evidence that the relations in the extra constraints represent horizontal relations, i.e., connect different subtrees in the feature tree. In fact, this makes sense since the purpose of the extra constraints it to add new relations to the feature tree and this commonly translates to connecting different branches in the tree. Therefore, generation tools should enforce that most of the clauses generated for a model represent horizontal relations.

Vertical and horizontal distribution of extra constraint relations in the feature tree

We say a relation r in the extra constraint is modularized at level n if the lowest-common ancestor of its variables, written LCA(r), is a node at level n in the tree. This gives an idea of the impact of adding the relation to the feature model, i.e., the larger the subtree rooted by LCA(r) the higher the impact the relation can potentially cause to the feature tree. We are particularly interested in the vertical and horizontal distributions of this modularization, i.e., how the LCA of the relations in the extra constraints span over the various levels of the feature tree (vertical distribution) and within a given level (horizontal distribution). We examined the models in Table 6.1. The average vertical distribution of all models was 43%, i.e., in average 43% of the levels in the feature trees in each model contained the lowest-common ancestor of a relation in the extra constraints. For instance, in model #1 the feature tree has a depth of 9 and the 21 clauses in the extra constraints are modularized in the first four levels, i.e., 9, 7, 2 and 3 clauses in levels 0, 1, 2 and 3, respectively. In addition, within a given level the relations are also spread horizontally, i.e., multiple same-level subtrees modularize extra constraint relations. This suggests that a generation tool needs to be able to distribute extra constraints relations vertically and horizontally throughout the many feature tree levels.

Satisfiability

All feature models in Table 6.1 are satisfiable as they allow at least one valid configuration to be specified. However, as we argued in Section 3.1.1 during the construction of feature models it is usually necessary to perform debugging tasks, e.g., to check whether the models specified are satisfiable. Therefore, we understand that generation tools should be able to generate both satisfiable and unsatisfiable models in order to support the evaluation of techniques for debugging feature models.

Based on the observations previously made we developed a feature model generation tool called *GenBench* that is capable of producing collections of feature models. Our understanding is that we should evaluate reasoning techniques using collections of models rather than a single model and report the results as averages for the collection. This not only improves fairness but also allows us to keep track of specific cases in which a given technique performed (perhaps unexpectedly) better than the others.

The *GenBench* tool can be parameterized according to the following major parameters shown in Table 6.2

Table 6.2: Configuration parameters of the feature model generation tool *GenBench*

Collection name	Name or path to identify the collection.
Size of collection	Indicates the number of feature models that should be generated in the collection.
Size of feature models	A fixed size for the models in the collection. The final size of the feature models can vary slightly from the size indicated to accommodate other parameters.
Feature type odds	Odds for mandatory, optional, and grouped features. Ideally, these parameters should vary from 0 to 100 and add up to 100.
Minimum and maximum children per parent node	Indicates a range for creating parent node's children. For instance, the range [1,3] indicates that each parent node has a minimum of 1 and a maximum of 3 children.
Maximum size of feature groups	Indicates the maximum number of grouped features in feature groups. It is assumed that the minimum size of groups is 2.

ECR	Indicates the ECR, i.e., the percentage of the number of non-repeated features in the feature tree that will be added to relations in the extra constraint. For instance, if a feature tree has 500 features and the ECR specified is 10%, 50 features will be chosen to be part of at least one extra constraint relation.
Number of clauses and arity	Indicates the number of CNF clauses to be generated in the extra constraints and a range indicating the allowed arities. Currently the tool only supports binary constraints.
Vertical and horizontal distribution of extra constraint relations	Indicates the levels to be considered for the vertical and horizontal distributions of the extra constraint relations. Since the depth of the tree is not known upfront, the levels are specified in terms of percentage numbers. For instance, if the tree has a depth of 9 then percentages 0%, 50% and 100% correspond to levels 0, 4 and 8, respectively (vertical distribution). In addition, for each level two other percentages are specified to indicate, respectively, the percentage of features in the extra constraints that should be allocated to this level, and the percentage of the nodes at this level that should modularize extra constraint relations (horizontal distribution). For instance, consider a feature tree containing 500 nodes, ECR of 10% and depth 9. If parameters 30% and 50% are specified for the 4^{th} level of the tree that contains 10 nodes then 30% of the extra constraint variables, i.e., 30% of 50 = 15 variables should be allocated for level 4, and 50% of the 10 nodes at this level, i.e., 5 nodes should be chosen randomly to modularize extra constraint relations.

Satisfiability	Indicates whether the models in the collection should be satisfiable (*true*) or unsatisfiable (*false*). This is implemented by testing the satisfiability of the models right after their generation using a SAT solver. If the goal is to produce unsatisfiable models the ECR value specified should ideally be high (e.g. at least 30%) otherwise it is very likely that the models generated will be satisfiable.

In addition to the parameters in Table 6.2, the *GenBench* tool was designed to generate horizontal relations in the extra constraints, i.e., to enforce as much as possible that the variables that appear together in a clause do not have an ancestral relation in the feature tree. As we have shown, this is typically the case observed in real feature models.

Several collections have been generated to support our experiments. The collections usually consisted of 50 models of pre-determined size and the size of the models varied among different collections to support different types of experiments. The ECR was also variable but typically fixed for a given collection. That is, a collection can be typically characterized by the number of models in the collection, the fixed size of its models and the specified ECR, e.g., a collection A with 50 feature models containing 500 features each and ECR of approximately 10%. For different collections the ECR ranged from 10% to 30% to mirror what has been observed for real feature models. Yet, some other parameters have been fixed for collections such as the odds for mandatory, optional, and grouped features for inclusive-OR and exclusive-OR groups, respectively, 25%, 35%, 20% and 20%. The odds approximated those for real feature models but we have adjusted slightly the odds for mandatory features to facilitate computing the size of simplified models, i.e., models for which mandatory features have been removed. For instance, if a given model has 100 features and 25% of its mandatory features have been removed it is easy to compute the new size of the model, i.e., 75 features.

The number of children per parent node and the size of groups varied from 1 to 5 and 2 to 10, respectively, in most of the collections. Those ranges match reasonably well what has been observed for real models but in practice they are rather hard to predict. Most of the models generated were satisfiable but a few collections contained unsatisfiable models. The satisfiability of the models is indicated in the appropriate sections when the experiments and their results are presented next. Finally, the horizontal and vertical distributions of extra constraint relations spanned over a range of 2 to 4 levels in the tree depending on the size of the models and vertically among 50% to 100% of the nodes within the level. These parameters were eventually adjusted whenever the ECR of the generated models did not quite matched the specified ECR for the collection. More details of the configuration of the parameters used in the test cases are presented in the corresponding sections next.

We have been very encouraged by other research groups to continue working on the benchmarks. For instance, a research team in Brazil that uses the Alloy system to reason on feature models has downloaded our tool and some of the benchmarks to test the performance of Alloy in handling large scale feature models. Another research group in Germany is now implementing support for our models in their feature configuration tool called FeatureIDE [53].

In the following, we present the experiments performed to test the various reasoning techniques presented in this thesis and discuss the results obtained.

6.3 Evaluating BDD Minimization Heuristics

We performed a series of experiments to evaluate the BDD variable ordering heuristics discussed in Chapter 4 including our proposed heuristics *Pre-CL-Size* and *Pre-CL-MinSpan*. Only the most competitive heuristics have been considered thus eliminating the *level* heuristic which performed very poorly. The goal was to measure the effectiveness of the proposed heuristics in reducing the size of BDDs for real and automatically-generated feature models in comparison to naive (yet generally efficient) implementations such as natural pre-order and other renowned heuristics such as Fujita's, FORCE and sifting. In addition, we wanted to measure the scalability of the heuristics in terms of the maximum size of the models that could be handled and how much this has been improved by the new heuristics.

6.3.1 Real Feature Models

Quality of BDD Size Reduction

Goal: This experiment aimed at measuring the quality of the *Pre-CL* heuristics in reducing the size of BDDs for real feature models in contrast with existing heuristics.

Benchmark: The real feature models illustrated in Table 6.1 supported the experiment. For this particular experiment mandatory features were not removed from

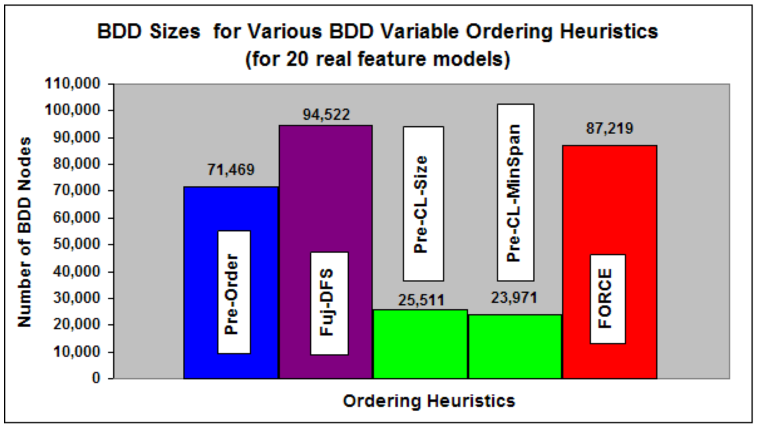


Figure 6.1: BDD sizes for various BDD ordering heuristics for real feature models.

the models. Recall that we refer to feature models from which mandatory features were safely removed as *simplified models* (see Section 4.2.2 on page 49).

Results & Analysis: Figure 6.1 shows the total size of the BDDs produced by each heuristic for the models listed in Table 6.1. Heuristics *Fuj-DFS* (Fujita's) and FORCE produced very large BDDs containing 94,522 and 87,219 nodes, respectively. Meanwhile, natural *pre-order* ranked 3^{rd} yielding a BDD of size 71,469. The *Pre-CL* heuristics produced the best variable orders, i.e., the orders that led to the smallest BDDs. Heuristics *Pre-CL-Size* and *Pre-CL-MinSpan* produced a total BDD size of 25,511 and 23,971 nodes, respectively, i.e., about 3 times smaller than those produced by natural *pre-order*.

In the specific case of models containing no extra constraints (models #2, #5, #12, and #17) the *Pre-CL* heuristics were never worse than natural *pre-order* and in some cases significantly better (e.g. 1.9 times smaller BDD for model #5). This shows that the pre-order traversals applied by those heuristics are usually much more effective than the natural pre-order traversal when it comes to reducing the size of BDDs.

The *Pre-CL* heuristics produced smaller BDDs in 12 (or 60%) of the cases. Yet, in all other cases the difference between the better heuristic and the *Pre-CL* heuristics was very low (about 13% smaller BDDs) and was usually observed for the smallest models (18 features or less).

We strongly believe that the poor performance of circuit heuristics such as *Fuj-DFS* is related to the fact that these heuristics are usually not able to capitalize on the hierarchical arrangement of features in the feature model. Yet, this arrangement certainly provides a good hint on how to order BDD variables for those models as we discussed when we proposed the *Pre-CL* heuristics. While features are nodes in the feature model, they are inputs (leaves) in the circuit and circuit nodes are represented by gates (Boolean functions). Therefore, the strategy for developing circuit heuristics can be very different from that of feature model heuristics.

Another important issue that should be considered to evaluate the performance of circuit heuristics, or generally any other domain-specific heuristic, is related to the translation of feature models to a domain-specific encoding such as a circuit. That is, there are many possibilities for translating the same feature model to a circuit graph (see Figure 2.7 on page 26 for an example) yet this can significantly impact the quality of the orders produced by circuit heuristics. Learning how to produce the "most valuable" circuit structures can be time-consuming and inefficient in practice. Even in the best cases, this approach would always require building an extra structure that can be orders of magnitude larger than the corresponding feature model before applying the heuristics. Therefore, we do not see real benefits in using circuit heuristics directly in the feature modeling domain but rather in learning the rationale behind these heuristics and examining whether the ideas can be somehow applied advantageously in that domain.

Summary:

- *Pre-CL* heuristics produced BDDs about 3 times smaller, on average, than other heuristics.
- *Pre-CL* heuristics were effective even for models without extra constraints.

Quality of BDD Size Reduction on Simplified Models

Goal: The goal of this experiment was to learn how much the heuristics can capitalize on the simplification of the models (elimination of mandatory features) in order to improve BDD size reduction. As we argued in Section 4.2.2 mandatory features play no role in variability analysis and should be removed for this purpose. Notice that this operation preserves the core semantics of the models and hence the BDDs obtained for the simplified models can also be used to reason on the original models.

Benchmark: The real feature models illustrated in Table 6.1 **after simplification** supported the experiment.

Results & Analysis: We observed that all the heuristics were able to take advantage of the simplification of the models to produce smaller BDDs (see Figure 6.2).

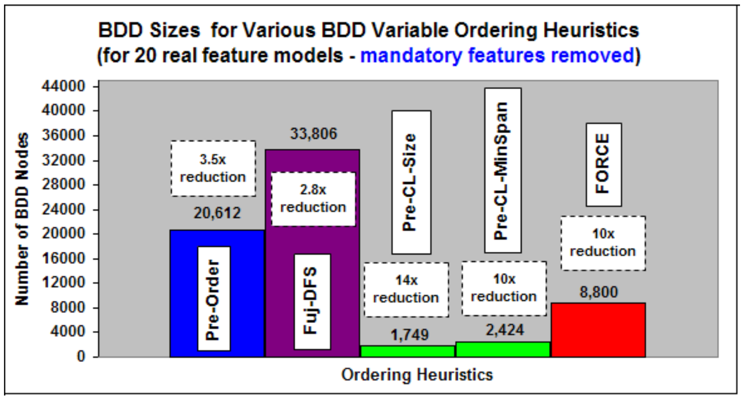


Figure 6.2: BDD sizes for various BDD ordering heuristics for simplified models.

Indeed, this was quite expected since the size of simplified models were about 30% smaller than the original models (odds for mandatory features in the models). The performance of the heuristics were similar to what was observed in the previous experiment with the exception of *FORCE* that performed better than *pre-order*.

However, it was quite intriguing that some heuristics achieved a much higher reduction rate than others. For instance, while BDDs produced by heuristics *pre-order* and *Fuj-DFS* were 3.5 and 2.8 times smaller, the *Pre-CL* heuristics and *FORCE* observed reduction rates of 10 times or higher. As a consequence, the difference in terms of quality of BDD size reduction among the heuristics increased significantly. That is, while for the original models the *Pre-CL* heuristics produced BDDs from 3 to 4 times smaller than the other heuristics this difference increased to a range between 3.6 to 20 times for simplified models. Considering that simplified models are the actual target of the heuristics it becomes critical to understand how the heuristics are impacted by the structural changes made to the original models after simplification.

Since *FORCE* is a non-deterministic heuristic it is hard to explain whether the improvements observed were real or incidental. On the other hand, we are interested in understanding the impact of model simplification on traversal-based heuristics such as *pre-order* and the *Pre-CL* heuristics.

Figure 6.3a shows how simplification impacts the structure of a feature model. In the figure, (a) shows a feature model containing 10 features, 2 of which are mandatory (B and I), and two constraints C_1 and C_2. The simplification of the model is depicted in (b). Notice that the simplification process updates both the feature tree structure and the extra constraint relations in the original model. For instance, since feature I has been removed the reference to this feature in relation

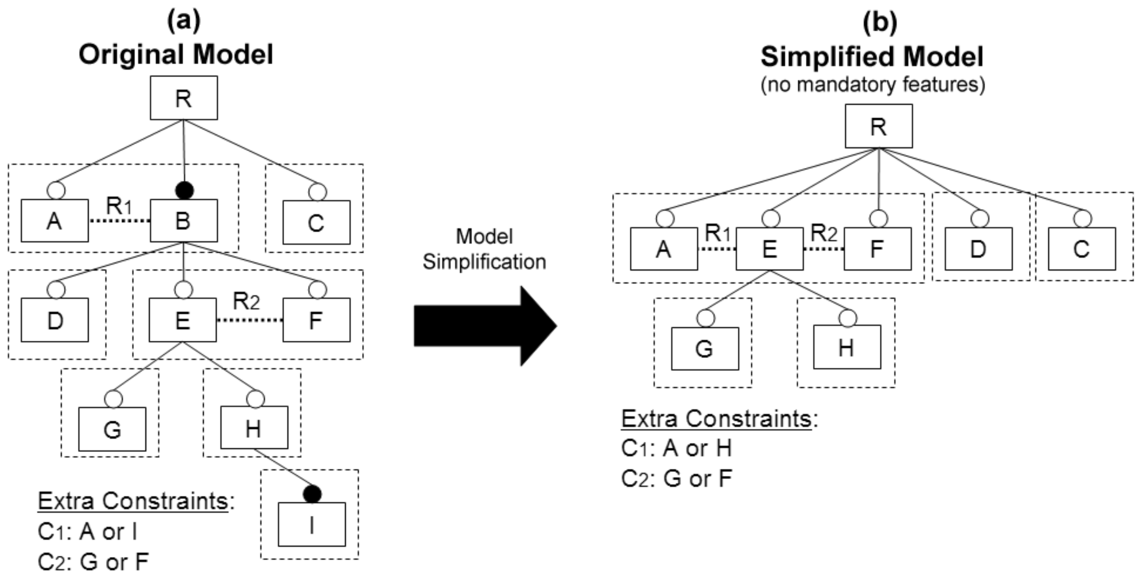


Figure 6.3: How feature model clusters are affected by model simplification

C_1 was updated to refer to feature H (since I can now be inferred from H). In addition, the simplification causes a reduction on the depth of the feature tree from 4 to 2. This results in a "flattening" of the feature tree and increases the number of children for certain nodes. That is, every parent of a mandatory node inherits the children of this node after its removal from the model. For instance, consider the root feature R in the model depicted in Figure 6.3a containing three children A, B, and C. When the model is simplified R's mandatory child feature B is removed from the model causing B's children to move one level up in the tree. As a result, R has now five children: A, D, E, F, and C as depicted in Figure 6.3b.

In addition, the model simplification can increase significantly the complexity of the clusters in the model. The clusters of each parent node are illustrated in Figure 6.3 as dashed rectangles involving features. For instance, in the model in (a) feature R has two clusters, one containing features A and B, and another containing feature C. As well, feature E has two single clusters, one for each child node. Other clusters are illustrated in the figure. Cluster relations indicate subtree dependency. For instance, relation R_1 shown in one of R's clusters indicate that the subtrees rooted by A and B have a dependency (caused by relation C_1). When a mandatory feature is removed from the model its clusters are inherited by its parent node. In addition, whenever original and inherited clusters share nodes they are combined into a single cluster. For instance, the removal of feature B caused feature R to inherit its two clusters. However, because feature E replaces feature B in relation

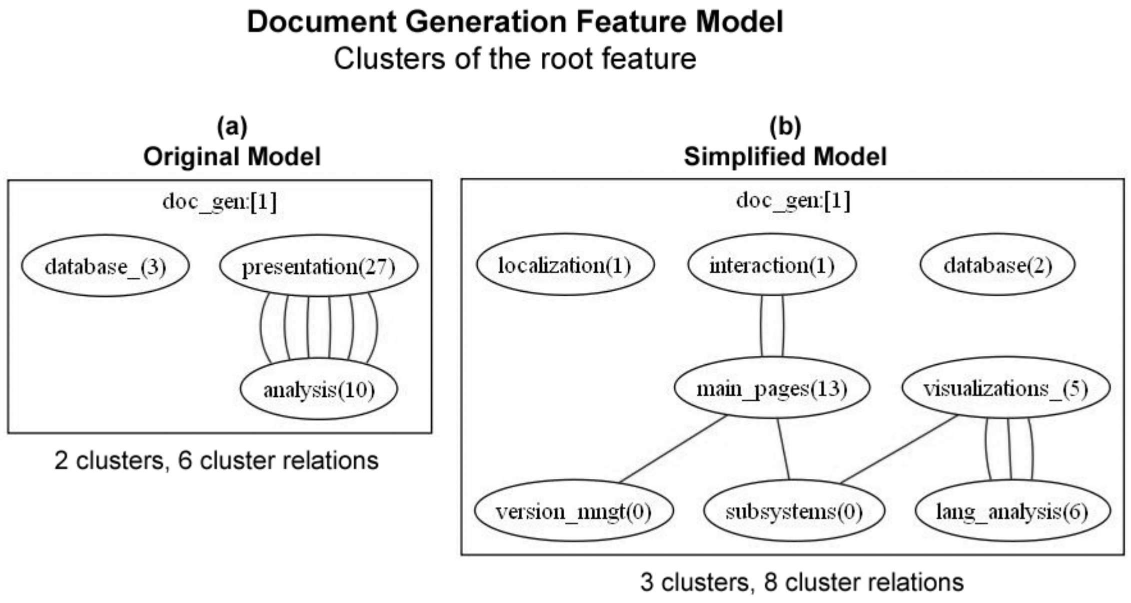


Figure 6.4: Level-one clusters of the "Document Generation" feature model (a) before and (b) after model simplification

R_1 and E is also related to feature F through relation R_2, A, E, and F are combined into a single cluster and attached to node R as shown in Figure 6.3b. As a result, R now has a more complex cluster containing 3 features and 2 relations. In practice, the collapsing of clusters caused by the removal of mandatory features increases significantly the complexity of the clusters in the model both in terms of number of nodes as well as in the number of relations.

Figure 6.4 shows the effects of simplification in the clusters at level one of the "Document Generation" feature model in Table 6.1 (model #4). In (a) two clusters for the root node *doc_gen* are shown, one containing feature *database* and the other containing features *presentation* and *analysis*. The latter comprises 6 relations indicating subtree dependencies caused by extra constraint relations. Parenthesized numbers following feature names indicate the number of nodes in the subtree rooted by the feature (e.g. 27 nodes in the subtree rooted by feature *presentation*). Figure 6.4b shows the effects of simplification to '*doc_gen* clusters. Mandatory features *presentation* and *analysis* are removed and replaced some of their descendant features in the cluster. In addition, several clusters residing at lower levels in the tree are now moved up and collapsed to form a large cluster comprising a complex network of relations. As a result, feature *doc_gen* now has 2 single clusters and a cluster with 6 features and 8 relations.

The raise in the number of children for many parent nodes in simplified models affect negatively the *pre-order* heuristic since the larger the number of child nodes the higher the chances of natural *pre-order* to be inefficient. That is, it becomes more likely that the heuristic will crosscut several subtree dependencies during the traversal of the feature tree to generate an order. On the other hand, what makes the *Pre-CL* heuristics effective for simplified models is the ability of the heuristics in modularizing subtree dependencies into clusters and enforcing feature arrangements that aim variable distance minimization.

However, the larger the number of relations within clusters the more likely the *Pre-CL-MinSpan* heuristic will outperform its counterpart. This happens because in clusters containing several relations it is common to have some of the nodes connected to many others and the *Pre-CL-MinSpan* is able to place those highly-connected nodes in between their dependent nodes thus minimizing their distances in the variable order. For instance, consider the largest cluster in Figure 6.4b containing 6 nodes and 8 relations. Nodes *main_pages* and *visualizations* are connected to many others. Despite, heuristic *Pre-CL-Size* will ignore this fact and will order the cluster's nodes according to the size of their subtrees, i.e., *version_mngt*, *subsystems*, *interaction*, *visualizations*, *lang_analysis*, and *main_pages* (0, 0, 1, 5, 6 and 13 nodes, respectively). Instead, *Pre-CL-MinSpan* will place nodes *main_pages* and *visualizations* in between their dependent nodes to produce the order: *version_mngt*, *interaction*, *main_pages*, *subsystems*, *visualizations*, and *lang_analysis*. Since we observed that in many cases the clusters of simplified models contain a high number of relations, we expect heuristic *Pre-CL-MinSpan* to be more effective than heuristic *Pre-CL-Size* for those models.

Summary:

- All heuristics capitalized on model simplification to improve BDD reduction.
- *Pre-CL* heuristics produced BDDs up to 20 times smaller, on average, than other heuristics.
- Simplified models significantly changed the structure of original models. However, *Pre-CL* heuristics did not performed poorer when models were simplified but rather improved even more over the other heuristics.

Table 6.3: Average running times and BDD sizes for 50 feature models with 500 features and 20% ECR

Heuristic	Heur. Time [%]	BDD Time [%]	Running Time [ms]	BDD Size	Best Results
Pre-CL-MinSpan	2%	98%	273.70	2,683	36
Pre-CL-Size	2%	98%	218.69	3,496	14
Pre-Order	0.4%	99.6%	241.23	27,600	0
FORCE	69%	31%	14,741.71	137,515	0
Fuj-DFS	2%	98%	427.86	41,176	0

6.3.2 Automatically-Generated Feature Models

Quality of BDD Size Reduction and Running Times

Goal: This experiment aimed at comparing the quality of the heuristics in reducing the size of BDDs for models generated automatically. As well, the times required by each heuristic to generate orders and build the BDDs were compared and analyzed.

Benchmark: The experiment used a collection consisting of 50 satisfiable feature models generated automatically each containing 500 features and average ECR of 20%. The odds for mandatory, optional, and grouped features of inclusive-OR and exclusive-OR groups were set to 25%, 35%, 20% and 20%, respectively. The models were simplified prior to running the experiments.

Results & Analysis:

Table 6.3 shows average space and time values for five different heuristics. Columns *Heur. Time* and *BDD Time* indicate the percentage of the running time for producing the variable order and building the BDD, respectively. The running time in milliseconds and the size of BDDs are shown in columns *Running Time* and *BDD Size*. The *Best Results* column indicates the number of test cases in which the heuristic had the best performance among all others.

BDD sizes for *Pre-CL-MinSpan* and *Pre-CL-Size* were significantly smaller than for any other heuristic. In particular, reduction rates of 10 and 8 times, respectively, were observed when compared to *pre-order* that ranked third. Interestingly, this accurately resembles the rates observed for those heuristics in the context of real models. *Pre-CL-MinSpan* led to smaller BDDs in 72% of the cases (36 models) while *Pre-CL-Size* performed best in 28% (14 models) (column *Best Results*). In terms of BDD reduction, Fuj-DFS was worse but still competitive with *pre-order*. *FORCE* produced poor results mainly due to its random starts. BDDs for *FORCE* were 52 times larger, on average, than those for *Pre-CL-MinSpan*.

Pre-order had the best heuristic running time due to its very simple algorithm that performs linearly on the size of the feature tree. However, *Pre-CL* heuristics were not far behind, just a few milliseconds worse but achieved total running times comparable to *pre-order*. *Fuj-DFS* also required low running times for producing variable orders. *FORCE* did not perform well. For all test cases, *FORCE* required more time to produce orders than to build the BDD (69% and 31%, respectively). As well, the total running time was very high when compared to the other heuristics which suggests that the heuristic can become prohibitively slow for larger models. In a specific run, the algorithm took 96 steps to reduce an initial span of 147,153 to a minimum span of 17,361. Each step took about 0.27 milliseconds to run which led to a total running time of 27 seconds.

In the experiment, we also tried to use *FORCE* to improve the orders produced by *Pre-CL* heuristics. *FORCE* was given initial orders produced by *Pre-CL-MinSpan* (the best heuristic) and strived for improvements based on span minimization. Despite the lower spans obtained *FORCE* was unable to improve the quality of *Pre-CL-MinSpan* orders for 80% of the cases (40 models). This suggests that this heuristic already produces high quality orders.

Summary:

- Heuristics' performance resembled accurately what was observed for real models, i.e., much superior orders were produced by *Pre-CL* heuristics.
- Time to run *FORCE* was higher than time to build the BDD.
- *FORCE* could not improve *Pre-CL-MinSpan* orders in 80% of the cases which suggests that the latter heuristic already produces high quality orders.

Scalability

Goal: This experiment measured the scalability of three of the best heuristics (*Pre-CL-Size*, *Pre-CL-MinSpan*, and *pre-order*). The goal was to find the upper bounds of each heuristic in terms of the maximum size of models that could be handled without producing "memory overflow" errors. The testing tool was set to run on 650 Mb of dedicated memory.

Benchmark: Supported the experiments 5 collections consisting of 10 satisfiable feature models each. The 10 models within a collection had the same size and the

Table 6.4: Scalability measures for the best heuristics

Heuristic	Feature Model Size (20% ECR)				
	1000	1500	2000	2500	3000
Pre-CL-MinSpan					
Successes [%]	100	100	100	70	0
Memory Overflows [%]	0	0	0	30	100
BDD sizes (# of nodes)	24,426	396,736	506,899	1,910,452	-
Running Times [seconds]	0.3	1.9	3.7	9.9	-
Pre-CL-Size					
Successes [%]	100	90	90	10	0
Memory Overflows [%]	0	10	10	90	100
BDD sizes (# of nodes)	36,512	1,313,058	1,469,996	4,033,693	-
Running Times [seconds]	0.3	6.6	10.6	19.4	-
Pre-order					
Successes [%]	100	70	30	10	0
Memory Overflows [%]	0	30	70	90	100
BDD sizes (# of nodes)	390,218	1,259,748	5,616,119	4,953,427	-
Running Times [seconds]	1.1	6.1	22.4	33.8	-

ECR of all models was set to 20%. However, the size of the models varied from one collection to another, i.e., 1000, 1500, 2000, 2500, and 3000 features. In all models the odds for mandatory, optional, and grouped features of inclusive-OR and exclusive-OR groups were set to 25%, 35%, 20% and 20%, respectively. The models were simplified prior to running the experiments.

Results & Analysis: Table 6.4 shows the results of scalability tests for the heuristics *Pre-CL-Min-Span*, *Pre-CL-Size* and *pre-order*. Table columns indicate feature models with different sizes. Rows indicate the completion or failure due to memory overflow to build the BDD as well as the average BDD size and running time. The running time represents the combined times of running the heuristic and building the BDD.

Heuristic *pre-order* did not fail to produce BDDs for any of the 1000-feature models. However, for larger models the percentage of failing cases increased significantly, i.e., memory overflows were observed for 30%, 70% and 90% of the cases for models containing 1500, 2000, and 2500 features. Meanwhile, the *Pre-CL* heuristics were able to produce BDDs for at least 90% of the cases for models with 2000 features or less. This represents an improvement of about 2 times compared to non-failing cases of *pre-order*. *pre-order* and *Pre-CL-Size* struggled to handle models containing 2500 features and were only able to build the BDD in a single case (1 model or 10%). Yet, *Pre-CL-MinSpan* handled 70% of those models satisfactorily. An interesting fact observed was that in none of the cases in which *Pre-CL-MinSpan* failed to build the BDD the other heuristics succeeded. Similarly, *pre-order* never succeeded for thoses cases in which *Pre-CL-Size* failed. None of the heuristics were able to produce BDDs for models containing 3000 features.

The size of the BDDs produced by *Pre-CL-MinSpan* were consistently smaller than those observed for the other heuristics and never exceeded 2 million nodes in average. Moreover, *Pre-CL-Size* produced smaller BDDs than *pre-order* in almost all cases. The only exception seemed to be for models with 1500 for which BDDs for *pre-order* had an average size of 1,259,748 nodes against 1,313,058 nodes of *Pre-CL-Size*. However, note that the average numbers did not consider the failing cases including the 3 models (30% of failures) not handled by *pre-order* that could have increased considerably the averages for this heuristic.

Heuristics *Pre-CL-MinSpan*, *Pre-CL-Size*, and *pre-order* ranked 1^{st}, 2^{nd}, and 3^{rd}, respectively, considering the total time to produce an order and build the BDD. This matched the size of the BDDs produced by those heuristic, i.e., smaller BDD structures were usually built faster. In general, the average running times observed were very low, i.e., well under 1 minute.

The largest BDD structure generated in the experiment contained 8,402,608 nodes and was produced by *pre-order* for a model with 2000 features. It took 58.5 seconds for the BDD library to count the number of satisfying assignments in the BDD structure which corresponds to the number of valid configurations in the feature model. For BDDs containing 2 million nodes or less the number of satisfying assignments could be counted in less than 3 seconds in most cases. This shows that BDD operations can be performed quite efficiently even in very large structures thus being building the BDD structure within a limited memory space the major issue.

Summary:

- The largest models handled by *pre-order* without any failures contained 1000 features.
- The largest models handled by *Pre-CL-MinSpan* without any failures contained 2000 features, an order of magnitude improvement over *pre-order*.
- *Pre-CL-MinSpan* handled 70% of the models containing 2500 features against only 10% of successes observed for the other heuristics.

- None of the heuristics handled models with 3000 features.
- The average running times for all three heuristics remained under 1 minute for models with up to 2500 features.
- The largest BDD produced in the experiment contained 8,402,608 nodes. It took 58.5 seconds for this BDD to count the number of satisfying assignments. This operation is usually performed in 3 seconds or less for BDDs with up to 2 million nodes.

Sifting

Goal: Several "memory overflow" errors were observed for the *Pre-CL* heuristics during the building of BDDs for very large feature models, e.g., models containing 2500 features or more. This experiment measured the practical use of *sifting* in preventing memory overflows during the building of such large BDDs. Once again, the *Pre-CL* heuristics were used to produce orders but this time the "sifting" option was turned on in the JavaBDD library. During BDD construction, whenever the size of the BDD structure reached 5 million nodes the "sifting" algorithm was automatically invoked by the library to reduce the size of the structure.

Benchmark: A collection containing 10 feature models with 2500 features and ECR of 20% (the same one used in the scalability experiment). The models were simplified prior to running the experiments.

Results & Analysis: The results shown were not encouraging. Even though memory overflows were prevented successfully none of the 10 models could be compiled after an hour of processing. Recall that *Pre-CL-MinSpan* was still able to generate BDDs for 70% of the models with average generation time of about 9.9 seconds as shown in Table 6.4. We observed many calls to the sifting algorithm during the construction of the BDD each taking several minutes to complete. Therefore, we do not see any real benefits of using *sifting* to build BDDs for feature models.

Summary:

- BDD construction using *sifting* never terminated after 1 hour of processing. We do not see any real benefits of using *sifting* to build BDDs for feature models.

6.4 Evaluating Constraint Solvers and the FMRS System

6.4.1 Satisfiability and solution counting with constraint solvers

Goal: The purpose of this experiment was to measure the efficiency of two popular SAT (SAT4J 1.7 - DPLL solver) and CSP (Choco 1.2) solver implementations in performing satisfiability checks and counting the number of solutions in feature models of varying sizes. The best solver instances available in each library (according to their authors) were selected and given the same variable order (pre-order traversal of the feature tree) and value ordering (*false* followed by *true*). The SAT solver library provides several state-of-the-art optimization techniques such as restarting strategies, watched literals and clause learning. Meanwhile, the CSP solution does not currently support such advanced features and was simply configured to apply forward checking to propagate assignments. In fact, the goal of the experiment was not to make a direct comparison of the solver's performance, which would obviously favor the SAT solution, but rather to understand how well general solver solutions could handle problems in a specific domain of interest, i.e., the feature modeling domain.

Benchmark: Supported the experiments 12 collections consisting of 10 satisfiable feature models each. The 10 models within each collection had the same size and the ECR of all models was set to 20%. However, the size of the models varied from one collection to another. For satisfiability tests 7 collections were used with model sizes of 500, 1000, 2000, 3000, 4000, 5000 and 10000 features. For solution counting tests 5 collections were used including the real feature models in Table 6.1 and 4 other automatically-generated collections with model sizes of 20, 30, 40 and 50 features. In all models the odds for mandatory, optional, and grouped features of inclusive-OR and exclusive-OR groups were set to 25%, 35%, 20% and 20%, respectively. The models were not simplified prior to running the experiments.

Table 6.5: Performance results for **satisfiability tests** on various collections of feature models. Timeouts indicate lack of response within 30 seconds. Running times are average results of the successful cases.

Solver	Feature Model Size (20% ECR)						
	500	1,000	2,000	3,000	4,000	5,000	10,000
SAT4J 1.7 (SAT solver)							
Timeouts [%]	0	0	0	0	0	0	0
Running Times [ms]	4	8	11	15	10	21	55
Choco 1.2 (CSP solver)							
Timeouts [%]	0	20	10	10	50	80	90
Running Times [ms]	14	20	61	239	78	53	197

Table 6.6: Performance results of SAT/CSP solvers in **counting solutions** on various collections of feature models. Timeouts indicate lack of response within 30 seconds. Running times are average results of successful cases.

Solver	Feature Model Size (20% ECR)				
	Real	20	30	40	50
SAT4J 1.7 (SAT solver)					
Timeouts [%]	20	0	0	90	100
Running Times [ms]	2,421	960	2,958	8,295	-
Choco 1.2 (CSP solver)					
Timeouts [%]	20	0	0	30	80
Running Times [ms]	310	79	2,869	7,833	9,228

Results & Analysis:

Satisfiability

Table 6.5 shows the performance results for the SAT and CSP solvers in performing satisfiability checks on various collections of feature models. The "Timeouts" rows indicate the cases in which no response was provided by the solver within 30 seconds of processing. The "Running times" rows are average results of the successful cases.

The SAT solver performed extremely well (first row in Table 6.5) with no timeouts even for models with 10,000 features. As well, the running times were very low never higher than 55 milliseconds. During the tests the SAT solver was able to "learn" from conflicts and adjust its searching strategy accordingly and for many times avoided unproductive steps. As a result, we noticed a reduced number of decisions made by the SAT solver during the search. A decision corresponds to a node in the search tree in which a truth value is assigned to a particular variable by the solver. For instance, an average number of decisions of 2,681 was observed for the collection of models containing 5,000 features, i.e., lower than the number of variables in the problem. This result is consistent with what was discussed in Chapter 5, i.e., during the search several variables corresponding to subtrees containing no extra constraint relations in the feature tree could be instantiated in one shot by assigning *false* to the root of the subtrees and propagating. In addition, the vast majority of the relations in the problems were represent by binary constraints which are usually an indication of easy SAT instances.

Meanwhile, the CSP solver (second row in Table 6.5) handled well models with up to 3,000 features with an average of only 10% of timeouts. For larger models the solver alternated cases of quick responses and no responses at all indicating that for times the solver wasted too much time examining unproductive branches of the search tree. For instance, only one model in the collection of models with 10,000 features could be handled within the time limit set for the experiment. Despite, there is certainly room for improvements such as considering different variable/value ordering heuristics or reducing the number of variables analyzed as will be discussed in the next experiment (Section 6.4.2).

Counting Solutions

Table 6.6 shows the performance results of the SAT and CSP solvers for counting the solutions on various collections of feature models within a 30 second time limit. Table rows show the average timeouts and running times for each collection.

The SAT solver was able to count the solutions of 16 (or 80%) of the real feature models shown in Table 6.1 in an average time of 2.4 seconds. The 4 models comprising the largest number of solutions could not be handled within the time limit, i.e., models #1, #3, #4 and #18 containing approximately from 750 thousands to 2 million solutions. No timeouts were produced for automatically-generated models with up to 30 features (20% ECR) for which the average time remained under 3 seconds. However, for models with 40 and 50 features the timeouts increased considerably to 90% and 100%, respectively, considering the substantial increase in the number of solutions in those models ($\approx 9.2\times 10^5$ and $\approx 15\times 10^6$, respectively).

The performance of the CSP solver was slightly better than the SAT solver. For the real models as well as models with 20 and 30 features (columns 2, 3, and 4 in Table 6.6) the solvers produced the exact same number of timeouts with a slight advantage for the CSP solver in the average running time. In addition, the CSP solver performed better on models with 40 and 50 features completing the solution counting for 30% and 80% of the models.

Considering the largest (smallest) number of solutions that could be counted (not counted) by each solver we attempted to determine approximate upper bounds. The SAT solver was able to count up to 80,658 solutions but failed for a model containing 85,654 solutions thus suggesting the upper bound to be near 83,000 solutions. The CSP solver successfully counted 654,720 solutions but failed to count 1.2 million of solutions thus suggesting an upper bound within the range [6.5×10^5, 1.2×10^6]. Another factor that should be considered is the number of steps taken by the solver to find the solutions. That is, the less the number of steps taken the higher the number of solutions that can be counted.

Ultimately, the results confirm what is known for general cases, i.e., that SAT/CSP solvers are not ideal for counting solutions of constraint problems. More broadly, constraint solvers are highly specialized in the satisfiability problem and hence are usually not expected to perform well when they are required to deal with a very large portion of the combinatorial space. Despite, the experiment was important in the sense that it allowed us to determine upper bounds for a given class of feature models and ultimately encouraged the development of alternative hybrid solutions for counting problem solutions as the hybrid reasoning system FMRS introduced in Chapter 5 and evaluated in the next experiment (Section 6.4.2).

Summary:

- The SAT solver was able to check the satisfiability of models with up to 10,000 features in 55ms or less. In several situations the SAT solver "learned" with conflicts and thus was able to reduce significantly the number of decisions in the search tree. The CSP solver performed well for models with up to 3,000 features (only 10% of timeouts) but struggled with larger models (80% and 90% timeouts for models with 5,000 and 10,000 features) most likely because it wasted a significant amount of time examining useless branches of the search tree.

- Both solvers performed poorly to count feature model solutions compared to their performance on satisfiability tests. The upper bound limits approximate 80,000 solutions for the SAT solver and between 6.5×10^5 and 1.2×10^6 solutions for the CSP solver. This usually corresponded to models with 50 features or less (20% ECR).

6.4.2 Satisfiability and solution counting with the FMRS system

In Chapter 5 we introduced a feature model reasoning system called FMRS. The FRMS combines a general-purpose constraint solver (GPCS) and a reasoning system tailored to feature trees (FTRS) to form a hybrid reasoning system for feature models. The FTRS component was presented in Chapter 5. Any SAT or CSP solver can play the role of the GPCS. As the performance of the SAT solver in the previous experiment was satisfactory and considering that the major strength of the FMRS regards its improved ability to count solutions we decided to implement a version of the FMRS using the Choco CSP solver as the GPCS component. Remember that Choco performed slightly better than SAT4J in the previous experiment for solution counting. In the next experiment we measure the benefits of the hybridization scheme implemented in the FMRS in terms of two operations: satisfiability checks and solution counting.

Goal: The purpose of this experiment was to measure the efficiency of the FMRS system in performing satisfiability checks and solution counting on collections of feature models against a pure constraint solver.

Benchmark: Supported the experiments 14 collections consisting of 10 satisfiable feature models each. The size of the models varied from one collection to another. For satisfiability tests 7 collections were used with model sizes of 500, 1000, 2000, 3000, 4000, 5,000, and 10,000 features, and 20% ECR for all models. For solution counting tests 7 collections were used including the real feature models in Table 6.1 and 6 other automatically-generated collections, i.e., 3 collections with model sizes of 100, 150, 200 and 20% ECR, and other 3 collections with sizes 500 (5% ECR), 1,000 (2% ECR), and 10,000 (0% ECR) features. In all models the odds for mandatory, optional, and grouped features of inclusive-OR and exclusive-OR groups were set to 25%, 35%, 20% and 20%, respectively. The models were not simplified prior to running the experiments.

Results & Analysis:

Satisfiability:

Table 6.7 shows the performance results for the FRMS and the CSP solver (CSP results copied from Table 6.5) for performing satisfiability checks on various collections of feature models. Once again, a time limit of 30 seconds was given to each solver to perform satisfiability checks on the sample models.

The FMRS succeeded in the satisfiability checks in 100% of the models analyzed

Table 6.7: Performance results of the FMRS and the CSP solver for **satisfiability tests** on various collections of feature models. Timeouts indicate lack of response within 30 seconds. Running times are average results of the successful cases.

Solver	Feature Model Size (20% ECR)						
	500	1,000	2,000	3,000	4,000	5,000	10,000
Choco 1.2 (CSP solver)							
Timeouts [%]	0	20	10	10	50	80	90
Running Times [ms]	14	20	61	239	78	53	197
FMRS (Choco as GPCS)							
Timeouts [%]	0	0	0	0	0	0	0
Running Times [ms]	15	43	146	248	518	1,242	2,967

Table 6.8: Performance results of the FMRS and the CSP solver for **counting solutions** on various collections of feature models. Timeouts indicate lack of response within 30 seconds. Running times are average results of successful cases. All models were satisfiable.

Solver	Feature Model Size					
	(20% ECR)			(5% ECR)	(2% ECR)	(0% ECR)
	100	150	200	500	1,000	10,000
FMRS						
Timeouts [%]	0	0	90	40	0	0
Running Times [ms]	330	9,506	8,015	9,960	9,796	5

even for very large models with 10,000 features with an average running time under 3 seconds. This was quite surprising since we did not expect much in terms of performance improvements for satisfiability checks. In fact, the running times were compatible with the pure CSP solution but the number of timeouts were reduced significantly. This suggests that the decisions made by the CSP solver playing the GPCS role were either quickly propagated or a conflict was found earlier than in a pure CSP solution. This likely avoided many unproductive searches. In addition, the number of decisions made by the CSP solver (GPCS) was reduced since now only extra constraint variables were considered which are usually a fraction of the total number of variables in the feature model. As a consequence, many variables remained uninstantiated after the satisfiability checks. For instance, about 30% of the variables in the models with 5,000 features remained uninstantiated upon the completion of the satisfiability checks. In a pure CSP solution those "skipped" variables (as well as their containing clauses) are likely to disturb the solver by potentially increasing the number of backtrackings which might have caused many timeout situations.

As an example, consider the feature model in Figure 6.5b in which only 5 (or 42%) features have been instantiated to confirm the satisfiability of the feature model. That is, once an assignment satisfies the extra constraint relation ($D \rightarrow E_1$) and is successfully propagated in the feature tree the satisfiability is of the entire feature model is confirmed by the FMRS. Instead, a pure CSP solution would still have to check the remaining feature tree clauses in order to come to a conclusion. It is known that a constraint solver does not have necessarily to instantiate all the variables in the problem during a satisfiability check. However, the hybridization scheme can significantly increase the number of uninstantiated variables.

Counting Solutions:

Table 6.8 shows the performance results for the FRMS for counting solutions on various collections of feature models within a time frame of 30 seconds. The FMRS succeeded in 100% of the cases (no timeouts) for models with up to 150 features and 20% ECR. Remember that the CSP solver struggled to handle models with 50 features generating timeouts in 80% of the cases. Considering only the cases where no timeouts occurred, the FMRS was able to handle models about 5 times larger than the CSP solver (from 30 to 150 features). For models with 200 features and 20% ECR the FMRS did not complete the counting in 90% of the cases.

As discussed earlier, the efficiency of a pure CSP-based solution for counting solutions is related to factors such as the total number of solutions in the feature model and the number of steps taken by the solver to find those solutions. Instead, the performance of the FRMS is only related to the number of solutions in the extra constraints as the solutions in the feature tree can be quickly counted by the FTRS, i.e., the FMRS's internal solver that handles feature tree operations. For details of the FTRS counting solutions algorithm please refer to Section 5.1.5.

Therefore, the FRMS can be especially effective for models with low ECR. Table 6.8 shows that models with 500 and 1,000 features (5% and 2% ECRs, respectively) had their solutions counted by the FMRS in 10 seconds or less. In particular, if the counting operation is applied only in the feature tree (0% ECR), models with 10,000 features are easily handled (average time of 5ms) (see last table column).

Figure 6.5 shows how the FMRS along with its two internal solvers, i.e., the FTRS and GPCS, count feature model solutions. In the figure, (a) shows a feature model with 12 features and a single extra constraint relation ($D \rightarrow E_1$). In (b), (c)

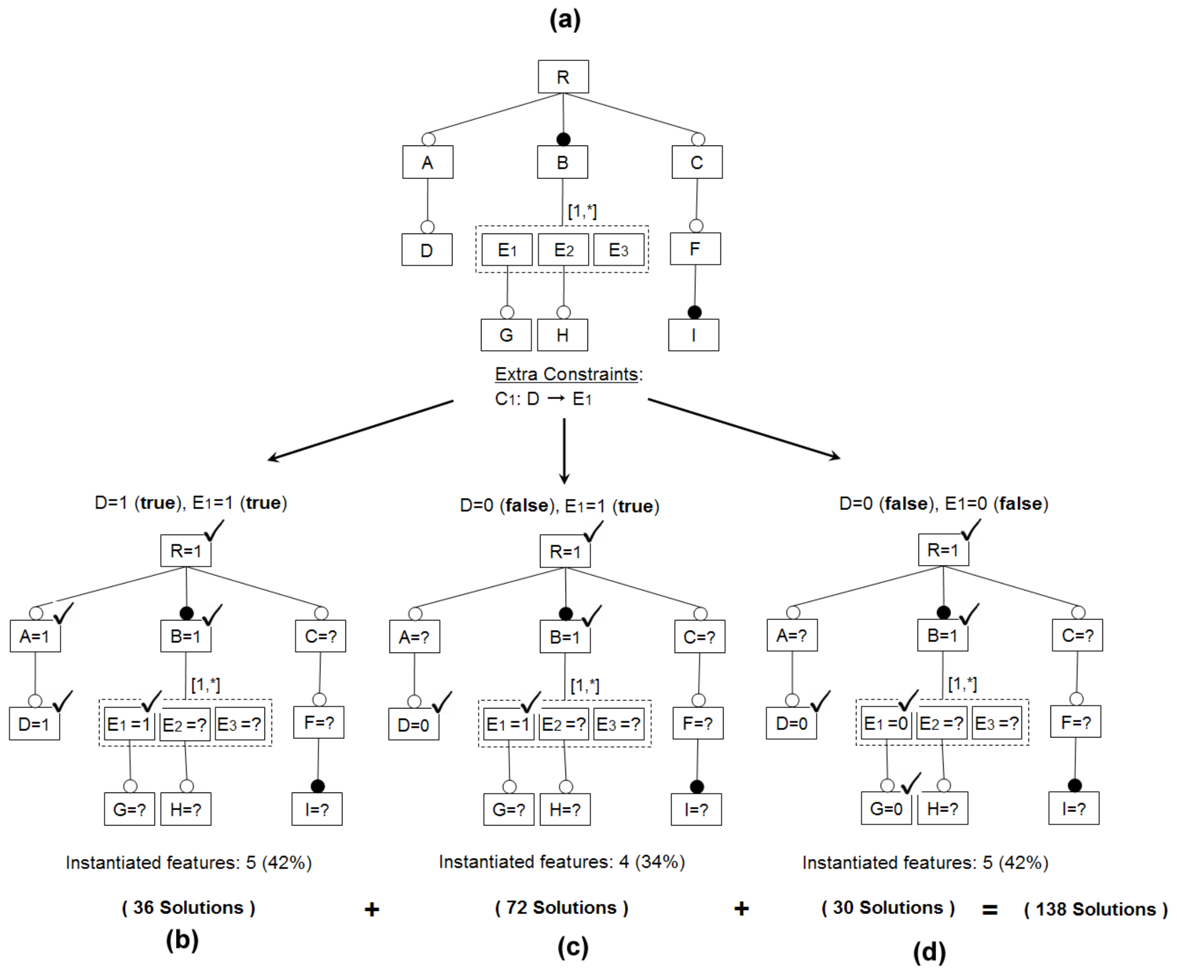

(a)
R
A
B
C
D
[1,*]
E1
E2
E3
F
G
H
I
Extra Constraints:
C1: D → E1
D=1 (true), E1=1 (true)
D=0 (false), E1=1 (true)
D=0 (false), E1=0 (false)
Instantiated features: 5 (42%)
Instantiated features: 4 (34%)
Instantiated features: 5 (42%)
(36 Solutions) + (72 Solutions) + (30 Solutions) = (138 Solutions)
(b)
(c)
(d)

Figure 6.5: How the FMRS system counts the solutions in a feature model. A feature model is shown in (a). For each solution found by the constraint solver in the extra constraints depicted in (b), (c) and (d) the FMRS cumulatively counts and adds the number of solutions in the "partially instantiated" feature tree to later obtain the total number of solutions in the feature model.

and (d) the feature model in (a) is “partially” instantiated after the propagation in the feature tree of each solution found by the GPCS in the extra constraint. For instance, in (b) the extra constraint solution (D=*true*, E_1=*true*) is propagated successfully in the feature tree thereby instantiating other features (see check marks in the figure for instantiated features). Once propagation succeeds, the feature tree is guaranteed to be consistent and satisfiable (property 5.2.1). The interesting fact is that only some features are instantiated (e.g. 5 or 42% in (b)) while many others are left uninstantiated. As a consequence, several solutions are represented in the feature tree in each step that can be quickly computed by the FTRS rather than by a slow search procedure of a pure CSP solution. By cumulatively counting and adding the solutions in each step (36, 72 and 30 solutions in (a), (b), and (c), respectively) the FMRS computes the total number of solutions in the feature model (138 solutions in the example).

As a result, the FMRS can be viewed as an alternative approach for those unfamiliar with BDDs for counting the solutions of medium size models or large models with low ECR. In addition, we are hopeful that the rationale of the FMRS, i.e., to combine efficient domain-specific algorithms in the feature tree with traditional SAT/CSP search procedures, can inspire researchers to explore other interesting contexts.

Summary:

- The FMRS was able to handle models about 5 times larger than the CSP solver to count problem solutions (from 30 to 150 features in the model).
- Large models with low ECR could also be handled satisfactorily, e.g., models with 500 (5% ECR) and 1,000 (2% ECR) features.
- The FMRS is an alternative approach for those unfamiliar with BDDs for counting problem solutions of medium size models or large models with low ECR.

6.5 Hardness and Phase Transition of Feature Model SAT Instances

We were intrigued during the experiments by the fact that the SAT solver performed extremely well in performing satisfiability checks for feature models containing up to 10,000 features and several extra constraints. This naturally arises the following question: Are SAT solvers always efficient in checking the satisfiability of feature models? If so, can we provide some explanations for that? We address this question in this section.

Recall that in Section 5.3 we showed that feature tree formulas can be solved in linear time by a typical SAT solver. This strongly suggests that it is the extra constraint formula that can potentially increase the hardness SAT instances derived from feature models. In the following, we provide a short background on the hardness of SAT problems and report on the results of empirical experiments to evaluate the hardness of feature model SAT instances.

6.5.1 Short Background

Research on the hardness of SAT problems has found many motivations in the past being the most prominent the needs of improving the performance of SAT algorithms such as the DPLL procedure [34]. As a result, many classes of SAT problems have been examined [42, 1, 70, 69, 87]. In particular, researchers have attempted to determine hardness threshold values for k-SAT [69, 38], i.e., the class of SAT problems consisting of CNF clauses containing exactly k literals. These thresholds were usually related to parameters such as the number of variables and clauses in the problem and represent crossover points in which an “easy” SAT instance becomes “hard”, and vice-versa. In this context, *hardness* is directly related to the number of steps required by a SAT solver to perform satisfiability checks. Obviously, the harder (easier) the instance the higher (lower) the number of steps required. In worst cases, the number of required steps is so large that it can never be performed in feasible time thus characterizing intractable instances.

An important discovery related the hardness of SAT instances to a phenomenon called *phase transition* [47, 78, 93]. The phase transition characterizes the transition of a SAT instance from a satisfiable to an unsatisfiable state given the variation of a specific order parameter. For k-SAT problems the prominent parameter considered has been the *clause density*, i.e., the ratio α of the number of clauses m to the number of variables n in the problem ($\alpha = \frac{m}{n}$). As the clause density increases so does the probability of k-SAT instances to become unsatisfiable. The notable finding was that during the phase transition an “easy-hard-easy” pattern was followed, i.e., a SAT solver that had been working efficiently starts to struggle up to a point that the problem becomes intractable and as the clause density value continues to grow the solver starts to perform efficiently again. The peak in hardness coincides with the 50% threshold point, i.e., the point where k-SAT instances switch from an "almost always satisfiable" state (underconstrained problem) to an "almost always unsatisfiable" state (overconstrained problem).

Nowadays, approximate bounds for the phase transition of uniform random 3-SAT problems are known. An instance of a uniform random 3-SAT formula is built by selecting three different random variables for each clause and negating each with probability $\frac{1}{2}$. The maximum number of clauses is then $8\binom{n}{3}$, where n represents the number of variables in the problem. For random 3-SAT the phase transition occurs for values of α varying from 3.42 to 4.506. The critical value is ≈ 4.25 that represents a 50% probability of satisfiability which, as mentioned, coincides with the peak in hardness of SAT algorithms. In other words, random 3-SAT instance containing n variables and $4.25 \times n$ clauses typically represent intractable SAT instances especially when the number of variables is large.

Based on these observations, our goal from now on is two fold: first, we want to find approximate phase transition thresholds for a certain class of feature model SAT instances; second, we want to examine the increase in hardness of those instances as they approximate the thresholds identified. To achieve these goals we conducted several tests summarized in the following experiment description.

6.5.2 Relating Hardness and Phase Transition

Experiment

Goal: The purpose of this experiment was to identify approximate phase transition threshold values for a certain class of feature model SAT instances and to examine the increase in hardness of those instances as they approximate the thresholds identified. We consider feature model formulas obtained by conjoining a feature tree formula and a uniform random 3-SAT formula representing the extra constraints attached to the tree. This is a quite reasonable configuration to consider giving that in practice extra constraint formulas usually consist of a combination of binary and ternary clauses. As a matter of fact, we are assuming the worst-case scenario (and the most likely to produce hard instances) in which all clauses are ternary. We consider "random" 3-SAT ensembles for which phase transition thresholds approximations are known. In addition, we consider feature trees containing mandatory, optional, inclusive-OR and exclusive-OR features with each type of feature having an equal probability to appear in the tree, i.e., 25% for each type. Also, we enforce that parent nodes have a minimum of 1 and a maximum of 6 children (branching

Table 6.9: Phase transition thresholds for feature model SAT instances

Size & ECR	50% Satisfiability Threshold			
	(10% ECR)	(20% ECR)	(30% ECR)	Size Range
1,000	2.35	1.75	1.40	[1.40 - 2.35]
2,000	2.15	1.45	1.10	[1.10 - 2.15]
3,000	1.90	1.30	1.00	[1.00 - 1.90]
5,000	1.65	1.05	0.80	[0.80 - 1.65]
10,000	1.30	0.75	0.50	[0.50 - 1.30]
ECR Range	[1.30 - 2.35]	[0.75 - 1.75]	[0.5 - 1.4]	[0.50 - 2.35]

factor). This is important since realistic feature trees are not "flat" but rather have the features distributed throughout the many levels of the tree. For the purposes of the experiment, we developed a tool that based on given ECR and clause density parameters randomly selects variables in the feature tree and creates a random 3-SAT formula with those variables.

Benchmark: One hundred models were generated "on-the-fly" for each combination of size (1,000, 2,000, 3,000, 5,000 and 10,000), ECR (10%, 20% and 30%) and clause density (from 0.1 to 3.5 in increments of 0.1) to support the experiment. In addition, other density ranges (from 3.6 to 5.0) were considered for models with 10,000 features and 30% ECR. The clause density refers to the density of clauses in the extra constraints not in the feature model. This makes sense as we want to examine how feature model SAT instance increase in hardness as new clauses are added to the extra constraints. In total, 54,000 models were generated during the experiment some of which are available online for download at [63]. The SAT solver (SAT4J [13]) was given 30 seconds to complete satisfiability checks on generated models.

Results & Analysis:

Phase Transition

Table 6.9 shows the 50% satisfiability thresholds found for several feature models of varying sizes and ECRs, i.e., extra constraint clause densities for which the feature model formula has equal probability of being satisfiable and unsatisfiable. The thresholds were calculated for each combination of model size and ECR by considering only density values within the range in which at most 70% and at least 30% of the models analyzed represented satisfiable instances. The median of the density values was defined as the threshold. For instance, consider the density values in the range from 2.0 to 2.7 observed for feature models with 1,000 features

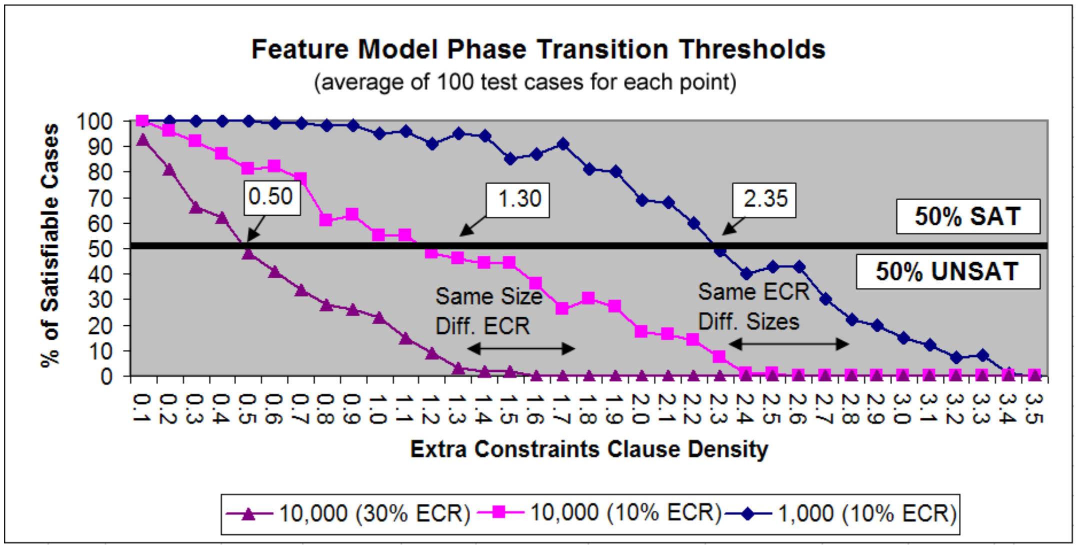


Figure 6.6: Threshold phase transition values for varying model size and ECR parameters

and 10% ECR. The median was then the average of the two middle values 2.3 and 2.4 which resulted in the threshold value of 2.35 (see second row/column on table). This means that if the CNF formula corresponding to a feature tree containing 1,000 nodes is conjoined to a random 3-SAT formula containing 100 of the feature tree variables (10% ECR) and 235 clauses (density of 2.35) the resulting SAT instance has 50% probability of being (un)satisfiable. This "satisfiability crosspoint" is also illustrated in Figure 6.6 (see value 2.35 in the figure). For densities higher (lower) than 2.35 SAT instances are more likely to be unsatisfiable (satisfiable).

Column "Size Range" in Table 6.9 depicts the threshold ranges for a given model size. For example, for models with 1,000 features (second row) the threshold decreases from 2.35 to 1.40 as the ECR increases. That is, *for a fixed size, the higher the ECR the earlier the feature model instance reaches the threshold and becomes unsatisfiable*. This is quite expected given that the number of variables and clauses in the extra constraints increases and makes the formula more constrained. Similarly, row "ECR Range" shows threshold ranges but now for fixed ECR values. For instance, densities decrease from 2.35 to 1.30 for models the 10% ECR as the size of the models increases. That is, *for a fixed ECR, the larger the model the earlier the feature model instance reaches the 50% satisfiability threshold and becomes unsatisfiable*. This is interesting since the number of variables in the extra constraints is equally proportional to the size of the models but yet larger models become unsatisfiable earlier. Figure 6.6 illustrates both cases, i.e., thresholds for models with same size but different ECRs and for models with same ECR but different sizes. For instance, given two models with 10,000 features and ECRs of 10% and 30%, the one with higher ECR reaches the 50% threshold first (see values 0.50 and 1.30 in the figure). Similarly, given two models with ECR of 10% and containing 1,000 and 10,000 features, the one with higher number of features reaches the 50% threshold first (see values 1.30 and 2.35 in the figure).

This is the first time phase transition has been studied and threshold parameters have been calculated in the context of feature model formulas. The results obtained provide a solid ground for further analysis of the hardness of feature model SAT instances as discussed next.

Hardness

As mentioned previously, for many classes of SAT problems (e.g. k-SAT) the peak in hardness occurs near the 50% satisfiability threshold when the number of steps performed by SAT algorithms grow exponentially on the size of the problem until they become too large to be processed in feasible time. Therefore, we want to examine how the hardness of feature model SAT instances increases as those instances approximate the thresholds identified.

We considered very large feature model SAT instances (10,000 features) with a high ECR value (30%) that should represent challenging SAT problems, at least when compared to most of realistic feature model instances. The threshold found for models with 10,000 features and 30% ECR was 0.5 as shown in Table 6.9. Figure 6.7 shows the running times (average of 100 test cases for each point) of the SAT solver (SAT4J) to perform satisfiability checks on those models for various clause density values. As it can be observed in the figure, the SAT solver was extremely efficient in handling all models regardless of the density considered. That is, the average running time for satisfiability checks was never higher than 61ms. In particular, there was no decline in performance of the SAT solver for instances near the 0.5 threshold (see (A) in the figure). This provides a strong evidence that *typical feature model formulas represent "easy" SAT instances for any standard SAT solver.*

Another interesting point is illustrated in Figure 6.7-B when the extra constraint formula (in isolation) reaches its threshold (from 3.42 to 4.506) and becomes computationally hard to solve. In fact, we ran the SAT solver for the extra constraints in isolation and observed many cases of unfeasibility (several hours without a response from the solver). As depicted in the figure, the experiment shows that *when such "hard" extra constraint formulas are conjoined to a feature tree formula the resulting feature model formula is unsatisfiable and trivially solvable.* In fact, cases

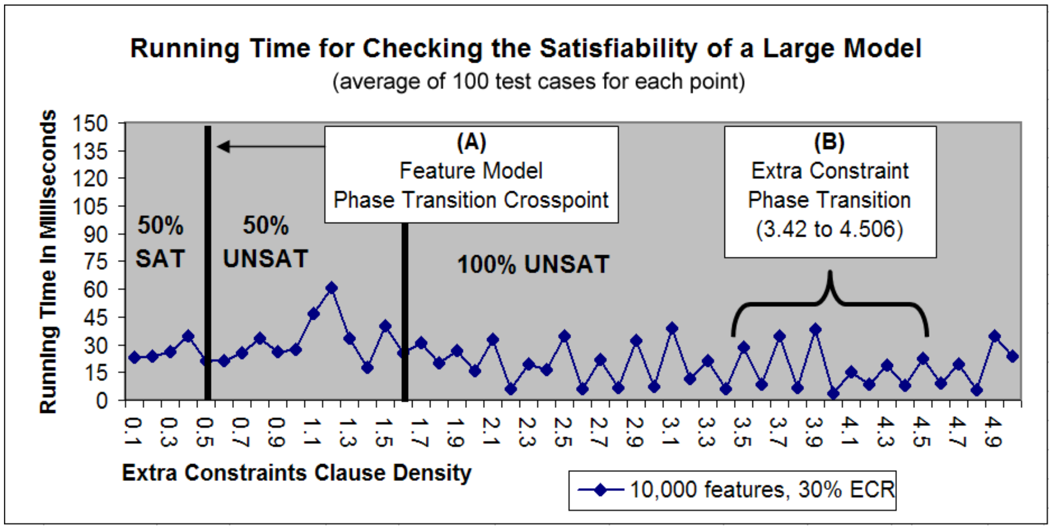


Figure 6.7: Running times of satisfiability checks for feature models with 10,000 features and 30% ECR

of 100% unsatisfiability started at density 1.6, i.e., all models with density 1.6 or higher were unsatisfiable.

We believe that the major reasons behind the consistent efficiency of SAT solvers in dealing with feature model formulas is due to the fact that the largest number of relations in the feature model, i.e., the feature tree, represent a very easy SAT instance that can be solved by a SAT solver in linear time in the size of the model. Because the extra constraints typically uses only a fraction of the variables in the feature tree the resulting conjoined formula is still an "easy" SAT instance, even when the extra constraints are considered "very hard" instances in isolation.

Despite the fact that the experiments considered a particular class of feature model formulas, i.e., models composed of random 3-SAT formulas as extra constraints, we believe that those models represent even harder SAT instances than what we expect from most of practical models. In fact, in most of the real feature models that we studied the extra constraints usually consisted of a mix of binary and ternary constraints what makes the problems potentially easier to solve.

We are very satisfied with the results of the experiments as they seem to provide better explanations to some challenging research questions recently posed related to the manipulation of very large feature models (particularly those containing up to 10,000) [7]. We have shown that certain operations such as satisfiability can be successfully handled by current technologies such as SAT solvers.

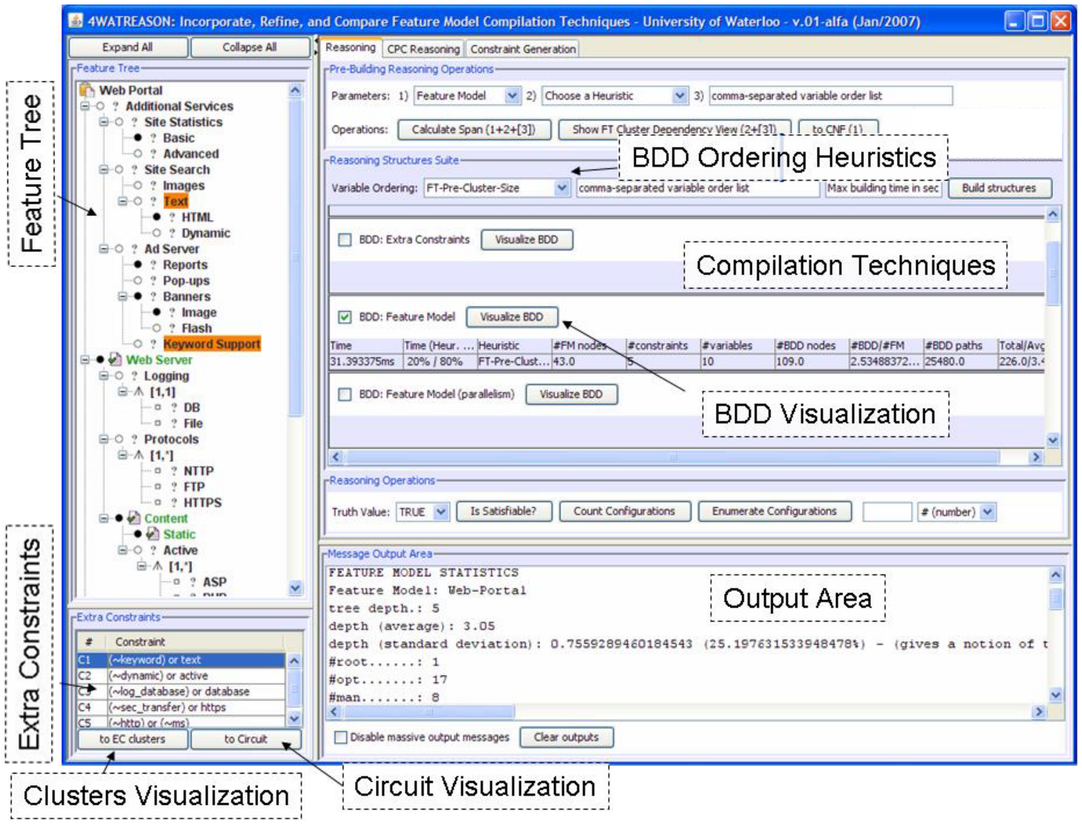
Summary:

- Various 50% satisfiability thresholds could be identified for feature models with different sizes and ECRs.
- For a fixed size, the higher the ECR the earlier the feature model instance reaches the 50% satisfiability threshold and becomes unsatisfiable.
- For a fixed ECR, the larger the model the earlier the feature model instance reaches the 50% satisfiability threshold and becomes unsatisfiable.
- SAT solvers can efficiently handle (61ms or less) feature models with up to 10,000 features and 30% ECR for which extra constraints are represented by random 3-SAT formulas. Hence, typical feature model formulas represent "easy" SAT instances for any standard SAT solver.
- When known "hard" extra constraint SAT instances (uniform 3-SAT) are conjoined to a feature tree formula the resulting feature model formula is unsatisfiable and trivially solvable.

6.6 Tool Support

Several tools were developed in our research to support automated model generation and analysis, to run the various experiments, and to visualize and reason on various aspects of feature models. The tools were built on top of a Java library we developed to provide proper interfaces for manipulating feature models, clausal and propositional formulas, variable ordering heuristics, Boolean circuits, hypergraphs, graphical interfaces for feature models, circuits and BDDs, and several reasoning techniques based on BDDs, and SAT and CSP solvers. Currently, the library contains 157 Java classes distributed in 12 packages and has about 17,144 lines of code. A tool called 4WATREASON was built to illustrate most of the components available in the library and can be accessed online at [63].

A screenshot of the main window of the 4WATREASON tool is depicted in Figure 6.8. On the left-hand side, the Web-Portal feature model listed in Table 6.1 is shown loaded in the tool. The tree structure represents the feature tree and the left-bottom table shows the extra constraints attached to the feature tree. Below this table, two buttons are provided that allow the visualization of feature model clusters (see Figure 6.4 for screenshots) and Boolean circuits (see Figure 2.7 for screenshots) representing the feature models. On the right-hand side, several BDD variable ordering heuristics can be explored to build BDDs for feature models.


4WATREASON: Incorporate, Refine, and Compare Feature Model Compilation Techniques - University of Waterloo - v.01-alfa (Jan/2007)
Feature Tree
Extra Constraints
Clusters Visualization
Circuit Visualization
BDD Ordering Heuristics
Compilation Techniques
BDD Visualization
Output Area
Expand All
Collapse All
Reasoning
CPC Reasoning
Constraint Generation
Pre-Building Reasoning Operations
Parameters: 1) Feature Model 2) Choose a Heuristic 3) comma-separated variable order list
Operations: Calculate Span (1+2+[3]) Show FT Cluster Dependency View (2+[3]) to CNF (1)
Reasoning Structures Suite
Variable Ordering: FT-Pre-Cluster-Size comma-separated variable order list Max building time in sec Build structures
BDD: Extra Constraints Visualize BDD
BDD: Feature Model Visualize BDD
BDD: Feature Model (parallelism) Visualize BDD
Reasoning Operations
Truth Value: TRUE Is Satisfiable? Count Configurations Enumerate Configurations # (number)
Message Output Area
Disable massive output messages
Clear outputs
to EC clusters
to Circuit
Web Portal
Additional Services
Site Statistics
Basic
Advanced
Site Search
Images
Text
HTML
Dynamic
Ad Server
Reports
Pop-ups
Banners
Image
Flash
Keyword Support
Web Server
Logging
[1,1]
DB
File
Protocols
[1,*]
NTTP
FTP
HTTPS
Content
Static
Active
[1,*]
ASP

Time	Time (Heur. ...	Heuristic	#FM nodes	#constraints	#variables	#BDD nodes	#BDD/#FM	#BDD paths	Total/Avg
31.393375ms	20% / 80%	FT-Pre-Clust...	43.0	5	10	109.0	2.53488372...	25480.0	226.0/3.

#	Constraint
C1	(~keyword) or text
C2	(~dynamic) or active
C3	(~log_database) or database
C4	(~sec_transfer) or https
C5	(~http) or (~ms)

```
FEATURE MODEL STATISTICS
Feature Model: Web-Portal
tree depth.: 5
depth (average): 3.05
depth (standard deviation): 0.7559289460184543 (25.197631533948478%) - (gives a notion of t
#root......: 1
#opt.......: 17
#man.......: 8
```

Figure 6.8: 4WATREASON Tool Main Screen

In addition, various reasoning techniques based on BDDs, SAT and CSP solvers for reasoning on feature trees, the extra constraints or the entire feature model are available. Multiple techniques can be used simultaneously which facilitates a direct comparison of their strengths and weaknesses. The FRMS, our proposed hybrid reasoner, is also part of the techniques available. In the right-bottom, an output area displays the results of applying the reasoning operations on the feature model. Various operations are supported including satisfiability checks, and the enumeration and counting of solutions using the different reasoning techniques. It is also possible to select and deselect features in the feature model while performing the operations.

For more screenshots and an online demo of the 4WATREASON tool please refer to [63].

Chapter 7

Conclusion

In this thesis, we have discussed two powerful techniques that can be used to automate support for feature model reasoning, i.e., SAT solvers and BDDs. We have shown that by better understanding the mechanics of BDDs and SAT solvers and the sensitive issues related to these techniques, we can provide more accurate explanations for the space and/or time (in)tractability of these techniques in the feature modeling domain, and enhance the algorithmic performance of these techniques for reasoning on feature models.

In particular, we discussed the space intractability problem associated with BDDs. That is, BDDs are very sensitive to the order of their variables and a bad ordering can lead to BDDs of intractable sizes. Since finding an optimal order is NP-hard, we argued that the problem has been typically approached by using heuristics. Hence, in Chapter 4 we explored several relevant properties of the feature modeling domain that should be considered when ordering the variables of BDDs compiled from feature models. Based on the insights provided, we then proposed two novel BDD variable ordering heuristics that considered clustering and sorting the nodes in the feature model tree recursively and traversing the tree in DFS in order to produce a good quality ordering for BDD variables. We showed empirically that our heuristics could produce orders that were consistently better than state-of-the-art heuristics for the feature modeling domain. In particular, we showed that the orders produced by our heuristics could lead to BDDs about 10 times smaller than BDDs produced by other heuristics considering real and automatically-generated feature models. As a result, feature models twice as large (2,000 features) as those of previous heuristics (1,000 features) could be built. The heuristics were implemented as part of an extensible algorithmic infrastructure that allows new heuristics to be easily incorporated as further properties of feature models are examined.

Furthermore, we explored several properties of feature models and showed how these properties can be used to develop efficient domain-specific algorithms to reason on a subset of the feature model. We also showed how these domain-specific algorithms could be integrated with existing SAT algorithms into hybrid solutions that could deliver improved performance for certain kinds of reasoning operations on feature models. We showed empirically that some of the hybrid algorithms can indeed be much more attractive than a pure SAT-based solution. For instance, a hybrid algorithm was developed to count the number of solutions in a feature model. Empirical experiments showed that the hybrid algorithm was able to address models up to 5 times larger (150 features) than a pure SAT solution (30 features). Ultimately, our goal was to inspire other researchers to examine further properties of feature models that can lead to the development of new efficient domain-specific algorithms and eventually to the integration of these algorithms with existing techniques such as SAT and BDDs.

In addition, we showed empirically that SAT instances derived from feature models could be easily solved by any standard SAT solver. For this purpose, we explored the correlation between hardness and phase transition for a class of feature model SAT instances and showed that during the phase transition, i.e., when there is an equal likelihood of a feature model to be satisfiable and unsatisfiable, no significant changes in the performance of the SAT solver were observed. That is, the solver remained extremely efficient even during the phase transition. This suggests that SAT instances derived from feature models containing up to 10,000 features and 30% ECR can be easily solved by a typical SAT solver. Since we expect realistic models to be much simpler than those used in the experiments we are confident about the suitability of SAT solvers to handle SAT problems derived from the feature modeling domain.

7.1 Addressing The Research Questions

In Chapter 1 we posed important research questions that shall now be answered.

1. Can the size of BDDs for realistic feature models ever become intractable? If so, what are the current limits?

Answer: Yes. The best heuristic examined prior to developing our own heuristics, i.e., a DFS of the feature model, was able to compile feature models with up to 1,000 features successfully. For larger models, the compilation process started to raise “memory overflow” errors consistently. For instance, only 30% of the models containing 2,000 could be compiled successfully by DFS. The other heuristics examined performed even poorer than DFS on average.

2. Can these limits be improved? If so, how and by how much?

Answer: Yes. We proposed two new heuristics, i.e., *Pre-CL-Size* and *Pre-CL-MinSpan* that were able to reduce significantly the size of BDDs compiled from feature models. The heuristics explored several structural properties of feature models and applied techniques such as clustering and sorting in the feature model to reduce the distance of connected variables in the final ordering. As a result, an average reduction of about 10 times in BDD size was observed for the new heuristics for BDD compilations of real and generated feature models. This enabled feature models twice as large (2,000 features) as those handled by previous heuristics (1,000 features) to be compiled. By using orders produced by heuristic *Pre-CL-MinSpan* we were able to compile models containing up to 2,500 features in 70% of the cases.

3. Are SAT solvers always efficient in checking the satisfiability of feature models? If so, can we provide some explanations for that?

Answer: Considering that we analyzed a class of models that in theory is harder (i.e., larger, with more relations) than most of the realistic models coped with by a SAT solver, we are confident that the answer is “yes”. We showed that a SAT solver can check the satisfiability of feature trees in linear time in the number of features in the tree. This suggested that it was the relations in the extra constraint that could potentially make feature model SAT instances hard to solve. We explored the correlation between the phase transition and the hardness of SAT instances for a given class of feature models for which the extra constraints were represented by random 3-SAT ensembles. We showed empirically for these instances that during the phase transition the performance of the SAT solver remained stable, i.e., the solver was still completing satisfiability checks very efficiently (i.e. in low milliseconds). The models considered in the experiments contained up to 10,000 and 30% ECR. In addition, we showed that even when the extra constraint formula became intractable in isolation, when this formula was conjoined to the feature tree the resulting feature model formula was unsatisfiable and trivially solvable. We believe that our evaluations increase substantially the level of confidence on the use of the SAT technology to handle most of the feature models in practice.

4. How can we take advantage of domain knowledge to improve the performance of algorithms to reason on feature models?

Answer: In the feature model, the feature tree arranges the variables into hierarchical relations. This facilitates devising recursive algorithms to traverse the variables and their relations and can greatly facilitate some kinds of computations. One such example is the counting of the number of legal configurations in the feature tree that can be computed in linear time in the size of the tree. Instead, a SAT solver would have to traverse all of these configurations, which ultimately involves an exponential number of steps in the size of the feature tree, in order to perform the same operation. By integrating the linear algorithm for counting feature tree configurations with a SAT algorithm to count configurations in the extra constraint we were able to improve the overall performance of the algorithm for counting the number of legal configurations in a feature model. We are confident that other properties of feature models exist that can lead to improved hybrid algorithms to reason on these models.

5. What kind of domain-specific algorithms can be developed and what are the improvements in performance in comparison to pure SAT solutions?

Answer: We gave an example of an improved algorithm for counting the valid configurations in a feature model. The algorithm was able to handle feature models up to 5 times larger (150 features) than those addressed by a pure SAT solution (30 features). Again, we are confident that many new efficient algorithms can be developed by further examining properties of feature models.

7.2 Future Work

In the following, we discuss interesting research opportunities that naturally extend the ideas proposed in our work.

New BDD variable ordering heuristics

We are optimistic that some improvements can be done to the two novel BDD variable ordering heuristics proposed in Chapter 4. Currently, both heuristics implement a pre-order traversal of the feature tree in order to produce variable orders. However, our empirical experiments have shown that post-order can be as good as pre-order in producing compact BDD patterns. Hence, we envision a new heuristic that combines both pre- and post-order traversals depending on which strategy is more advantageous to minimize the distance of extra constraint variables at a given branch of the feature tree. A possible implementation of this idea could be to run a decision procedure at each cluster in the feature model to decide which strategy should be followed to traverse the subtrees in the cluster. However, there is a difficulty to overcome in this approach. That is, the fact that the feature tree is a hierarchical structure can make a good traversal decision at a given cluster of the tree a bad choice for other clusters at lower levels in the same branch of the tree. Hence, the decision procedure would have to take into account factors such as the number of clusters in a given branch of the feature tree and the density of the relations within these clusters to decide for an overall advantageous traversal strategy.

Another possible improvement to the heuristics could be to combine their approach to sort the nodes within a cluster into a single strategy. Currently, heuristic *Pre-CL-Size* uses a sort-by-size strategy that can be very effective for sparsely-connected clusters with large subtrees while heuristic *Pre-CL-MinSpan* handles better highly-connected clusters by placing nodes connected to many others in central positions in the cluster. These two approaches could be combined into a single strategy that evaluates whether a given cluster is either sparsely or highly connected and applies the most advantageous sorting procedure.

BDD constraint ordering heuristics

In this thesis, we did not discuss strategies for ordering constraints for BDD construction. In fact, we assumed a pre-order traversal of the feature tree followed by a natural-order traversal of the extra constraints to order the constraints. While the order of the constraints has no influence on the final size of the BDD it can have a great impact on the intermediate sizes the BDD structure reaches during the BDD construction. Therefore, it is possible that the final size of a given BDD is tractable but yet the BDD cannot be built because it reaches an infeasibly-large size during construction.

In a preliminary experiment, we noticed that the size of the BDD remains manageable as the relations in the feature tree are processed and only when the extra constraints start to be taken into account the growth in size starts to be a real problem. Therefore, we think that a promising approach could be to group and process together related constraints in the feature tree and in the extra constraints rather than postponing to the end the processing of the extra constraints. A possible alternative to implement this approach could be to take advantage of the clustering of child nodes in the feature tree to process the extra constraint relations associated with each cluster following the processing of the feature tree relations in the cluster. By doing so, whenever the relations in the subtrees of a cluster are processed so are the extra constraints associated with the cluster. This approach might lead to a more manageable BDD size growth as a group of related constraints are processed together. However, more detailed analysis and empirical experiments have to be carried out to evaluate the real benefits of this approach.

Parallel algorithms for BDD construction

An alternative to speed up the BDD construction process could be to design a parallel algorithm. A good parallel algorithm would quickly build and merge intermediate BDD structures into a final consolidated structure. However, merging intermediate BDDs efficiently can be a challenging task especially if these BDDs share many variables. We think that the clustering procedure applied to the feature tree can give a good hint on how to split the building process into separate concurrent tasks and yet merge the intermediate BDD structures produced by each task efficiently. For instance, consider two clusters A and B of the same parent node in the feature tree. Clearly, clusters A and B share no variables or relations in their subtrees otherwise they would have been combined into a single cluster by the clustering algorithm. Therefore, intermediate BDDs for clusters A and B could be processed in parallel and merged cheaply afterwards. Since the feature tree is a recursive structure this strategy could also be applied recursively by merging intermediate BDDs bottom-up until the root node is reached. We developed a preliminary parallel algorithm based on these ideas and were very encouraged by the results. The merging strategy seemed to work very well. However, the BDD library we used, i.e., the JavaBDD, does not support concurrency as it does not synchronize access to shared resources and uses a shared memory space for BDD manipulation. Hence, we were forced to make use of locks to implement our parallel algorithm and the final results were not as good as we expected. Despite this fact, we are convinced that the clustering of the feature tree can indeed support the development of parallel algorithms for BDD construction.

Domain-specific algorithms, SAT, and BDDs

In Chapter 5 we discussed several properties of feature trees that led to the development of highly efficient algorithms to reason on those trees. We also showed how these algorithms could be integrated with a SAT solver to form a hybrid solution for reasoning on feature models. Our major goal was to provide as many insights as possible to inspire the development of new algorithms in the future. The strategy for building new algorithms can be as follows. First, feature tree properties must be identified that could be used to build an efficient reasoning procedure. Second, the procedure is built by exploring the properties identified. Third, the algorithm developed is integrated with a SAT solver (or another external system such as a BDD) that will take care of the extra constraints to form a hybrid reasoning procedure for the feature model. The integration is also based on feature model properties such as those discussed in Chapter 5. The rationale is that the feature tree algorithm developed will be efficient enough to improve the overall performance of the hybrid algorithm. In this thesis, we gave an example of such an algorithm that computes the number of legal configurations in a feature tree. The algorithm was further integrated with a SAT solver to address feature models and, as we showed empirically, can address models up to 5 times larger than those handled by a pure SAT solution.

We believe that there are many opportunities for developing efficient domain-specific algorithms for feature models if the "relevant" properties are identified. For instance, there are quite a few properties that can encourage the development of algorithms to detect "dead" features on feature models. For instance, it can be proved that feature trees do not have "dead" features (proof is omitted here). Hence, it is the extra constraint relations that can cause the feature model to contain "dead" features. Second, if a given feature is "dead" in the feature tree so are all its descendants. Therefore, computations of related "dead" features is usually cheap. Most likely it is possible to build an algorithm that analyzes the extra constraints to extract some facts, prunes the feature tree according to the facts found, checks whether or not features are "dead" in the feature tree, and finds all related "dead" features. We plan to sketch a preliminary version of this algorithm as a next step in our research.

Bibliography

[1] Dimitris Achlioptas, Lefteris M. Kirousis, Evangelos Kranakis, and Danny Krizanc. Rigorous results for random (2 + p)-SAT. *Theoretical Computer Science*, 265:109–129, 2001. 139

[2] Fadi A. Aloul, Igor L. Markov, and Karem A. Sakallah. FORCE: a fast and easy-to-implement variable-ordering heuristic. In *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*, pages 116–119, New York, NY, USA, 2003. ACM. 28

[3] Vander Alves, Rohit Gheyi, Tiago Massoni, Uirá Kulesza, Paulo Borba, and Carlos Lucena. Refactoring product lines. In *GPCE '06: Proceedings of the 5th International Conference on Generative Programming and Component Engineering*, pages 201–210. ACM Press, 2006. 2, 32, 45

[4] Henrik Reif Andersen. *Binary Decision Diagrams.* Department of Information Technology, Technical University of Denmark, Lyngby, Denmark, 1997. Lecture notes for 49285 Advanced Algorithms E97, `http://www.itu.dk/people/hra/notes-index.html`. 21

[5] MichałAntkiewicz and Krzysztof Czarnecki. FeaturePlugin: Feature modeling plug-in for Eclipse. In *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004. Paper available from `http://www.swen.uwaterloo.ca/~kczarnec/etx04.pdf`. Software available from `gp.uwaterloo.ca/fmp`. 2, 3, 4, 39, 43, 47

[6] Pranav Ashar, Abhijit Ghosh, and Srinivas Devadas. Boolean satisfiability and equivalence checking using general binary decision diagrams. In *ICCD '91: Proceedings of the 1991 IEEE International Conference on Computer Design on VLSI in Computer & Processors*, pages 259–264, Washington, DC, USA, 1991. IEEE Computer Society. 38

[7] Don Batory, David Benavides, and Antonio Ruiz-Cortes. Automated analysis of feature models: challenges ahead. *Communications of the ACM*, 49(12):45–47, 2006. 6, 31, 112, 144

[8] Don S. Batory. Feature models, grammars, and propositional formulas. In *Software Product Lines, 9th International Conference, SPLC 2005, Rennes, France, September 26-29, 2005, Proceedings*, volume 3714 of *Lecture Notes in Computer Science*, pages 7–20. Springer, 2005. 2, 3, 4, 16, 42, 111

[9] Thomas Bednasch, Christian Endler, and Markus Lang. CaptainFeature, 2002-2004. Tool available on SourceForge at `https://sourceforge.net/projects/captainfeature/`. 2, 41

[10] David Benavides, Antonio Ruiz Cortes, Pablo Trinidad, and Sergio Segura. A survey on the automated analyses of feature models. In José Riquelme and Pere Botella, editors, *JISBD 2006: XV Jornadas de Ingeniería del Software y Bases de Datos*, Barcelona, 2006. 3, 31, 37

[11] David Benavides, Antonio Ruiz-Cortes, and Pablo Trinidad. Using constraint programming to reason on feature models. In *Proceedings of the The 17th International Conference on Software Engineering and Knowledge Engineering (SEKE'05), Taipei, Taiwan, Republic of China*, 2005. 36, 111

[12] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortes. Automated reasoning on feature models. In *Proceedings of the 17th Conference on Advanced Information Systems Engineering (CAiSE'05), Porto, Portugal, 2005*, LNCS. Springer, 2005. 2, 3, 4, 36, 37, 42

[13] Daniel Le Berre, Anne Parrain, Olivier Roussel, and Lakhdar Sais. *SAT4J: A satisfiability library for Java*, 2005. `http://www.objectweb.org/phorum/download.php/16,291/sat4j-D-Le-Berre.pdf`. 100, 110, 141

[14] Danilo Beuche. *Composition and Construction of Embedded Software Families*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, Germany, December 2003. Available from `http://diglib.uni-magdeburg.de/Dissertationen/2003/danbeuche.pdf`. 111

[15] Danilo Beuche. pure::variants Eclipse Plugin. User Guide. pure-systems GmbH. Available from `http://web.pure-systems.com/fileadmin/downloads/pv_userguide.pdf`, 2004. 2, 40

[16] Danilo Beuche and Mark Dalgarno. Software product line engineering with feature models. In *pure-systems and Software Acumen*, pages 9–17, 2006. Available at: `http://www.methodsandtools.com/PDF/mt200604.pdf`. 111

[17] Danilo Beuche, Holger Papajewski, and Wolfgang Schröder-Preikschat. Variability management with feature models. *Sci. Comput. Program.*, 53(3):333–352, 2004. 3, 46

[18] Armin Biere, Alessandro Cimatti, Edmund Clarke, Fujita, and Yunshan Zhu. Symbolic model checking using SAT procedures instead of BDDs. *Design Automation Conference*, 0:317–320, 1999. 38

[19] Beate Bollig and Ingo Wegener. Improving the variable ordering of OBDDs is NP-Complete. *IEEE Transactions on Computers*, 45(9), September 1996. 24, 47

[20] Yves Bontemps, Patrick Heymans, Pierre-Yves Schobbens, and Jean-Christophe Trigaux. Semantics of FODA feature diagrams. In Tomi Männistö and Jan Bosch, editors, *Proceedings SPLC 2004 Workshop on Software Variability Management for Product Derivation – Towards Tool Support*, pages 48–58. Technical Report 6 – HUT-SoberIT-C6, August 2004. Available from `http://www.soberit.hut.fi/SPLC-DWS/`. 111

[21] Franc Brglez and Fujiwara. A neutral netlist of 10 combinatorial benchmark circuits and a target translator in FORTRAN. In *International Symposium on Circuits and Systems, Special Session on ATPG and Fault Simulation*, 1985. 27

[22] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, August 1986. 21

[23] Cambazard Hadrien et al. Choco: constraint programming system, 2003. Website: `http://sourceforge.net/projects/choco/`. 100, 110

[24] Craig Cleaveland. *Program Generators with XML and Java.* Prentice-Hall, Upper Saddle River, NJ, 2001. 10

[25] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns.* Addison-Wesley, Boston, MA, 2001. 1

[26] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications.* Addison-Wesley, Boston, MA, 2000. 1, 10

[27] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Online Proceedings of the 2nd OOPSLA03 Workshop on Generative Techniques in the Context of MDA*, Anaheim, Oct 2003. 111

[28] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration using feature models. In Robert L. Nord, editor, *Software Product Lines: Third International Conference, SPLC 2004, Boston, MA, USA, August 30-September 2, 2004. Proceedings*, volume 3154 of *Lecture Notes in Computer Science*, pages 266–283, Heidelberg, Germany, 2004. Springer-Verlag. 35

[29] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Formalizing cardinality-based feature models and their specialization. *Software Process Improvement and Practice*, 10(1), 2005. Special issue on Software Variability: Process and Management, `http://swen.uwaterloo.ca/~kczarnec/spip05a.pdf`. 111

[30] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged configuration through specialization and multi-level configuration of feature models. *Software Process Improvement and Practice*, 10(2):143–169, 2005. `http://swen.uwaterloo.ca/~kczarnec/spip05b.pdf`. 2, 39

[31] Krzysztof Czarnecki and Chang Hwan Peter Kim. Cardinality-based feature modeling and constraints: a progress report. In *International Workshop on Software Factories*, San Diego, California, Oct 2005. Paper available at `http://softwarefactories.com/workshops/OOPSLA-2005/SoftwareFactoryWorkshopAnnouncement.htm`. 36

[32] Krzysztof Czarnecki and Andrzej Wasowski. Feature models and logics: There and back again. In *Proceedings of 10th International Software Product Line Conference (SPLC 2007)*. IEEE Press, 2007. 3, 16, 31, 42

[33] P. Trinidad D. Benavides, S. Segura and A. Ruiz-Cortes. FAMA: Tooling a framework for the automated analysis of feature models. *VAMOS 2007*, page 129134, 2007. 3, 43, 47

[34] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962. 16, 21, 139

[35] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960. 16, 21

[36] Deepak Dhungana, Paul Grunbacher, and Rick Rabiser. Decisionking: A flexible and extensible tool for integrated variability modeling. In *First International Workshop on Variability Modelling of Software-intensive Systems*, January 16-18 2007. 2, 41

[37] Alexander Felfernig, Gerhard Friedrich, Dietmar Jannach, and Markus Zanker. Towards distributed configuration. In *KI '01: Proceedings of the Joint German/Austrian Conference on AI*, pages 198–212, London, UK, 2001. Springer-Verlag. 111

[38] Ehud Friedgut. Sharp thresholds of graph properties, and the k-sat problem. *J. Amer. Math. Soc*, 12:1017–1054, 1999. 139

[39] Masahiro Fujita, Hisanori Fujisawa, and Nobuaki Kawato. Evaluation and improvement of boolean comparison method based on binary decision diagrams. In *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pages 2–5, nov 1988. 27

[40] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A theory for feature models in alloy. In *First Alloy Workshop*, pages 71–80, Portland, United States, November 2006. 3, 32, 45

[41] Rohit Gheyi, Tiago Massoni, and Paulo Borba. A complete and minimal set of algebraic laws for feature models. In *Brazilian Symposium on Programming Languages (SBLP)*, Fortaleza, Brazil, August 2008. 2, 45

[42] Andreas Goerdt. A threshold for unsatisfiability. *J. Comput. Syst. Sci.*, 53(3):469–486, 1996. 139

[43] Evgueni Goldberg, Mukul Prasad, and Robert Brayton. Using SAT for combinational equivalence checking. *Design, Automation and Test in Europe Conference and Exhibition*, 2001. 45

[44] Evguenii Goldberg, Mukul Prasad, and Robert Brayton. Using SAT for combinational equivalence checking. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 114–121, Piscataway, NJ, USA, 2001. IEEE Press. 38

[45] Tarik Hadzic, Rune M. Jensen, and Henrik Reif Andersen. Calculating valid domains for bdd-based interactive configuration. *CoRR*, abs/0704.1394, 2007. 2, 23, 36

[46] Robert M. Haralick and Gordon L. Elliott. Increasing tree search efficiency for constraint satisfaction problems. *Artif. Intell.*, 14(3):263–313, 1980. 19

[47] Bernardo A. Huberman and Tad Hogg. Phase transitions in artificial intelligence systems. *Artif. Intell.*, 33(2):155–171, 1987. 139

[48] Kyo Kang, Sholom Cohen, James Hess, William Nowak, and Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, November 1990. 1, 10

[49] Kyo Kang, Jaejoon Lee, and Patrick Donohoe. Feature-oriented product line engineering. *Software, IEEE*, 19(4):58–65, Jul/Aug 2002. 111

[50] Charles W. Krueger. Biglever software gears and the 3-tiered SPL methodology. In *OOPSLA Companion*, pages 844–845, 2007. 2, 40

[51] Sean Quan Lau. Domain analysis of e-commerce systems using feature-based model templates. Master's thesis, Dept. Electrical and Computer Engineering, University of Waterloo, Canada, 2006. Available at: `http://gp.uwaterloo.ca/files/2006-lau-masc-thesis.pdf`. 111

[52] Jaejoon Lee and Dirk Muthig. Feature-oriented variability management in product line engineering. *Communications of the ACM*, 49(12):55–59, 2006. 111

[53] Thomas Leich, Sven Apel, Laura Marnitz, and Gunter Saake. Tool support for feature-oriented software development: featureide: an eclipse-based approach. In *eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 55–59, New York, NY, USA, 2005. ACM. 118

[54] Lind-Nielsen and Haim Cohen. The *BuDDy* BDD library, 2004–2008. Available at: `http://sourceforge.net/projects/buddy`. 21

[55] Alan Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1):99–118, 1977. Reprinted in *Readings in Artificial Intelligence*, B. L. Webber and N. J. Nilsson (eds.), Tioga Publ. Col., Palo Alto, CA, pp. 69-78, 1981. 19

[56] Alan K. Mackworth. On reading sketch maps. In *Proc. Fifth International Joint Conf. on Artificial Intelligence*, pages 598–606, MIT, Cambridge, MA, 1977. 19

[57] Sliarad Malik, Albert Wang, Robert Brayton, and Alberto Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Computer-Aided Design, 1988. ICCAD-88. Digest of Technical Papers., IEEE International Conference on*, pages 6–9, nov 1988. 27

[58] Mike Mannion. Using first-order logic for product line model validation. In Gary J. Chastek, editor, *Proceedings of the Second International Conference on Software Product Lines*, volume 2379 of *Lecture Notes in Computer Science*, pages 176–187, Heidelberg, Germany, 2002. Springer-Verlag. 31, 42

[59] Mario Selbig. AmiEddi, 2000-2004. Tool available at `http://www.generative-programming.org`. 2, 41

[60] J. J. McGregor. Relational consistency algorithms and their application in finding subgraph and graph isomorphisms. *Inf. Sci.*, 19(3):229–250, 1979. 19

[61] Hong Mei, Wei Zhang, and Fang Gu. A feature oriented approach to modeling and reusing requirements of software product lines. *Computer Software and Applications Conference, 2003. COMPSAC 2003. Proceedings. 27th Annual International*, pages 250–256, Nov. 2003. 111

[62] Christoph Meinel and Thorsten Theobald. *Algorithms and Data Structures in VLSI Design.* Springer-Verlag, 1998. 24, 28, 47

[63] Marcilio Mendonca. Research project web-page for 'efficient compilation techniques for large scale feature models', 2008. Available at: `http://csg.uwaterloo.ca/~marcilio/fmcompilation/index.html`. 141, 145, 146

[64] Marcilio Mendonca, Thiago Bartolomei, and Donald Cowan. Decision-making coordination in collaborative product configuration. In *23rd Annual ACM Symposium on Applied Computing.* ACM, March 2008. 8, 36, 111

[65] Marcilio Mendonca, Toacy Oliveira, and Donald Cowan. Collaborative product configuration in software product lines: formalization and dependency analysis. *Journal of Software*, 3(2):69–82, 2008. 2, 8, 36

[66] Marcilio Mendonca, Andrzej Wasowski, Krzysztof Czarnecki, and Donald D. Cowan. Efficient compilation techniques for large scale feature models. In *International Conference on Generative Programming and Component Engineering (GPCE'08)*, pages 13–22, 2008. 4, 8, 42, 47

[67] Marcilio Mendonça, Donald D. Cowan, and Toacy Cavalcante de Oliveira. A process-centric approach for coordinating product configuration decisions. In *Hawaii International Conference on System Sciences*, page 283, 2007. 9

[68] Marcilio Mendonça, Krzysztof Czarnecki, Toacy Cavalcante de Oliveira, and Donald D. Cowan. Towards a framework for collaborative and coordinated product configuration. In *OOPSLA Companion, Doctoral Symposium and Poster Session*, pages 649–650, 2006. 9

[69] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *Proceedings of the 10th National Conference on AI - American Association for Artificial Intelligence*, pages 459–465, 1992. 139

[70] R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. 2+p-SAT: Relation of typical-case complexity to the nature of the phase transition, 1999. 139

[71] Viresh Paruthi and Andreas Kuehlmann. Equivalence checking combining a structural SAT-solver, BDDs, and simulation. *Computer Design, International Conference on*, 0:459, 2000. 38

[72] Klaus Pohl. Varmod-prime tool-environment, 2003-2008. http://www.sse.uni-due.de/wms/en/index.php?go=256. 2, 41

[73] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005. 1

[74] pure-systems GmbH. Variant management with pure::consul. Technical White Paper. Available from `http://web.pure-systems.com`, 2003. 2, 40

[75] Ondrej Rohlik and Alessandro Pasetti. *XFeature Modeling Tool.* Automatic Control Laboratory, ETH Zürich, 2005. `http://www.pnp-software.com/XFeature/`. 2, 39

[76] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming.* Elsevier Science, 2006. 14, 67

[77] Richard Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD '93: Proceedings of the 1993 IEEE/ACM international conference on Computer-aided design*, pages 42–47, Los Alamitos, CA, USA, 1993. IEEE Computer Society Press. 28

[78] Barbara M. Smith. Locating the phase transition in constraint satisfaction problems. In *Artificial Intelligence*, 1994. 139

[79] Periklis Sochos, Matthias Riebisch, and Ilka Philippow. Feature-oriented development of software product lines: Mapping feature models to the architecture. In Springer, editor, *Lecture notes in computer science*, pages 138–152. Net.ObjectDays, 2004. 111

[80] Fabio Somenzi. The *CUDD* BDD library, 2005. Available at: `http://vlsi.colorado.edu/~fabio/CUDD/`. 21

[81] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Details of formalized relations in feature models using OCL. *Engineering of Computer-Based Systems, 2003. Proceedings 10th IEEE International Conference and Workshop on the*, pages 297–304, April 2003. 46

[82] Detlef Streitferdt, Matthias Riebisch, and Ilka Philippow. Details of formalized relations in feature models using OCL. *Engineering of Computer-Based Systems, 2003. Proceedings. 10th IEEE International Conference and Workshop on the*, pages 297–304, April 2003. 111

[83] Jing Sun, Hongyu Zhang, Yuan Fang, and Li Hai Wang. Formal semantics and verification for feature modeling. *Engineering of Complex Computer Systems, 2005. ICECCS 2005. Proceedings of 10th IEEE International Conference on*, pages 303–312, June 2005. 3, 46, 111

[84] Bedir Tekinerdogan and Mehmet Aksit. Managing variability in product line scoping using design space models. *Proceedings of Software Variability Management Workshop, Gronin-gen, IWI 2003-7-01, The Netherlands*, pages 5–12, 2003. 111

[85] Pablo Trinidad, David Benavides, Antonio Ruiz-Cortes ands, Sergio Segura, and Alberto Jimenez. FAMA framework. *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 359–359, Sept. 2008. 43

[86] Arie van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002. Available from `http://homepages.cwi.nl/~arie/papers/fdl/fdl.pdf`. 31, 36, 111

[87] Basil Vandegriend and Joseph Culberson. The gn,m phase transition is not hard for the hamiltonian cycle problem. *Journal of Artificial Intelligence Research*, 9:9–219, 1998. 139

[88] Thomas von der Maen and Horst Lichter. Determining the variation degree of feature models. In J. Henk Obbink and Klaus Pohl, editors, *SPLC*, volume 3714 of *Lecture Notes in Computer Science*, pages 82–88. Springer, 2005. 3, 36, 111

[89] Thomas von der Maßen and Horst Lichter. *RequiLine*. RWTH Aachen, 2005. `http://www-lufgi3.informatik.rwth-aachen.de/TOOLS/requiline/`. 2, 40

[90] Hai H. Wang, Yuan Fang Li, Jing Sun, Hongyu Zhang, and Jeff Pan. Verifying feature models using OWL. *Web Semant.*, 5(2):117–129, 2007. 46

[91] John Whaley. The *JavaBDD* BDD library, 2003–2007. Available at: `http://javabdd.sourceforge.net/`. 21, 110

[92] Jules White, Douglas Schmidt, David Benavides, Pablo Trinidad, and Antonio RuizCortes. Automated diagnosis of product-line configuration errors in feature models. *Software Product Line Conference, 2008. SPLC '08. 12th International*, pages 225–234, Sept. 2008. 2, 3, 36, 44

[93] Ke Xu and Wei Li. Exact phase transitions in random constraint satisfaction problems. *CoRR*, cs.AI/0004005, 2000. 139

[94] Wei Zhang, Haiyan Zhao, and Hong Mei. A propositional logic-based method for verification of feature models. In Jim Davies, Wolfram Schulte, and Mike Barnett, editors, *Formal Methods and Software Engineering: 6th International Conference on Formal Engineering Methods, ICFEM 2004, Seattle, WA, USA, November 8-12, 2004. Proceedings*, volume 3308 of *Lecture Notes in Computer Science*, pages 115–130, Heidelberg, Germany, 2004. Springer-Verlag. 31

Appendix A

Auxiliary Algorithms

Algorithm 23 Propagates a *true* assignment UP in the feature tree

Inputs:
f: propagation starting point; f is a feature assigned *true*

Function **FT-prop-trueAS**(f:feature)

```
1: if ($f \neq nil$) then
2:    $parent \leftarrow$ parent($f$)
3:    if ($parent \neq nil$ and $parent$ is NOT the root node and $parent$ is uninstantiated) then
4:       FT-assign($parent$,$true$)
5:       if ($f$ is NOT a grouped feature) then
6:          FT-prop-trueDS($parent$, $f$)
7:          if ($parent$ is a grouped feature ) then
8:             FT-prop-trueGS($parent$)
9:          end if
10:      end if
11:      FT-prop-trueAS($parent$) {recursive call}
12:   end if
13: end if
```

Algorithm 24 Propagates a *true* assignment DOWN in the feature tree

Inputs:
f: propagation starting point; f is a feature assigned *true*
v: child of f that will be excluded from propagation

Function **FT-prop-trueDS**(f:feature, v:boolean)

```
1: if ($f \neq nil$) then
2:   if ($f$ is a feature group) then
3:     if (the sum of true and uninstantiated features in the group equals the
       group lower bound) then
4:       for (each uninstantiated feature $G$) do
5:         if (FT-get-value($G$)$\neq v$) then
6:           FT-assign($G$, true)
7:           FT-prop-trueDS($G$,$v$)
8:         end if
9:       end for
10:    end if
11:  else
12:    for (each child feature $C$ of $f$) do
13:      if (FT-get-value($C$) $\neq v$) then
14:        if ($C$ is a feature group w/ lower bound $> 1$ or a mandatory feature)
           then
15:          FT-assign($C$, true)
16:          FT-prop-trueDS($C$,$v$)
17:        end if
18:      end if
19:    end for
20:  end if
21: end if
```

Algorithm 25 Propagates a *true* assignment within a feature group

Inputs:
f: propagation starting point; f is a grouped feature assigned $true$

Function **FT-prop-trueGS**(f:feature)

```
1: if ($f \neq nil$ and $f$ is a grouped feature) then
2:    if (number of grouped features assigned $true$ is equal to group upper bound)
      then
3:       for (each uninstantiated grouped feature $G$ in the group) do
4:          FT-assign($G$, $false$)
5:          FT-prop-falseDS($G$, $nil$)
6:       end for
7:    else if (the sum of $true$ and $uninstantiated$ grouped features is equal to the
      group lower bound) then
8:       for (each uninstantiated grouped feature $G$ in the group) do
9:          FT-assign($G$, $true$)
10:         FT-prop-trueDS($G$, $nil$)
11:      end for
12:   end if
13: end if
```

Algorithm 26 Propagates a $false$ assignment UP in the feature tree

Inputs:
f: propagation starting point; f is a grouped feature assigned FALSE

Function **FT-prop-falseAS**(f:feature)

```
 1: if ($f \neq nil$) then
 2:    $parent$ ← parent($f$)
 3:    if ($parent \neq nil$ and $parent$ is NOT the root node) then
 4:       if ($f$ is mandatory feature or a feature group with lower bound > 0) then
 5:          FT-assign($parent$,$false$)
 6:          FT-prop-falseDS($parent$, $f$)
 7:          if ($parent$ is a grouped feature) then
 8:             FT-prop-falseGS($parent$)
 9:          end if
10:          FT-prop-falseAS($parent$) {recursive call}
11:       end if
12:    else if ($f$ is a grouped feature) then
13:       if (number of grouped features assigned $false$ > (group size - lower bound))
          then
14:          FT-assign($parent$, $false$)
15:          FT-prop-falseAS($parent$)
16:       end if
17:    end if
18: end if
```

Algorithm 27 Propagates a $false$ assignment DOWN in the feature tree

Inputs:
f: propagation starting point; f is a feature assigned $false$
v: child of f that will be excluded from propagation

Function **FT-prop-falseDS**(f:feature, v:boolean)

```
1: if ($f \neq nil$) then
2:    for (each child feature $C$ of $f$) do
3:       if (FT-get-value($C \neq v$ and $C$ is uninstantiated) then
4:          FT-assign($C$, $false$)
5:          FT-prop-falseDS($C$,$v$)
6:       end if
7:    end for
8: end if
```

Algorithm 28 Propagates a $false$ assignment within a feature group

Inputs:
f: propagation starting point; f is a grouped feature assigned FALSE

Function **FT-prop-falseGS**(f:feature)

```
1: if ($f \neq nil$ and $f$ is a grouped feature) then
2:    if (number of true and uninstantiated features is equal to group lower bound)
      and (parent($f$) is $true$) then
3:       for (each uninstantiated grouped feature $G$ in the group) do
4:          FT-assign($G$, $true$)
5:          FT-prop-falseDS($G$, $nil$)
6:       end for
7:    else if (number of FALSE-assigned features is greater than group upper
      bound) then
8:       for (each uninstantiated grouped feature $G$ in the group) do
9:          FT-assign($G$, $false$)
10:         FT-prop-trueDS($G$, $nil$)
11:      end for
12:   end if
13: end if
```

Algorithm 29 Prepares the child iterators of an iterator so that the next solution can be found. Returns *true* if a solution can still be found or *false* otherwise.

Inputs:
child-iterator: the list of child iterator objects

Function **FT-prepare-child-iterators**(*child-iterators*:sol-iterator{})
: Boolean

```
1: child-iterator = get-last-iterator(child-iterators)
2: while (child-iterator <> NIL AND NOT has-next-sol(child-iterator)) do
3:     reset-iterator(child-iterator)
4:     child-iterator = get-previous-iterator(child-iterators)
5: end while
6: return (child-iterator <> NIL)
```

Algorithm 30 Get the next solution out of a list of child iterator objects

Inputs:
child-iterator: the list of child iterator objects

Function **FT-get-child-iterators-sol**(*child-iterators*:sol-iterator{})
: feature {}

```
1: sol = {}
2: child-iterator = get-first-iterator(child-iterators)
3: while (child-iterator <> get-last-iterator(child-iterators)) do
4:     sol = sol ∪ FT-current-sol(child-iterator)
5: end while
6: sol = sol ∪ FT-next-sol(child-iterator)
7: return sol
```

Algorithm 31 Create an initial set of clusters for a given feature F. Each of F's children will form a cluster with a single relation $\{F\}$

Inputs:
F: a feature for which initial clusters will be created

Function **create-initial-clusters-set**(F:feature) : cluster{}

```
1: $CS$ = new empty clusters set
2: for (each $C$ child of $F$) do
3:     $CL$ = new empty cluster
4:     add node $C$ to cluster $CL$
5:     add cluster $CL$ to clusters set $CS$
6: end for
7: attach clusters set $CS$ to feature $F$
8: return $CS$
```

Algorithm 32 Merge clusters in clusters set CS that share elements in R. A cluster NC representing the merge is returned.

Inputs:
CS: clusters set
R: features set used to merge clusters in CS

Function **merge-clusters-sharing-elements**(CS: cluster{}, R: features{}): cluster

```
 1: $NC$ = new empty cluster
 2: for (each $CL$ cluster of $CS$) do
 3:     if (any node in $R$ appears in $CL$) then
 4:         copy nodes of cluster $CL$ to cluster $NC$
 5:         copy relations of cluster $CL$ to cluster $NC$
 6:         remove cluster $CL$ from clusters set $CS$
 7:     end if
 8: end for
 9: add cluster $NC$ to clusters set $CS$
10: return $NC$
```