TU Braunschweig

Master's Thesis

Presence Condition Reasoning with Feature Model Interfaces

Frederik Kanning

December 7, 2016

Prof. Dr.-Ing. Ina Schaefer
Institut für Softwaretechnik und Fahrzeuginformatik

Advisors:

Dr.-Ing. Thomas Thüm
Stephan Mennicke

Abstract

Family-based static analysis techniques allow to efficiently analyze the exponential variant space of a *software product line (SPL)*. Rather than analyzing each variant completely on its own, a family-based analysis processes information shared among multiple variants only once. The analysis delegates parts of the exponential complexity to reasoning about boolean *presence conditions*, which control the inclusion or exclusion of code fragments in a particular variant. The *feature model* of the SPL, which determines the set of valid variants, must be included in the reasoning process to obtain correct results. However, typically not all parts of a feature model are relevant for a particular condition. In this work, we propose a method to accelerate presence condition reasoning by decreasing the size of the model used for reasoning. In particular, we use the concept of *feature model interfaces* to decompose a *feature diagram* according to its hierarchical structure, and obtain an abstraction that can be selectively refined on the fly for a given condition. We formalize our approach in terms of propositional feature diagram semantics and prove its correctness. In our evaluation, we demonstrate that our approach accelerates reasoning by up to 24 percent.

Contents

List of figures

List of tables

List of code listings

1. Introduction

A *software product line* (SPL) allows to compose custom-tailored software *variants* from reusable artifacts [Clements and Northrop, 2001; Czarnecki and Eisenecker, 2000]. A product line is modeled by a set of *features*, which represent individual units of functionality that may be present or absent in concrete variants. Typically, a product line provides a configuration mechanism that receives a set of desired features, and produces the variant that behaves accordingly. To this end, developers associate code artifacts with particular features or feature combinations. Such mechanisms range from compile-time generators (e.g., the C preprocessor [Hunsen et al., 2015]) to runtime conditionals (e.g., `if` statements [Apel et al., 2013a]). The ability to produce different products from a common code base is called *variability*.

In general, not all configurations of a product line are valid. For example, two features may be mutually exclusive due to technical restrictions. Therefore, a product line has a *feature model* that defines the set of valid feature combinations [Kang et al., 1990]. The configuration mechanism enforces such restrictions imposed by the model, so that only valid variants can be derived. In particular, *feature diagrams* provide a graphical representation of feature models, and are for example used to communicate among stakeholders. Feature diagrams organize features hierarchically in form of a tree, such that subordinate features concretize the specification of their parents.

Static analysis of product lines, such as parsing [Kästner et al., 2011] or type checking [Kästner et al., 2012], requires enhanced techniques due to the large variant space [Thüm et al., 2014a]. Assuming boolean configuration options, a product line with n features gives rise to 2^n variants in the worst case. Analyzing each variant individually (called *brute-force approach*) is therefore not feasible. Consequently, *family-based analyses* have been developed, which process a product line as a whole, and do not examine individual variants. Often, variants share some portions of the code. By processing such commonalities only once, family-based analyses avoid redundant computations and hence are more efficient than brute-force analyses. For example, a 6,000 feature version of the Linux kernel has been parsed in 85 hours using a family-based technique [Kästner et al., 2011], whereas parsing 2^{6000} individual variants in reasonable time would be clearly impossible.

Family-based analyses process program code along with the variability mechanism of the product line (e.g., preprocessor `#ifdefs`). The variability mechanism associates code fragments with features or combinations thereof, typically through propositional formulas called *presence conditions* that range over the available features. Reasoning about such conditions is a central concern of family-based analyses, as information from different code fragments can be combined only if those are present together in some variant. Consequently, to produce correct results, the analysis needs to respect the feature model while reasoning. To this end, feature models are translated to propositional formulas [Batory, 2005; Benavides et al., 2010], such that presence conditions can be checked against them.

Satisfiability solvers (SAT solvers) can be used to reason about propositional formulas [Harrison, 2009]. Although deciding satisfiability is NP-complete and thus presumably has exponential worst-case complexity in the number of formula atoms, solvers operate in acceptable times on problems that are related to feature models [Mendonca et al., 2009; Liang et al., 2015], and static analysis of product line code [Liebig et al., 2013]. Nevertheless, Liebig et al. [2013] have reported that solving a typical condition during an analysis of the Linux kernel still took 0.5 seconds on average when the feature model was involved. To accelerate some of their analyses, they removed the feature model from SAT checks on intermediate results, and instead only used it when final results were returned. In the evaluation of this work, we found that reasoning takes about 12 seconds in total when type checking a typical Linux source file.

In this thesis, we present an approach to accelerate family-based analyses by reducing reasoning times. Our goal is to decrease the size of the feature model that is used by the solver. To this end, we compose a specific abstract model on the fly for each presence condition from parts of the original model. This model contains fewer features than the original model and therefore facilitates faster reasoning. Nevertheless, we ensure by construction that the composed model is sound and complete with respect to the particular presence condition.

To construct such abstractions of feature models, we use the concept of *feature model interfaces* [Schröter et al., 2016]. An interface of a feature model has fewer features, yet preserves all dependencies between the remaining features. Feature model interfaces can therefore be used as a substitute for the original model if one is interested in analyzing properties (e.g., *dead features* [Benavides et al., 2010]) for only a subset of features. Furthermore, composing interfaces of parts of a feature model yields an interface of the overall model under certain restrictions.

We follow the general approach of Schröter et al. [2016] in this thesis to decompose and reduce a feature model. In a nutshell, we select subtrees of a given feature diagram as elements of our decomposition, the intuition being that these subtrees correspond to cohesive implementation units (e.g., modules) of the product line. In particular, we hypothesize that presence conditions which arise during the analysis of such an impementation unit mostly contain features that belong to such a single subtree. For example, we would expect that networking code does not involve checking feature model constraints of the filesystem code and vice versa. Using feature model slicing [Acher et al., 2011], we then obtain an interface of each subtree, which only contains features that are required for re-composition with the other subtrees (e.g., features in cross-tree constraints).

The composition of those interfaces is an interface of the original model, which can be selectively refined to a more concrete interface by re-composing it with individual concrete subtrees. We show that selecting all decomposition subtrees which share a feature with a given presence condition yields a sound and complete model for this condition. In other words, feature-based selection on such a feature diagram decomposition is sufficient to ensure correct reasoning. Since we aim to choose a decomposition that corresponds to the implementation structure, we expect that a selection returns few (or even a single) model for most presence conditions, and consequently the speed gain in the solver because of the smaller model outweighs the recomposition costs.

Schröter et al. [2016] define feature model interfaces on sets of product line configurations. By contrast, we base our theory directly on propositional logic, since we are mainly concerned with reasoning about propositional formulas. Furthermore, we formalize a generic feature diagram decomposition, which only requires a basic tree structure, whereas all other feature dependencies (e.g., group constraints) are considered as arbitrary formulas. Subsequently, we prove the correctness of our approach.

To evaluate the potential of our approach for accelerating family-based analyses, we analyze characteristics of presence conditions that arise while type checking a version of the Linux kernel. In particular, we measure reasoning times, and other properties such as the number of distinct features. To compare the performance of our approach with the conventional technique that uses the full feature model, we implement both strategies in a prototype based on FeatureIDE [Thüm et al., 2014b]. We use the data obtained from the type checking experiment to generate a set of realistic presence conditions, which we process with both strategies while measuring computation times.

Contribution

In particular, we make the following contributions.

- We formalize a simple approximate but complete strategy to select relevant parts from a set of propositional formulas, in order to reason about another given formula, which is based on the set of shared atomic propositions.

- We transfer the notion of feature model interfaces to propositional logic, thereby defining the concept of a *propositional interface* as a means for sound and complete reasoning for a subset of atomic propositions.

- We formalize feature diagrams in a generic way that only requires a tree structure, and provide semantics in propositional logic for this definition.

- We define a decomposition and abstraction operation on feature diagrams that is based on subtrees, and that yields a set of formulas that can be selectively composed for presence condition reasoning.
- We prove the correctness of our approach. That is, our decomposition satisfies the requirements of the atom-based selection strategy we have defined.
- We analyze properties of presence condition reasoning in practical family-based analysis.
- Finally, we compare the performance of our approach with a conventional reasoning technique, using a prototypical implementation.

Structure of the thesis

This thesis is organized as follows. In Chapter 2, we provide the necessary background for this thesis, particularly propositional logic, software product lines, feature models and feature model interfaces. In Chapter 3, we present our approach to reason with decomposed feature diagrams and prove its correctness. Subsequently, we analyze properties of practical presence conditions, and compare effectiveness of our approach with conventional reasoning in Chapter 4. Chapter 5 surveys related work. Finally, we summarize the results of this thesis in Chapter 6.

2. Background

In this chapter, we provide the necessary background for this thesis. In Section 2.1, we formally define syntax and semantics of standard propositional logic as a means to reason about feature models. In Section 2.2, we explain relevant aspects of software product lines. In particular, we describe *feature models* and *feature model interfaces*. Using the concept of *annotation-based product lines*, we then illustrate how feature models and implementation artifacts are connected. Finally, we explain how such product lines can be statically analyzed using *family-based analysis*.

2.1 Propositional logic

Propositional logic [Huth and Ryan, 2004] allows to formalize statements about concepts that are either true or false. These concepts are called *atomic propositions* or short *atoms*. For example, we could define the statements "it rains" (R) and "sun shines" (S) as atoms. Depending on the weather, these statements are either true or false. Unless stated otherwise, we adapt notions from Huth and Ryan [2004] in this section.

Formulas in propositional logic consist of atoms connected by logical operators, which are called *connectives*. For example, the $\land$ connective represents the logical "and", whereas $\neg$ represents "not". We could therefore formalize the statement "it rains and the sun does not shine" as the propositional formula $R \land \neg S$.

A *valuation* assigns a truth value (i.e., *true* or *false*) to each atom of a formula. As a consequence, also each formula has a truth value, which depends on the truth values of the atoms, and the meaning of the particular connectives. For example, the above statement $R \land \neg S$ is only true, if it indeed rains (i.e., R is true), and the sun does not shine (i.e., S is false).

Furthermore, a propositional formula may have other statements as a consequence. This relation is called *entailment*. If we assume that $R \land \neg S$ is true, then consequently R must be true. Therefore, $R \land \neg S$ *entails* R. There are also formulas that have the same semantic meaning, but different syntactic structure. For example, the formulas

$R \wedge \neg S$ and $\neg S \wedge R$ obviously represent the same statement about the weather, but in different order. Such formulas are called *equivalent*. We are able to computationally prove or disprove such entailments and equivalences using *satisfiability solvers* (SAT solvers).

We now precisely define the notions that we have informally explained above. First, we formally define the syntax of propositional formulas. Second, we give meaning to the syntactic elements by defining a formal semantics for them. Finally, we define entailment and equivalence, and relate them to satisfiability solving.

2.1.1 Syntax

We define the syntax of propositional formulas inductively over the atomic propositions (atoms) and connectives as follows. Additionally, we define special symbols for contradiction ($\bot$), which is always false, and tautology ($\top$), which is always true.

Definition 2.1 (Propositional formula)**.** Let A be a set of atoms. The symbols $\bot$ and $\top$, and every atom $t \in A$ are propositional formulas. Furthermore, if φ and ψ are propositional formulas, then also

- $\neg\varphi$,
- $\varphi \wedge \psi$,
- $\varphi \vee \psi$, and
- $\varphi \Rightarrow \psi$

are propositional formulas. The set of atoms of a formula φ is denoted as $a(\varphi)$. Moreover, the set of formulas over atoms A is denoted as $\Theta(A)$.

We assume the usual precedence rules for the connectives. That is, $\neg$ binds stronger than $\wedge$, $\vee$ and $\Rightarrow$. Moreover, $\wedge$ and $\vee$ bind stronger than $\Rightarrow$.

Example 2.1. The term $R \wedge \neg S$ is a propositional formula, and its set of atoms $a(R \wedge \neg S)$ is $\{R, S\}$. Also, the formula is in the set $\Theta(\{P, Q, R, S\})$, but not in $\Theta(\{R\})$. By the precedence rules, the formula $\neg P \wedge Q \Rightarrow R$ actually represents the formula $((\neg P) \wedge Q) \Rightarrow R$.

2.1.2 Semantics

To define the truth value of a formula, we specify how to evaluate each atom and each connective. The truth value of a formula depends on the concrete choice of truth values for each atomic proposition. Since atomic propositions are either true or false, we define a function α from atoms into a binary set $\mathbb{B} = \{0, 1\}$, where 0 represents false and 1 represents true.

Definition 2.2 (Valuation)**.** Let φ be a propositional formula. A *valuation* for φ is a function $\alpha : A \rightarrow \mathbb{B}$, where A is a set of atoms such that $a(\varphi) \subseteq A$.

Note that some authors define valuations for precisely those atoms that occur in the formula (i.e., $A = a(\varphi)$). This would be unnecessarily restrictive for our purposes, since we intend to use a single valuation on multiple formulas, which may have different sets of atoms. Therefore, we merely require A to be a superset of $a(\varphi)$. For example, a valuation for the formula $P \wedge Q$ is thus also a valuation for $\neg P$.

So far, we have only defined how to evaluate atoms. Now, we define a function that also evaluates connectives [Harrison, 2009]. To this end, we extend each valuation α defined for atoms A to a function $\widehat{\alpha}$ on all formulas over A (i.e., the set $\Theta(A)$). For atoms, the value of $\widehat{\alpha}$ is simply α, whereas for connectives, we define the value either directly, or express it using other connectives[1].

Definition 2.3 (Truth value)**.** Let $\alpha : A \to \mathbb{B}$ be a valuation. The function

$$\widehat{\alpha} : \Theta(A) \to \mathbb{B}$$

determines a *truth value* for every formula in $\Theta(A)$, and is defined recursively as follows:

$$\begin{aligned}
\widehat{\alpha}(\bot) &= 0 \\
\widehat{\alpha}(t) &= \alpha(t) \text{ for all } t \in A \\
\widehat{\alpha}(\neg\varphi) &= \begin{cases} 1 & \text{if } \widehat{\alpha}(\varphi) = 0 \\ 0 & \text{otherwise} \end{cases} \\
\widehat{\alpha}(\top) &= \widehat{\alpha}(\neg\bot) \\
\widehat{\alpha}(\varphi \wedge \psi) &= \begin{cases} 1 & \text{if } \widehat{\alpha}(\varphi) = 1 \text{ and } \widehat{\alpha}(\psi) = 1 \\ 0 & \text{otherwise} \end{cases} \\
\widehat{\alpha}(\varphi \vee \psi) &= \widehat{\alpha}(\neg(\neg\varphi \wedge \neg\psi)) \\
\widehat{\alpha}(\varphi \Rightarrow \psi) &= \widehat{\alpha}(\neg\varphi \vee \psi)
\end{aligned}$$

Example 2.2. Consider the formula $\varphi = P \Rightarrow Q \wedge R$. Now let α be a valuation for φ such that $\alpha(P) = 1$ and $\alpha(Q) = 1$ and $\alpha(R) = 0$. Then, we can use $\widehat{\alpha}$ to determine the truth value of φ with respect to valuation α as follows. First, we evaluate the outer formula:

$$\begin{aligned}
\widehat{\alpha}(P \Rightarrow Q \wedge R) &= \widehat{\alpha}(\neg P \vee (Q \wedge R)) \\
&= \widehat{\alpha}(\neg(\neg\neg P \wedge \neg(Q \wedge R)))
\end{aligned}$$

Next, we evaluate the inner term $\neg\neg P \wedge \neg(Q \wedge R)$ in bottom-up order. Since $\widehat{\alpha}(P) = \alpha(P) = 1$, we have $\widehat{\alpha}(\neg P) = 0$ and consequently $\widehat{\alpha}(\neg\neg P) = 1$. Moreover, $\widehat{\alpha}(Q \wedge R) = 0$, because $\alpha(R) = 0$. Therefore, $\widehat{\alpha}(\neg(Q \wedge R)) = 1$. Thus, $\widehat{\alpha}(\neg(Q \wedge R)) = 1$. Consequently, it follows that $\widehat{\alpha}(\neg\neg P \wedge \neg(Q \wedge R)) = 1$. Since we actually needed to evaluate the negation of the previous term, we finally obtain that $\widehat{\alpha}(P \Rightarrow (Q \wedge R)) = 0$.

2.1.3 Satisfiability and entailment

We are now able to evaluate a propositional formula to a value. This thesis is concerned with deciding entailments, which can be reduced to solving a *satisfiability problem*, that is, whether a formula can be evaluated to true at all. Thus, we define satisfiability and entailment next, and link these notions through a theorem.

[1]Since the set $\{\neg, \wedge\}$ of connectives is *adequate*, we can use these two connectives to express all other connectives [Harrison, 2009].

Definition 2.4 (Satisfiability)**.** A valuation α *satisfies* a propositional formula φ if $\widehat{\alpha}(\varphi) = 1$. Consequently, a formula is *satisfiable* if there exists a valuation that satisfies it.

Example 2.3. The formula $P \lor Q$ is satisfied by a valuation α such that $\alpha(P) = 1$ and $\alpha(Q) = 0$. By contrast, the formula $P \land \neg P$ is not satisfiable by any valuation.

The details of SAT solving algorithms are beyond the scope of this thesis. Therefore, we treat SAT solvers as black box tools that decide the satisfiability of arbitrary propositional formulas. In particular, we use the *Sat4j* solver [Berre and Parrain, 2010] for our implementation. It is generally assumed that deciding the satisfiability of a formula has a worst-case complexity exponential in the number of its atoms [Harrison, 2009]. However, such SAT solvers are typically faster for formulas that arise in real-world applications, especially in the context of software product lines [Liang et al., 2015].

Sat4j expects input formulas to be in *conjunctive normal form* (*CNF*). The conjunctive normal form of a formula is an equivalent formula that has a particular structure. A CNF is a conjunction of so-called *clauses*. A clause is a disjunction of *literals*, where a literal is defined as either an atomic proposition or its negation. For example, the formula $(P \lor Q) \land (P \lor \neg R)$ is a CNF of $P \lor (Q \land \neg R)$.

Now that we have described satisfiability, we explain entailment next. In propositional logic, we can derive formulas from other formulas. Particularly, if a formula φ is true for some valuation, there are formulas that must also be true for the same valuation. In other words, these formulas are a semantic consequence of φ. For example, regarding software product lines, we may be interested whether a feature model entails a formula that expresses a certain dependency between features. We formalize this property in the following definition. However, it is customary to refer to a set of formulas Γ instead of a single formula φ.

Definition 2.5 (Semantic entailment)**.** Let Γ be a set of propositional formulas, and let φ be a propositional formula. The set Γ *entails* φ, denoted as $\Gamma \models \varphi$, if every valuation that satisfies all formulas in Γ also satisfies φ.

Example 2.4. Let $\Gamma = \{P \Rightarrow Q, P\}$. Then $\Gamma \models Q$, but $\Gamma \not\models R$. Note that contradictions (e.g., $\bot$) entail arbitrary formulas, since they have no satisfying valuation.

For convenience, we use sets of formulas and the conjunction over their elements interchangeably, depending on the context[2]. For example, let $\Gamma = \{P, Q\}$. Then we interpret $\Gamma \land R$ as the formula $P \land Q \land R$, or as the set $\{P, Q, R\}$, as needed. Consequently, we also write $\varphi \models \ldots$ instead of $\{\varphi\} \models \ldots$ for a single formula φ.

If two formulas entail each other, they are interchangeable. More precisely, both are satisfied by the same sets of valuations. Therefore, we can define an equivalence relation $\equiv$ on propositional formulas that goes beyond syntactic equality and that precisely represents semantic equivalence.

[2]This is valid, since we only deal with finite sets of formulas in this work. We can transform a finite Γ into a conjunction and vice versa using the conjunction introduction and elemination rules from natural deduction [Huth and Ryan, 2004].

Definition 2.6 (Semantic equivalence)**.** Two propositional formulas φ_1 and φ_2 are *equivalent*, denoted as $\varphi_1 \equiv \varphi_2$, if $\varphi_1 \vDash \varphi_2$ and $\varphi_2 \vDash \varphi_1$.

Finally, we show that we can use satisfiability to decide entailment. Intuitively, if $\varphi \vDash \psi$, then by definition there cannot exist a valuation that satisfies φ, but does not satisfy ψ. Consequently, the formula $\varphi \land \neg\psi$ cannot be satisfiable. Note again that φ may be represented as a finite set of formulas.

Theorem 2.1. *Let φ and ψ be propositional formulas. Then $\varphi \vDash \psi$ if and only if the formula $\varphi \land \neg\psi$ is not satisfiable.*

Proof. From Huth and Ryan [2004], combine Proposition 1.45 with Lemma 1.41. □

2.2 Software product lines

A software system that allows to derive a family of related programs from a common code base is called a *software product line* [Clements and Northrop, 2001; Apel et al., 2013a]. This kind of engineering reduces development costs for software that is targeted at *mass customization*. Instead of manually developing multiple similar systems for overlapping sets of requirements, reusable artifacts for each unit of functionality are implemented, and then automatically integrated for a concrete set of requirements, to form a *variant* or *product*. This ability to produce different variants from a single code base is called *variability*, and is a key concept of software product lines.

To create a tailor-made variant, developers use a configuration mechanism to adjust functional and non-functional properties, ideally without editing any code. This configuration then fully determines the particular variant. Depending on the mechanism, derivation of the variant happens at compile, load, or run time. The configuration options are commonly called *features*, since they often represent high-level functionality with direct meaning to users or other stakeholders.

In the remainder of this section, we illustrate product line methodology using a simplified version of the *graph product line* (*GPL*) [Lopez-Herrejon and Batory, 2001], which is a standard example in product line research. The GPL models a family of graph algorithm libraries. For example, the GPL has a feature for weighted edges, and another feature for the computation of shortest paths in a graph.

2.2.1 Feature models

The *feature model* of a product line describes the set of available features and their dependencies. Due to technical reasons, often not all feature combinations are valid. For example, the GPL feature model demands the all products with the shortest path algorithm also must have graphs with weighted edges. In other words, there cannot be a product with the former feature, but not the latter one.

At the bare minimum, a feature model determines the set of valid product configurations (i.e., combinations of features). The following definition, which is also called *configuration semantics* [Acher et al., 2011], makes this explicit.

Definition 2.7. A *feature model* is a pair (F, P), where F is the set of features, and $P \subseteq 2^F$ is the set of valid configurations.

In this definition, each element $p \in P$ corresponds to a configuration. Thus, if some feature $t \in F$ is in p, then it is regarded as selected for that particular configuration.

Since P contains up to $2^{|F|}$ elements, there are several notations to specify feature models in a more compact form. For example, the Linux kernel uses a textual modeling language called *Kconfig*[3] to describe selectable kernel functions (e.g., driver modules) and their dependencies [Berger et al., 2010]. This language is also integrated with the build system and configuration tools of Linux. By contrast, *feature diagrams*, first proposed by Kang et al. [1990], provide a graphical notation for feature models, which is only concerned with features themselves and their relationships.

Feature diagrams

Feature diagrams arrange features hierarchically as a tree, such that child features concretize the concepts of their parent features. To express dependencies between features, feature diagrams have notations for groups of alternative features, mandatory features, cross-tree dependencies, or even arbitrary boolean constraints. To date, several types of feature diagrams have been proposed, which have different notational capabilities and expressiveness [Schobbens et al., 2007].

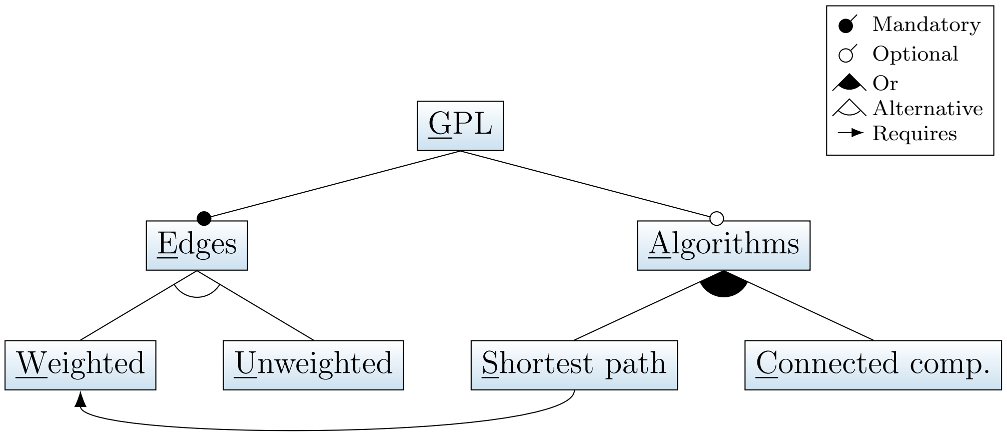


Figure 2.1: Feature diagram of the graph product line

We show the GPL feature diagram in Figure 2.1 and explain its notational elements. At the root of the tree, there is the GPL feature, which simply represents the whole product line. The root feature is usually implicitly mandatory (i.e., it is present in every product). The GPL supports different types of graph edges, and two graph algorithms. Since graphs always have edges, the Edges feature is mandatory. As graphs cannot be weighted and unweighted at the same time, the corresponding features are mutually exclusive. By contrast, multiple algorithms can be included in the product. Hence, their features are defined as an or-group, which means that we

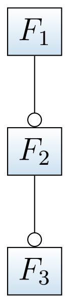
[3]Note however, that Kconfig models can only be approximated by configuration semantics, since they also permit numeric values for features.

have to select at least one of them, if the parent (Algorithms) is selected. Moreover, if we want to compute shortest paths on a graph, we need a weight value for each edge to measure distances. Therefore, the diagram contains a cross-tree constraint between Shortest path and Weighted. Since we could also use only the graph data structures without the algorithms, the Algorithms feature is optional. As a general rule, we must not select a child feature without selecting its parent. Thus, for example we cannot select Shortest path without selecting Algorithms, which also would be intuitively wrong.

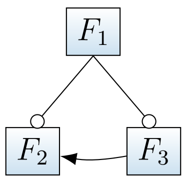
$$\begin{aligned}
F =&\{G, E, W, U, A, S, C\}\\
P =&\{\{G, E, W\},\\
&\{G, E, U\},\\
&\{G, E, W, A, S\},\\
&\{G, E, U, A, C\},\\
&\{G, E, W, A, C\},\\
&\{G, E, W, A, C\},\\
&\{G, E, W, A, S, C\}\}
\end{aligned}$$

Figure 2.2: Configuration semantics (F, P) of the GPL feature diagram

In Figure 2.2, we express the valid feature combinations of the GPL diagram as configuration semantics. We abbreviate the feature names with the corresponding underlined letters in the diagram.



(a) Parent-child relationship



(b) Cross-tree dependency

Figure 2.3: Feature diagrams with different structure, but same feature combinations

In general, feature diagrams are not unique [Czarnecki and Eisenecker, 2000]. Often there are several different diagrams that represent the same set of feature combinations. In Figure 2.3 we show an example of two such diagrams, which are equivalent with respect to the configurable products. Diagram (a) has a parent-child relationship between features F_2 and F_3. Hence, informally F_3 is some sub-functionality of F_2, and selecting F_3 implies that we also select F_2. In Diagram (b), we have changed the hierarchy such that F_2 and F_3 are now on the same level in the tree. Regarding the informal meaning, F_3 is therefore no longer a concretization or specialization of F_2. However, because of the newly introduced cross-tree dependency, we still cannot select F_3 without F_2.

Reasoning with feature diagrams

Feature models can be analyzed with regard to several properties [Benavides et al., 2010]. For example, it could be interesting to decide whether some feature model permits any configuration at all, which is called the *void feature model analysis*. As another example, we could want to extract non-obvious dependencies between features: Regarding our GPL model, we might want to know whether selecting Algorithms but not Connected comp. requires us to deselect Unweighted (which is indeed the case). While the former analysis is concerned with the model itself, the result of the latter analysis could be required by some other analysis (e.g., static code analysis).

To this end, many types of feature models can be translated to a propositional logic formula. That way, we can give precise semantics to feature models, and moreover use off-the-shelf tools like SAT solvers for analysis. Such a translation works for all diagram types that have only boolean features (i.e., features are either selected or not) [Batory, 2005; Benavides et al., 2010]. In other words, they must be expressible in the configuration semantics from Definition 2.7. However, there are for example feature models with numerical attributes and numerical constraints [Passos et al., 2011], which cannot be represented this way. Those diagrams are beyond the scope of this work.

Translation to propositional logic is straightforward. Since features are either selected or deselected, and feature models determine the valid combinations of such binary parameters, a feature directly corresponds to an atomic proposition. For each diagram element, we can then construct a propositional formula over the involved features. For example, or-groups are translated to disjunction, dependencies (child-parent, cross-tree) correspond to implication. The overall formula for the diagram is then the conjunction of all those element formulas. In Figure 2.4, we show the propositional representation of the GPL feature diagram.

Having a propositional logic representation of a feature diagram, we can use a SAT solver to compute results for the aforementioned analyses. The void analysis is simply equivalent to the unsatisfiability of the model formula, whereas the analysis of feature dependencies can be expressed as entailments. For the example above, we would decide the entailment

$$FM \models (Algor. \land \neg Conn.) \Rightarrow \neg Unweigh.,$$

where FM is the GPL model formula. As explained in Section 2.1.3, this can be achieved using a SAT solver as well.

Note that we describe the translation only informally in this section. We give a generic formal definition of feature diagrams and a corresponding translation in Chapter 3.

2.2.2 Feature model interfaces

As feature models may contain over 10,000 features in practice, and often not all features are of interest for a particular analysis [Passos et al., 2011], *feature model*

Parent-child dependencies

$$\begin{aligned}
Edges &\Rightarrow GPL\\
Algorithms &\Rightarrow GPL\\
Weighted &\Rightarrow Edges\\
Unweighted &\Rightarrow Edges\\
Shortest\ path &\Rightarrow Algorithms\\
Connected\ components &\Rightarrow Algorithms
\end{aligned}$$

Constraints

$$\begin{aligned}
GPL &\quad \text{(Root)}\\
GPL \Rightarrow Edges &\quad \text{(Mandatory feature)}\\
Edges \Rightarrow (Weigh. \lor Unweigh.) \land \neg(Weigh. \land Unweigh.) &\quad \text{(Alternative group)}\\
Algorithms \Rightarrow Shortest\ path \lor Connected\ components &\quad \text{(Or-group)}
\end{aligned}$$

Figure 2.4: Propositional representation of the GPL feature diagram

interfaces [Schröter et al., 2016] have been proposed to facilitate *compositional* analysis. A feature model interface abstracts away some features from a feature model while preserving the valid combinations for all remaining features. For several analyses such as *core features* and *atomic sets* [Benavides et al., 2010], we may therefore analyze an interface containing all features under consideration instead of analyzing the full model and then filtering the result for those features [Schröter et al., 2016]. Computing such an interface and its subsequent analysis are potentially faster than analyzing the full model, if one intends to analyze only a subset of features of the full model.

In terms of configuration semantics, an interface is a feature model that omits features which are not of interest from each configuration. In other words, an interface (F', P') of a feature model (F, P) is determined by its set of features F', such that all other features (i.e., $F \setminus F'$) are removed with set intersection. Since the removed features are not in F', their absence in all configurations does not mean deselection. Instead, their status is merely unspecified by the interface. This boils down to the following definition.

Definition 2.8 (Feature model interface)**.** A feature model $M' = (F', P')$ is an interface of another feature model $M = (F, P)$, denoted as $M' \preceq_c M$, if

- $F' \subseteq F$, and
- $P' = \{\, p \cap F' \mid p \in P \,\}$.

Example 2.5. Recall the GPL feature model (F, P) from Figure 2.2. Now let $F' = \{G, E, A, S\}$ be the set of interface features. From each configuration in P, we remove all other features, and obtain the set

$$\begin{aligned}
P' =\{&\{G, E\},\\
&\{G, E, A, S\},\\
&\{G, E, A\}\}.
\end{aligned}$$

Hence, the feature model $M' = (F', P')$ is an interface of $M = (F, P)$ (i.e., $M' \preceq_c M$). Since some of the reduced configurations are no longer distinguishable in terms of features, the number of configurations in this interface is smaller than the number of configurations in the original model M. Note however that all dependencies between the remaining features are preserved in each configuration. For example, there still is no configuration where Shortest path is selected, but Algorithms is not.

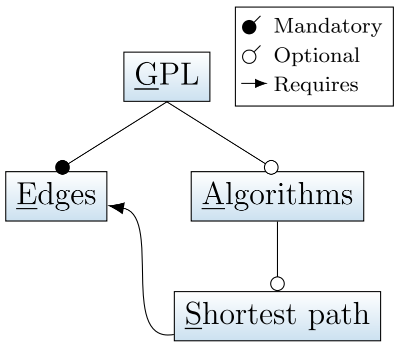


Figure 2.5: Feature diagram for the interface from Example 2.5

In Figure 2.5, we show the interface from the previous example as a feature diagram. Obviously, the features not in F' are not present in the diagram. In addition, we have to adjust the dependencies between the remaining features. The Shortest path feature is now optional, as in the original model there was a configuration where Algorithms was selected, but Shortest path was not. Moreover, Shortest path now requires Edges instead of Weighted, since interfaces preserve transitive dependencies: Shortest path required Weighted, which in turn required Edges, because it was a direct child of Edges. In this special case, we could have actually omitted the requires constraint altogether, since Edges is mandatory and therefore present in all variants. Concerning the diagram structure, the interface diagram omits the entire left subtree, as well as a part of the right subtree. Note however that it would also be possible to construct interface diagrams where only intermediate nodes (e.g., the Edges feature) are removed.

Feature model analysis with interfaces

Since feature model interfaces preserve all dependencies between their contained features, we can analyze interfaces instead of the full model, provided that we are interested only in features that are present in the particular interface. We illustrate this approach using the core feature analysis[4]: A feature is a *core feature* if it is contained in every valid configuration [Benavides et al., 2010]. With the following theorem, we formalize that an interface $M' = (F', P')$ of another model M preserves the core property for all features in F' (i.e., the interface features).

Theorem 2.2. *Let M and $M' = (F', P')$ be feature models such that $M' \preceq_c M$. Then $t \in F'$ is a core feature of M' if and only if it is a core feature of M.*

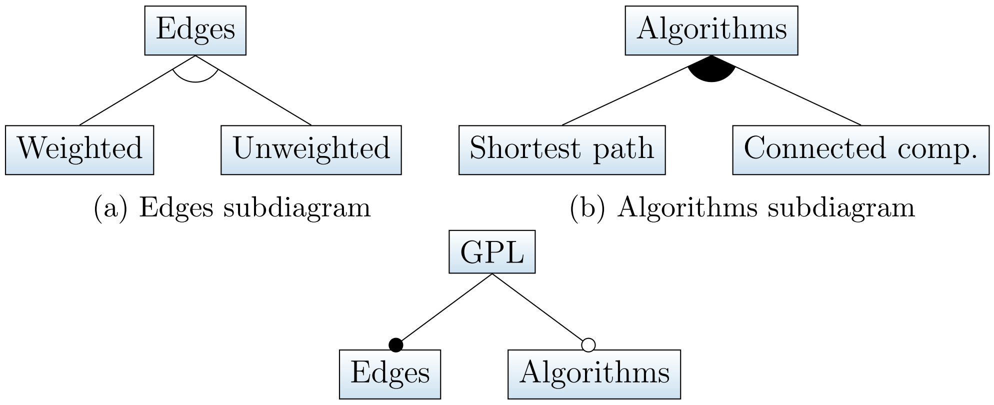
Proof. See Theorem 13 from Schröter et al. [2016]. □

[4]For further analyses that can be used this way, see the work of Schröter et al. [2016].

For example, if we want to know whether the Edges and Algorithms features of the GPL feature model are core features, we could compute the set of all core features for this model. This would result in the set $\{G, E\}$. Therefore, Edges is a core feature, whereas Algorithms is not. However, we could instead compute the core features of the interface from Example 2.5, since it also contains both features. In this case, we would again obtain the set $\{G, E\}$, from which we could draw the same conclusion. By contrast, we could not use this interface to decide whether Unweighted is a core feature.

Composing interfaces

Hiding irrelevant features with interfaces enables compositional analysis. Suppose a scenario where a feature model is composed of several smaller models that are maintained independently. If we are only interested in analyzing a subset of all features of the composed model, we can instead compose *interfaces* of the submodels, which then yields an interface of the overall model. This approach has several advantages. First, the composed interface is potentially smaller, which speeds up the analysis. Second, we can compute each interface in isolation. Thus, if a submodel changes, we do not have to recompute interfaces of the other submodels. Finally, if a submodel only changes in a way that its interface remains the same, we know that the analysis result of the composed interface does not change.



(a) Edges subdiagram (b) Algorithms subdiagram (c) Abstract GPL diagram

Figure 2.6: GPL feature diagram decomposed into three parts

To compose two feature diagrams, we can add one diagram as a subtree of the other diagram, possibly with additional constraints. For example, we could compose our GPL feature diagram from Figure 2.1 from three smaller diagrams and an additional constraint. In Figure 2.6, we show these diagrams. Diagram 2.6a represents the subtree of the Edges feature, and similarly, the Algorithms feature is modeled by Diagram 2.6b. To obtain the complete GPL diagram, we integrate the two diagrams into Diagram 2.6c using their root features, and add the dependency of Shortest path on Weighted. Similarly, we can instead use interfaces of the diagrams for composition.

However, we cannot remove arbitrary features from the submodels to construct their interfaces. Otherwise, the composition of those interfaces would not be an interface of the overall model. More precisely, the submodel interfaces must retain all features that are shared with some other model of the composition, since those features express the dependencies between models[5]. In particular, the root feature of each subdiagram cannot be removed, as well as all features that are involved in constraints between subdiagrams. By contrast, we may remove features that are only present in constraints *within* a subdiagram.

We illustrate these restrictions on interface features using the GPL submodels from Figure 2.6. From the Edges subdiagram, we may only remove the Unweighted feature, as the Edges feature connects the tree to the abstract GPL diagram, and the Weighted feature is present in the constraint between this subdiagram and the Algorithms subdiagram. For similar reasons, we can only remove the Connected components feature from the Algorithms subdiagram, and we must retain the Algorithms and Edges features in interfaces of the abstract GPL diagram.

By contrast, if we would for example construct an interface of the Edges subdiagram that lacks the Weighted feature, we would lose the information that Weighted and Unweighted are mutually exclusive. Thus, the composition of this interface with the other two diagrams plus the requires constraint between Shortest path and Weighted would contain spurious variants: We could now select Shortest path together with Unweighted. Hence, this composition would not be an interface of the original model.

Feature model slicing

The formal definition of feature model interfaces already gives instructions on how to compute an interface from a set of desired interface features F'. However, since this approach is based on configuration semantics, we cannot use it in practice. Instead, there is a technique called *feature model slicing* [Acher et al., 2011], which operates on the propositional representation of a feature model.

Feature model slicing was first proposed using existential quantification over propositional formulas [Acher et al., 2011]. To remove an atom t from a formula φ, it is existentially quantified, while the other atoms remain free. Intuitively, the resulting formula $\varphi' = \exists t.\varphi$ expresses that there exists some arbitrary truth value for t such that φ can be satisfied. As a consequence, the valuations that satisfy φ' are the same valuations that satisfy φ, except for t. In particular, valuations for φ' may omit t altogether, since it is bound by the quantifier. In a scenario where atoms correspond to features, all dependencies are therefore preserved, precisely as required by the definition of feature model interfaces[6].

Existential quantification for propositional formulas is commonly defined similar to a *Shannon expansion*, but without replicating the atom with each cofactor [Huth and Ryan, 2004]. In particular, to eliminate the existential quantifier, two formulas are derived from φ by replacing t with either $\bot$ or $\top$. The existential quantification over t is the disjunction of these formulas. The resulting formula then contains neither the atom t nor the quantifier. We use this technique to define a *slice* operation on propositional formulas, which removes arbitrary sets of atoms.

[5]Of course, we could directly construct an interface of the overall model, which would be correct for arbitrary removed features. However, this approach would not be compositional, as we would not construct submodel interfaces in isolation.

[6]We formally prove this in Chapter 3.

Definition 2.9 (Slice)**.** Let φ be a propositional formula, and $A = \{t_1, \ldots, t_n\}$ be a set of atoms. The *slice* $s(\varphi, A)$ is defined as

$$s(\varphi, A) = \exists t_1. \ldots \exists t_n.\varphi,$$

where

$$\exists t.\psi = \psi[t \leftarrow \bot] \vee \psi[t \leftarrow \top],$$

and $\psi[t \leftarrow v]$ is the formula ψ with every occurrence of atom t replaced by term v.

Example 2.6. Consider the formula $\varphi = (P \Rightarrow Q) \wedge (Q \Rightarrow R)$. We use the slice operation to remove the atom Q from φ. First, we expand the definition of $s(\cdot, \cdot)$. Then, we apply trivial simplifications to the formula to obtain the final result.

$$\begin{aligned} s(\varphi, \{Q\}) &= \exists Q.((P \Rightarrow Q) \wedge (Q \Rightarrow R)) \\ &= ((P \Rightarrow \bot) \wedge (\bot \Rightarrow R)) \vee ((P \Rightarrow \top) \wedge (\top \Rightarrow R)) \\ &\equiv (\neg P \wedge \top) \vee (\top \wedge R) \\ &\equiv \neg P \vee R \\ &\equiv P \Rightarrow R \end{aligned}$$

In the original formula φ, atom P transitively implied R through Q. Since slicing preserves all dependencies, this implication is now explicit in the sliced formula.

As slicing is based on the substitution of atoms, it has to be applied only to those parts of a formula that contain the atoms to be sliced. In other words, terms without those atoms are not altered by the slicing operation. In the following theorem, we formalize this property for the conjunction of two formulas, because we will need it for a proof later in this thesis.

Lemma 2.1. *Let φ, ψ be propositional formulas, and t be an atom such that $t \notin a(\psi)$. Then*

$$\exists t.(\varphi \wedge \psi) \equiv (\exists t.\varphi) \wedge \psi.$$

Proof. See Büning and Bubeck [2009]. □

Example 2.7. The formula $\exists P.(\neg P \wedge Q)$ is equivalent to the formula $(\exists P.\neg P) \wedge Q$, since Q is not affected by the quantifier:

$$\begin{aligned} \exists P.(\neg P \wedge Q) &= (\top \wedge Q) \vee (\bot \wedge Q) \\ &\equiv Q \\ &\equiv (\top \vee \bot) \wedge Q \\ &\equiv (\exists P.\neg P) \wedge Q \end{aligned}$$

Besides existential quantification, there are other approaches to implement slicing. For a method that is based on logical resolution [Huth and Ryan, 2004], see the work of Krieter et al. [2016].

2.2.3 Implementation techniques

In this section, we illustrate the implementation of a software product line. To automatically derive a variant from a configuration, software product lines have a mechanism to map features to code artifacts. Those artifacts are then selected according to the configuration, and integrated to form the particular variant.

We illustrate such a variability mechanism using the *annotative* approach: Source code is mapped to a particular feature (or certain combinations of features) using some language construct together with a feature name or a conditional expression that ranges over the available features. Such expressions are called *presence conditions*, since they control whether the annotated functionality is included in a particular variant with respect to the selected features. In contrast to *compositional* approaches such as feature-oriented programming [Batory et al., 2004; Apel et al., 2013b], code belonging to a particular feature is potentially scattered over the whole code base.

The standard example of this technique is the *C Preprocessor* (*CPP*) [Kernighan and Ritchie, 1988], which is widely used in both open source and industrial software systems [Hunsen et al., 2015] such as the Linux kernel. Through its **`#ifdef`** directive, the CPP facilitates *conditional compilation*. Typically, a developer (or user) uses some command line mechanism or menu (e.g., *Kconfig* [Berger et al., 2010]) to pass the desired features for a concrete variant to the preprocessor. The CPP then derives the corresponding variant from the code base by excluding code of any features that have not been selected. Afterwards, the resulting code is fed into the compiler to obtain an executable program.

In Listing 2.1, we show an example of preprocessor-based variability. The listing contains a possible implementation fragment for the GPL. In Lines 1–17, we define data structures for nodes, edges and graphs. Lines 19–28 contain a function to compute the shortest path between two nodes `from` and `to` in a graph `g`. To map the modeled features Weighted and Shortest path to their implementation code, there are corresponding preprocessor macros `WEIGHTED` and `SHORTEST_PATH`. The **`#ifdef`** directives in Lines 8–10 and 18–29 check whether these macros are defined, and include or exclude the enclosed code accordingly. In other words, these sections of the code are *variable*, whereas the remaining code is present in all variants. Since the Weighted feature specifies a weight value for edges, the code for this feature defines a member **`double`** `weight` for the edge structure. Similarly, the `shortest_path` function is annotated with the `SHORTEST_PATH` macro. Thus, `WEIGHTED` and `SHORTEST_PATH` are simple examples of presence conditions.

To generate the implementation for a given product line configuration from Listing 2.1, the preprocessor is called with arguments that define all corresponding feature macros. We can therefore derive a total of three variants from the code fragment, since including `SHORTEST_PATH` code but excluding `WEIGHTED` is not allowed according to the feature model. We show the variants in Figure 2.7, where we highlight all code belonging to a feature in blue.

Note that we have chosen the CPP to illustrate variability because of its widespread use. However, this thesis also applies to other mechanisms that aggregate all variants

```
typedef struct {
    char* name;
} node;

typedef struct {
    node* source;
    node* target;
#ifdef WEIGHTED
    double weight;
#endif
} edge;

typedef struct {
    node* nodes;
    edge* edges;
} graph;

#ifdef SHORTEST_PATH
edge* shortest_path(graph g,
        node* from, node* to) {
    // ...
    while(...) {
        edge next_edge = ...;
        double weight
            = next_edge.weight;
        // ...
    } // ...
}
#endif
```

Listing 2.1: GPL code fragment with CPP directives

in a single code base and that are capable of mapping feature conditions to code fragments. For example, the variability in Listing 2.1 could also be implemented using normal **`if`** statements and distinguished variables for features. Consequently, derivation of the variant then happens at run time rather than compile time. Accordingly, this approach is called *runtime variability* [Apel et al., 2013a]. There are also several further mechanisms for variability with different characteristics and flexibility, such as plugin frameworks, Feature-oriented [Batory et al., 2004; Apel et al., 2013b] or Delta-oriented programming [Schaefer et al., 2010; Koscielny et al., 2014].

2.2.4 Analysis of product lines

Static analysis of software product lines such as type checking or data-flow analysis leads to scalability issues. In the worst case, a system with n independent[7] configuration options gives rise to 2^n derivable variants. Analyzing the whole system (i.e., *all* variants) by analyzing each variant on its own is therefore infeasible because of the exponential complexity. Due to feature model constraints and alternative features or annotations (e.g., `#ifdef ... #else ...`), in general there is no configuration that includes all features. Moreover, *feature interactions* [Apel et al., 2013c] lead to

[7]i.e., no feature model constraints

```c
typedef struct {
    char* name;
} node;

typedef struct {
    node* source;
    node* target;
} edge;

typedef struct {
    node* nodes;
    edge* edges;
} graph;
```

(a) None

```c
typedef struct {
    char* name;
} node;

typedef struct {
    node* source;
    node* target;
    double weight;
} edge;

typedef struct {
    node* nodes;
    edge* edges;
} graph;
```

(b) Weighted

```c
typedef struct {
    char* name;
} node;

typedef struct {
    node* source;
    node* target;
    double weight;
} edge;

typedef struct {
    node* nodes;
    edge* edges;
} graph;
edge* shortest_path(graph g,
        node* from, node* to) {
    // ...
    while(...) {
        edge next_edge = ...;
        double weight
            = next_edge.weight;
        // ...
    } // ...
}
```

(c) Weighted and Shortest path

Figure 2.7: Variants derivable from Listing 2.1

behavior that is specific to the presence or absence of some combinations of features. Therefore, analyzing such a hypothetical feature-complete variant would generally yield incomplete results.

Fortunately, in typical product lines large fractions of functionality are shared among many variants, as the code of any feature is potentially present in $2^n/2$ variants. *Family-based analysis* [Thüm et al., 2014a] exploits this redundancy in order to scale well-known analyses such as type checking, data-flow analysis and testing to large numbers of features. In a nutshell, those analyses attempt to share information for as many variants as possible during the process. To this end, no variants are derived in advance. Instead the analysis only considers differences locally when variability (e.g., an annotation) is encountered. Consequently, such analyses are *variability aware*, as they are able to interpret presence conditions together with the actual code. However, this implies that existing off-the-shelf analysis implementations cannot be used as-is for this purpose. Instead, the analysis must be modified (conceptually and programmatically) to handle product lines rather than single products.

A family-based analysis processes the presence conditions that arise from the annotations together with the actual analysis information. For example, if the analysis combines information from several annotated elements (e.g., program statements), the result must be annotated with the conjunction of all original conditions. Furthermore, such elements should only be analyzed in combination if they are present *together* in some variant. To this end, the resulting presence condition must be checked against the feature model. Therefore, family-based analysis requires efficient techniques for reasoning about presence conditions.

Type checking

We use variability-aware type checking [Kästner et al., 2012] as a concrete example of a family-based analysis. In a nutshell, type checking verifies that variables and functions, and also names of complex types (e.g., **`struct`**s) in a program are only used according to their declared types. Details depend on the type system of the particular programming language. For example, in C we cannot assign an integer value to a variable with a **`struct`** type, or call a function that is not declared.

C compilers type check a program during compilation after preprocessing. At this point, all preprocessor directives have been removed. Hence, every variable has a definite type, and functions are either declared or not. By contrast, when viewing the product line as a whole, uses of variables may refer to different declarations in different variants, and declarations may even be absent in some variants. For example, in our GPL implementation from Listing 2.1, the struct member `edge.weight` is only present in variants with the Weight feature. Thus, code that uses this member must ensure with its presence condition that Weighted is selected. Indeed, Line 25 in the listing uses the `weight` member, and its presence condition is Shortest path, which according to the GPL feature model implies that also the code for Weighted is included. Hence, variability-aware type checking verifies that the presence condition φ_u of an identifier usage implies the presence condition φ_d of the referenced declaration by the feature model FM; formally $FM \vDash \varphi_u \Rightarrow \varphi_d$.

```
 1 typedef struct {
 2     char* name;
 3 } node;

 5 typedef struct {
 6     node* source;
 7     node* target;
 8 #ifdef WEIGHTED
 9     double weight;
10 #endif
11 } edge;

18 #ifdef SHORTEST_PATH

23         edge next_edge = ...;
24         double weight
25             = next_edge.weight;

29 #endif
```

Listing 2.2: Example of presence conditions in type checking

In Listing 2.2, we show an excerpt of Listing 2.1 to illustrate presence conditions that arise during type checking. The reference to the `edge` data type in Line 19 has presence condition Shortest path, whereas the declaration of the type has presence condition $\top$, since it has no annotation (Lines 5ff.). Therefore, we need to check whether *Shortestpath* $\Rightarrow \top$, which is trivially true. Furthermore, for the `weight` member access of the `edge` struct in Line 25, we need to verify that *Shortestpath* $\Rightarrow$ *Weighted* as explained above. In the same line, there is also a use of the `next_edge` variable, which is declared in the same annotation. Thus, the implication to check is *Shortestpath* $\Rightarrow$ *Shortestpath*, which is also trivially true. Similarly, the conditions to check the `node` type references in Lines 6f. are $\top \Rightarrow \top$.

Dead code analysis

Apart from family-based analyses, which are derived from existing single-variant analyses, product lines also give rise to new analyses types, which are concerned with the variability itself. The void feature model analysis as explained in Section 2.2.1 is an example of such an analysis. Another example is the so-called *dead code* analysis [Tartler et al., 2011], which deals with code variability itself. Developers may by mistake annotate code with presence conditions that are not satisfiable with respect to the feature model. Hence, such code fragments are not present in any variant at all. From a developer perspective, this is undesirable and should therefore be detected and rectified: Although the code cannot actually be executed, it is still maintained. Furthermore, such a defect can indicate a problem with the feature model, or uncover a misconception about the interaction of certain features.

```
1 #ifdef SHORTEST_PATH
2 // ...
3 #ifdef UNWEIGHTED
4     double weight = 1;
5 #endif
6 // ...
7 #endif
```

Listing 2.3: Example of dead code

To perform a dead code analysis, we simply check the presence condition of each annotation for satisifiability. In our GPL implementation from Listing 2.1, we have no dead code, since the conditions of both annotations are simply features that are not dead. That is, there exist variants in which those features are selected (see the configuration semantics of the GPL feature model in Figure 2.2). In Listing 2.3, we show a possible fragment of dead code in the GPL. The code is concerned with the implementation of the Shortest path feature. When the GPL is configured with unweighted edges, the algorithm shall assume an edge weight of 1 for each edge. This is realized with a nested annotation (Lines 3-5). Therefore, the statement that assigns the default value (Line 4) has the conjunction of the outer and inner annotation conditions as its presence condition, which results in *Unweighted* $\wedge$ *Shortestpath*. However, from the GPL feature model we know that there is no such variant where both features are selected together. Hence, the statement in Line 4 is dead.

3. Reasoning with decomposed feature models

In this chapter, we explain our approach for reasoning with decomposed feature models. Our overall goal is to reduce computation time when reasoning about a propositional formula in the context of a feature model. To this end, we define an algorithm that composes a specific model for each solver query. This model is equivalent the original feature model with respect to the given query, but potentially contains less features and therefore facilitates faster reasoning.

First, we formalize our general method in terms of propositional logic. In particular, we define generic properties for decompositions of propositional formulas. We then introduce two strategies for reasoning with such decompositions, and show that these strategies are correct. The first strategy always composes a specific formula for each query, whereas the second strategy resorts to using the original formula whenever the composition process would be more computationally expensive.

Second, we define a concrete decomposition for feature diagrams that is guided by their hierarchical structure. To this end, we first formalize a generic notion of feature diagrams, which embraces all definitional flavors that permit a complete semantics in propositional logic. Then, we define an operation that decomposes such a feature diagram with respect to a set of selected features. In particular, each component reflects the subtree of a selected feature. We then apply feature model slicing to obtain a reduced version of the original model, which can be correctly re-extended by sets of those components.

We finally prove that the resulting decomposition adheres to our generic decomposition properties, and thus can be used for compositional reasoning. To this end, we use the notion of feature model interfaces. We provide a characterization of feature model interfaces in propositional logic, and subsequently show that, for a subset of features, such interfaces are equivalent to their concretizations.

3.1 Selectable decompositions

Family-based analyses need to decide entailments in the form of $\Phi \vDash \psi$, where Φ represents a feature model, and ψ is some aggregated condition. This is equivalent to deciding the unsatisfiability of $\Phi \wedge \neg\psi$, which can be achieved through a SAT solver. While the performance of such solvers depends on several characteristics of Φ and ψ, such as the number of valid products [Liang et al., 2015], the worst-case complexity is exponential in the number of atoms in the input formula. Therefore, we use this measure as a heuristic to assess the complexity of the solver input. Consequently, our objective is to remove atoms from $\Phi \wedge \neg\psi$ that are not necessary to decide $\Phi \vDash \psi$. In particular, we focus on the atoms of Φ (i.e., the feature model).

We formalize the standard procedure for deciding entailment with a SAT solver in the following reasoning strategy FULL-MODEL. This strategy serves as a baseline against which we evaluate our method in Chapter 4.

Strategy 3.1 (FULL-MODEL)**.** Given a feature model represented as a propositional formula Φ, and a propositional query formula ψ, decide whether $\Phi \wedge \neg\psi$ is unsatisfiable. If the formula is unsatisfiable, then return that $\Phi \vDash \psi$, otherwise return that $\Phi \nvDash \psi$.

Often, not all parts of Φ are relevant for deciding whether $\Phi \vDash \psi$ holds for some query ψ. For example, let $\Phi = P \wedge Q \wedge R$, where P, Q and R are atoms. To prove $\Phi \vDash P$, we do not need the $Q \wedge R$ part, since it does not influence the truth value of P. Instead, we could use a part Φ' that is equivalent to Φ with respect to the query P. In this example, Φ' is simply P. Indeed, we can easily construct a sound and complete abstraction of Φ for any ψ in this special case: We first disassemble Φ into the set $\Gamma = \{P, Q, R\}$ of its subformulas. For every query ψ, we select precisely those elements from Γ that share an atom with ψ, and decide if their conjunction entails ψ. For example, let $\psi = P \wedge R$. To prove $\Phi \vDash P \wedge R$, we would then merely prove $P \wedge R \vDash P \wedge R$ instead of $P \wedge Q \wedge R \vDash P \wedge R$.

We formalize this atom-based selection from a set of formulas in the following definition.

Definition 3.1 (Selection)**.** The *selection* $\Delta|_A$ from a set of propositional formulas Δ with respect to a set of atomic propositions A is defined as

$$\Delta|_A = \{\, \delta \in \Delta \mid a(\delta) \cap A \neq \varnothing \,\}\,.$$

Clearly, the previous example does not account for more complex situations such as transitive implications. Consider for example $\Gamma = \{P, P \Rightarrow X, X \Rightarrow Q\}$ and $\psi = Q$. Using the selection $\Gamma|_{\{Q\}} = \{X \Rightarrow Q\}$, the entailment $\Gamma|_{\{Q\}} \vDash Q$ does not hold, although $\Gamma \vDash Q$ holds. Therefore, simple atom-based selection does not guarantee semantic completeness.

We can remedy this shortcoming in several ways. For example, we could extend the selection operation to the transitive closure of formulas connected by at least one atom. Although this solution would be semantically complete, it tends to select superfluous elements from Γ that are actually irrelevant. The most precise method would be a semantic analysis of Γ with respect to ψ that returns only relevant formulas. However, this would involve additional round trips to the solver with subsets of Γ. We rule out this solution, since it potentially leads to more computations at runtime. Instead, we strive for a method that reduces solver effort at the cost of additional preprocessing on the feature model, and admits a more precise selection than transitive closure. To this end, we propose the following method: We decompose the original formula into *two* sets of formulas. One of the sets consists of parts of the original formula. We apply the atom-based selection to this set for each query. Independently from the query, we add all formulas from the other set to the selection, and use the resulting set of formulas for reasoning (instead of the original formula). Intuitively, the latter set represents an abstraction of the original formula, which can be refined by elements of the former set. To ensure correctness, we demand that such a decomposition must be sound and complete for all queries when used as described. We formalize this property in the following definition.

Definition 3.2 (Selectable decomposition)**.** Let Γ, Δ be sets of propositional formulas. The pair (Γ, Δ) is a *selectable decomposition* of a propositional formula Φ, if for all formulas φ such that $a(\varphi) \subseteq a(\Phi)$,

$$\Delta|_{a(\varphi)} \cup \Gamma \models \varphi \text{ if and only if } \Phi \models \varphi.$$

Note that this definition does not require the decomposition elements to have disjoint sets of atoms.

Example 3.1. Let $\Phi = P \wedge Q \wedge (P \Rightarrow R) \wedge (Q \Rightarrow S)$, and let $\Gamma = \{P \wedge Q\}$ and $\Delta = \{P \Rightarrow R, Q \Rightarrow S\}$. Then (Γ, Δ) is a selectable decomposition of Φ. Now assume that we want to decide whether $\Phi \models S$. The selection $\Delta|_{\{S\}}$ yields $\{Q \Rightarrow S\}$, which would not be sufficient to prove S. However, if we combine the selection with Γ, then indeed $\Gamma \cup \Delta|_{\{S\}} \models S$. Clearly, also $\Phi \models S$. If we furthermore try to decide $\Phi \models Q$ using this method, we see that atom-based selection is an overapproximation: The selection again is $\{Q \Rightarrow S\}$, although semantically, Γ alone would suffice in this case.

The previous example suggests that a selectable decomposition must consist of verbatim parts of the original formula. However, we could also derive new formulas from the original formula, and use them to construct a decomposition.

Example 3.2. Consider $\Phi = (P \Rightarrow Q) \wedge (Q \Rightarrow R)$. From Φ we can deduce the transitive implication $P \Rightarrow R$. Therefore, let $\Gamma = \{P \Rightarrow R\}$ and $\Delta = \{P \Rightarrow Q, Q \Rightarrow R\}$. Then (Γ, Δ) is a selectable decomposition.

We are now able to specify strategies that given a feature model decomposition and a query formula, decide whether the feature model entails the query. The strategies reduce this problem to satisfiability and can therefore be implemented using an arbitrary SAT solver. The REDUCED-MODEL strategy constitutes our basic approach. It selects all formulas from the decomposition that share some feature with the query formula. Subsequently, the strategy uses the conjunction of the selected formulas to compose a potentially smaller model, which is then used for reasoning in place of the original model.

Strategy 3.2 (REDUCED-MODEL)**.** Given a feature model represented as a formula Φ, a query formula ψ, and a selectable decomposition (Γ, Δ) of Φ, decide whether the formula

$$\bigwedge_{\delta \in \Delta'} \delta \wedge \Gamma \wedge \neg\psi,$$

is unsatisfiable, with

$$\Delta' = \Delta|_{a(\psi)}.$$

If the formula is unsatisfiable, then return that $\Phi \vDash \psi$, otherwise return that $\Phi \nvDash \psi$.

During early experiments, we found a computational trade-off between composing a query-specific formula and using the original formula. Querying the solver with a set of formulas was sometimes slower than just using the original formula. The culprit was some formula in the decomposition that contained approximately half of the features of the original model. Consequently, the formula was so large that composing within the solver took longer than the actual solving[1]. Therefore, we introduce a measure $m(\Delta')$ that represents the composition overhead for the selection Δ'. Beyond a fixed threshold overhead τ we use the FULL-MODEL strategy instead of REDUCED-MODEL. In Chapter 4, we evaluate candidates for such a measure and determine threshold values.

Strategy 3.3 (ADAPTIVE-MODEL)**.** Given a model formula Φ, a query formula ψ, a selectable decomposition (Γ, Δ) of Φ, and a threshold value τ,

- use strategy REDUCED-MODEL if $m\left(\Delta|_{a(\psi)}\right) < \tau$,
- use strategy FULL-MODEL otherwise.

We suspect that the optimal threshold depends on several factors, such as the size distribution of the decomposition elements, the queries and the internals of the particular solver algorithm.

Finally, we prove the correctness of all three strategies.

Theorem 3.1 (Correctness)**.** *The strategies* FULL-MODEL*,* REDUCED-MODEL*, and* ADAPTIVE-MODEL *are correct. To be precise, they decide $\Phi \vDash \psi$, where Φ represents a feature model and ψ is a query formula.*

Proof. Strategy FULL-MODEL decides the unsatisfiability of $\Phi \wedge \neg\psi$, which is equivalent to deciding if $\Phi \vDash \psi$. Therefore, the strategy is correct.

Strategy REDUCED-MODEL decides the unsatisfiability of $\bigwedge \left\{\Delta|_{a(\psi)}\right\} \wedge \Gamma \wedge \neg\psi$, which is equivalent to deciding $\Delta|_{a(\psi)} \cup \Gamma \vDash \psi$. Since (Γ, Δ) is a selectable decomposition of Φ, claim immediately follows by the definition.

Strategy ADAPTIVE-MODEL is correct, since it merely delegates to either FULL-MODEL or ADAPTIVE-MODEL, which are both correct. □

[1]We explain this in more detail in Chapter 4.

3.2 Decomposing feature diagrams

Our decomposition definition does not describe how to actually decompose feature models. Indeed, there are several decompositions for a single model. In this work, we propose a decomposition technique adapted from Schröter et al. [Schröter et al., 2016], which incorporates the structure of feature diagrams. Since a feature diagram hierarchically represents the functionality of a product line, we hypothesize that implementation artifacts also reflect this structure. More precisely, we expect that each implementation unit (e.g., module consisting of a set of source code files) corresponds to some subtree in the feature diagram. Consequently, a family-based analysis of each implementation unit mostly leads to solver queries that only contain features from the implementation subtree. Therefore, most of the features in other subtrees are irrelevant and could thus be omitted from reasoning. However, we cannot simply use only the subtrees for reasoning, since we must respect cross-tree constraints and group constraints of parent features. Moreover, reasoning must remain correct whenever the query involves features from multiple subtrees.

In this section, we describe a method for decomposing feature diagrams according to their tree structure. Using the concept of *feature model interfaces*, we subsequently prove that our method produces selectable compositions, and can therefore be used for correct reasoning as described in Section 3.1.

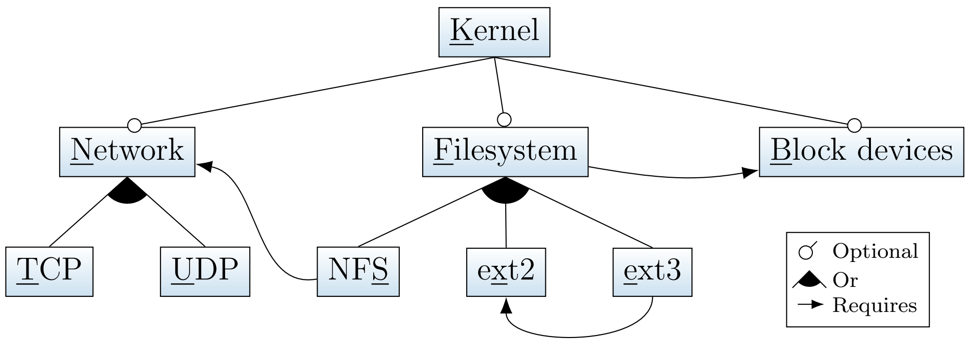


Figure 3.1: Feature diagram of simple kernel product line

3.2.1 Overall approach and running example

We illustrate our method using a simple fictional system kernel product line as a running example, which is loosely inspired by the Linux kernel. In Figure 3.1, we show its feature diagram. The kernel supports networking, some filesystems, and block devices such as hard disks. All of those features are optional. However, file system functionality requires block device support, and the network file system[2] (NFS) requires networking. We show fragments of corresponding implementation files in Figure 3.2. File "network.c" implements the networking and all of its subfeatures. For example, networking code needs to allocate and deallocate memory for socket data structures (functions `create_socket` and `free_socket`). The function `close_tcp` of the TCP feature therefore uses the latter function to free up memory after closing a connection. In file "filesystem.c", which implements the Filesystem

[2]https://tools.ietf.org/html/rfc1094

feature, we show an exemplary function of the NFS feature. Since NFS mounts remote file systems via network, it needs to create sockets, and thus references `create_socket` from the Network feature.

```
#ifdef NETWORK
int create_socket(...);
void free_socket(...);
...
#endif


...


#ifdef TCP
void close_tcp(int s) {
    ...
    free_socket(s);
}
#endif

#ifdef UDP
...
#endif
```

(a) network.c

```
#ifdef FILESYSTEM
...
#endif


...


#ifdef NFS
int mount_nfs(...) {
    create_socket(...);
    ...
}
#endif

...
```

(b) filesystem.c

Figure 3.2: Implementation fragments of the kernel product line

If we want to typecheck file "network.c" with a family-based technique, we would for example need to check whether `free_socket` is declared in every variant in which the TCP feature is selected. Since only the Network feature declares such a function, we need the solver to decide whether $FM \vDash TCP \Rightarrow Network$, where FM represents the feature diagram from Figure 3.1. Since basic networking typically is concerned with neither file systems nor block devices, the queries that arise from file "network.c" would only consist of features from the Network subtree of the feature diagram. Ideally, we would want to only use this subtree instead of the full model FM in order to accelerate reasoning. However, such an approach would be incorrect in general: To typecheck file "filesystem.c", we need to check if $FM \vDash NFS \Rightarrow Network$, since the NFS feature uses a function that is defined by the Network feature. If we consider the Filesystem subtree in isolation, we cannot prove that implication, although it actually holds. Instead, we also need the Network subtree and the shared constraint.

In the above example, using the two subtrees without the Kernel feature would suffice. However, in general even such sets of subtrees are *incomplete* with respect to reasoning. We show another two examples of constructs that lead to incompleteness in Figure 3.3 (highlighted in red). Since F_1 requires F_3, and F_3 requires F_2, feature F_1 also transitively requires F_2. However, we cannot prove this relation using the subtree S_1 alone. A more subtle problem arises from the alternative group of S_1 and S_2. Since S_1 and S_2 are mutually exclusive, their subfeatures cannot occur together in any variant, thus for example $F_1 \Rightarrow \neg F_4$ holds, which we cannot prove even when we consider both subtrees S_1 and S_2 (without B).

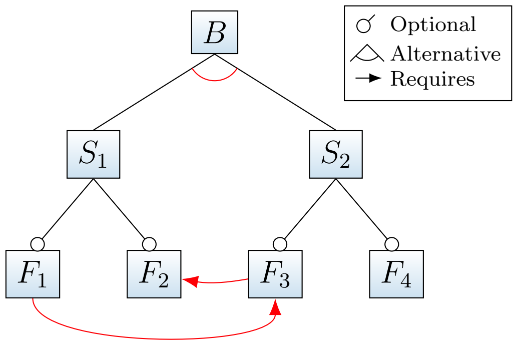


Figure 3.3: Additional constructs that lead to incompleteness

Consequently, we need a more sophisticated decomposition method that accounts for dependencies between subtrees of implementation units. Intuitively, dependencies *between* those subtrees reflect dependencies between implementation units, such as global[3] declarations or a certain programmatic behavior, whereas dependencies *within* a subtree represent implementation details of a single (compound) feature.

Therefore, we propose the following decomposition technique: Given a set of subtrees, we construct an interface of the original feature diagram that models exactly the dependencies between those subtrees. In the interface, we omit all subtree features that are not directly (i.e., not transitively) involved in such dependencies. We call such features *local* features. The interface is already complete with respect to its features. However, when a query contains a local feature, we refine the interface by reinstantiating the full subtree that the local feature belongs to. Since we retain all cross-tree dependencies in the abstraction, we also ensure completeness for all reinstantiated subtrees. To abstract away from different definitional styles of feature diagrams, we perform interface construction and reinstantiation on the semantic level (i.e., propositional formulas).

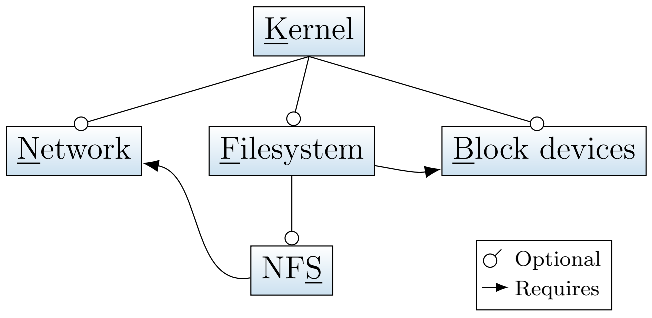


Figure 3.4: Interface of the feature diagram in Figure 3.1

We illustrate the decomposition using our running example from Figure 3.1. First, we select the features Network, Filesystem and Block devices as subtree roots, since they correspond to our implementation units. Then, we identify the local features of each subtree. In the example, the features TCP, UDP, ext2, and ext3 are local, since they are not involved in constraints between subtrees. Finally, we construct the interface by slicing out those features. In Figure 3.4, we show the resulting interface.

[3]in terms of features

Now reconsider the query *TCP* $\Rightarrow$ *Network*. As the TCP feature was local, it is not present in the interface. We therefore need to reinstantiate the Network subtree before reasoning. We show the resulting feature diagram in Figure 3.5. Similarly, we would reinstantiate the Filesystem subtree when needed.

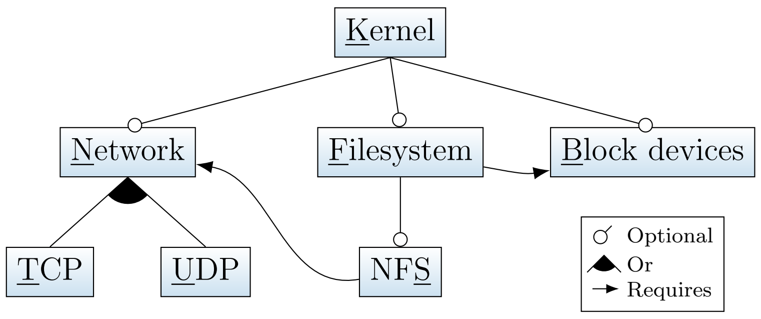


Figure 3.5: Interface from Figure 3.4 with reinstantiated Network subtree

In the remainder of this section, we formalize the decomposition method we have described above, and prove that it satisfies the properties of a selectable composition from Definition 3.2. To this end, we first formalize feature diagrams in a generic way. Then, we provide a formal semantics for this definition, which is based on propositional logic. Using this propositional representation of feature diagrams, we define our decomposition operation by means of feature model slicing. We then specify feature model interfaces in terms of propositional logic, and subsequently prove that all slices are interfaces of their original models. Moreover, we prove that reasoning with an interface is sound and complete for all of its features. To establish correctness, we finally show that interfaces with reinstantiated subtrees are still interfaces of the original model.

3.2.2 Formalization of feature diagrams

Since we are only interested in the tree structure, we abstract away from all other syntactic elements of feature diagrams, such as alternative groups or cross-tree constraints. To capture the semantics of such constructs, we provide a set for arbitrary propositional constraints in our definition. The definition also accommodates the various styles of feature diagrams, as long as they admit a complete semantics in propositional logics. As a result, we define feature diagrams as follows.

Definition 3.3 (Feature diagram)**.** A feature diagram is a tuple (F, E, r, Ψ), where

- F is a set of features,
- (F, E) is a directed out-tree[4] with root $r \in F$, and
- Ψ is a set of propositional formulas over F as atoms.

Example 3.3. We illustrate the definition using the feature diagram from our running example (Figure 3.1). For brevity, the letters we have underlined in the diagram shall represent the features. Therefore, the diagram is represented by the tuple $(\mathcal{F}, \mathcal{E}, K, \Psi_G \cup \Psi_C)$, where

- $\mathcal{F} = \{K, N, F, B, T, U, S, x, e\}$,

[4]i.e., edges point away from the root

- $\mathcal{E} = \{(K,N),(K,F),(K,B),(N,T),(N,U),(F,S),(F,x),(F,e)\}$,
- $\Psi_G = \{N \Rightarrow T \vee U, F \Rightarrow S \vee x \vee e\}$ (groups), and
- $\Psi_C = \{S \Rightarrow N, e \Rightarrow x, F \Rightarrow B\}$ (require constraints).

Next, we define subdiagrams that are determined by a subtree root. Such a *subtree diagram* contains all features of its subtree, and all constraints that range over only those features.

Definition 3.4 (Subtree diagram)**.** Let $D = (F, E, r, \Psi)$ be a feature diagram, and $t \in F$ be a feature. The *subtree diagram* $\downarrow t$ is a feature diagram $(F_{\downarrow t}, E_{\downarrow t}, r, \Psi_{\downarrow t})$, where

- $(F_{\downarrow t}, E_{\downarrow t})$ is the subtree in (F, E) with root t, and
- $\Psi_{\downarrow t} = \{\, \psi \in \Psi \mid a(\psi) \subseteq F_{\downarrow t} \,\}$.

As a convention, whenever we write $F_{\downarrow t}$, $E_{\downarrow t}$, or $\Psi_{\downarrow t}$ autonomously, we implicitly refer to a subtree diagram $\downarrow t$ that defines these sets.

Example 3.4. Consider the Filesystem feature (F) of our running example. For $\downarrow F = (\mathcal{F}_{\downarrow F}, \mathcal{E}_{\downarrow F}, F, \Psi_{\downarrow F})$ we have that

- $\mathcal{F}_{\downarrow F} = \{F, S, x, e\}$,
- $\mathcal{E}_{\downarrow F} = \{(F,S),(F,x),(F,e)\}$, and
- $\Psi_{\downarrow F} = \{F \Rightarrow S \vee x \vee e, e \Rightarrow x\}$.

So far we have only defined the syntactic structure of feature diagrams. The purpose of feature diagrams is to specify all valid configurations for a product line (i.e., combinations of selected features). Therefore, propositional logic is an adequate means to assign semantics to feature diagrams. We use features as atomic propositions, and construct a formula which maps all tree edges to implications such that child nodes imply their parents. We simply add all formulas from Ψ. For our later proof, we need to ensure that the semantic formula always contains all features as atoms. Therefore, we add the term $\bot \Rightarrow t$ for each feature. This term does not alter the semantics, hence we will omit it outside of the definition.

Note that from now on, we will use the terms "atom" and "feature" interchangeably, depending on the context.

Definition 3.5 (Propositional semantics)**.** Let $D = (F, E, r, \Psi)$ be a feature diagram. The *propositional semantics* of D, denoted as $\Phi(D)$, are defined as follows:

- $\Phi(D) = \bigwedge_{t \in F} \Phi_E(t) \wedge \Psi$
- $\Phi_E(t) = \bigwedge_{(t,t') \in E} (t' \Rightarrow t) \wedge (\bot \Rightarrow t)$

We omit E in $\Phi_E(t)$ if the context is unambiguous.

Corollary 3.1. *Let $D = (F, E, r, \Psi)$ be a feature diagram. Then $a(\Phi(D)) = F$.*

Example 3.5. Recall the subtree diagram $\downarrow F$ from Example 3.4. The semantics of $\downarrow F$ are expressed as follows:

$$\Phi(\downarrow F) = (S \Rightarrow F) \wedge (x \Rightarrow F) \wedge (e \Rightarrow F) \wedge (F \Rightarrow S \vee x \vee e) \wedge (e \Rightarrow x)$$

We have already informally explained what a local feature is. Intuitively, a feature is local to a subtree, if it has no direct dependencies outside this tree. More formally, the feature must only occur in constraints that belong to the subtree diagram. Since the root of a subtree is always connected to its parent (apart from the global root), we exclude it from the definition.

Definition 3.6 (Local feature)**.** Let $D = (F, E, r, \Psi)$ be a feature diagram and $\downarrow t = (F_{\downarrow t}, E_{\downarrow t}, r, \Psi_{\downarrow t})$ be a subtree diagram of D. A feature r *is local to* $\downarrow t$, if

- $r \in F_{\downarrow t}$,
- $r \neq t$, and
- $r \notin a(\Psi \setminus \Psi_{\downarrow t})$.

We denote the set of local features of a subtree diagram as $l(\downarrow t)$.

Example 3.6. For $\downarrow F$ from Example 3.4, we have that $l(\downarrow F) = \{x, e\}$.

For our decomposition example above, we have implicitly assumed that the chosen subtrees do not overlap. Although recursive decompositions and successive reinstantiations of subtrees might be possible, we leave such situations for future work to simplify the proof. Consequently, we introduce the notion of *disjoint features* to rule out overlapping subtrees.

Definition 3.7 (Disjoint features)**.** Two features t, t' of a feature diagram are *disjoint*, if their subtree diagrams have no common features, formally

$$F_{\downarrow t} \cap F_{\downarrow t'} = \varnothing.$$

Example 3.7. In the feature diagram of our running example, the features N and F are disjoint, whereas K and F are not.

We are now able to formally specify our subtree-guided decomposition of feature diagrams. Given a feature diagram and set of disjoint features, the decomposition produces two sets of formulas. The first set represents a slice of the original diagram such that all local features are removed, whereas the second set contains a subtree diagram for each of the specified disjoint features. These subtree diagrams can then be fully reinstantiated by mere conjunction with the slice.

Definition 3.8 (Subtree decomposition)**.** Let $D = (F, E, r, \Psi)$ be a feature diagram and $S \subseteq F$ be a set of disjoint features. The *subtree decomposition* of D with respect to S, denoted as D/S, is a pair of sets such that

$$D/S = (\{s(\Phi(D), L)\}, \{\, \Phi(\downarrow t) \mid t \in S \,\}),$$

where

$$L = \bigcup_{t \in S} l(\downarrow t).$$

In Figure 3.4 and Figure 3.5, we have illustrated the decomposition of our running example with respect to the set $\{N, F, B\}$ of disjoint features. Note however that the formal definition applies the slice operation to formulas instead of diagrams, and produces propositional formulas instead of feature diagrams and trees. For subsequent reasoning, we only need the propositional representation and are not interested in the diagram structure.

3.2.3 Propositional interfaces

We have based our decomposition operation on feature model slicing. To capture how slices semantically relate to their original models, we adopt the notion of feature model interfaces, which were originally defined on sets of product line configurations. Informally, a feature model is an interface of another feature model, if it admits exactly those configurations for a subset of features that also the original model would admit. We transfer this definition to propositional formulas, which leads to interesting properties that might also be useful in a broader context. Therefore, we formulate the following theory independently of feature diagrams.

To define interfaces for propositional formulas, we first need to define two properties of valuations. Typically, a valuation for a formula is defined for a superset of the formula atoms. Therefore, a valuation might be unnecessarily overspecified, which prevents us from extending it with further atoms in some cases. We thus specify *minimal* valuations, which are defined for exactly those atoms that occur in the formula. Moreover, since we want to extend valuations, we specify a partial order on them: A valuation includes another valuation, if it contains the same assignments for all common features.

Definition 3.9 (Minimal valuation)**.** A valuation α is *minimal* with respect to a propositional formula φ, if α is only defined for atoms in $a(\varphi)$.

Example 3.8. For the formula $P \vee Q$, the valuation $\alpha : \{P, Q\} \to \mathbb{B}$ is minimal, whereas the valuation $\alpha' : \{P, Q, R\} \to \mathbb{B}$ is not.

Definition 3.10. A valuation α *includes* another valuation α', denoted as $\alpha' \subseteq \alpha$, if for all atoms t for which α' is defined, $\alpha'(t) = \alpha(t)$.

Example 3.9. Let $\alpha = \{P \mapsto 1, Q \mapsto 0\}$ and $\alpha' = \{P \mapsto 1, Q \mapsto 0, R \mapsto 0\}$ be valuations. Then $\alpha \subseteq \alpha'$. By contrast, the valuation $\alpha'' = \{P \mapsto 0\}$ is not included in α', formally $\alpha'' \nsubseteq \alpha'$.

Using the two previous definitions, we specify an interface relation on propositional formulas. As in the general notion of an interface, with this definition we abstract away some details of a formula, while the remaining elements exhibit the same semantics as the original formula. To be more precise, an interface formula of a concrete formula contains less atoms. However, regarding the remaining atoms, the interface formula is satisfied by exactly the same valuations as the concrete formula. This property directly corresponds to the subsets of valid configurations of feature model interfaces. As an interesting consequence of this property, interfaces are sound and complete with respect to their atoms. Therefore, when reasoning, we can use an interface instead of the concrete formula, as long as the query ranges only over a subset of the interface atoms.

We can apply every valuation of concrete formula to its interfaces, since such a valuation is defined on all atoms of the interface. By contrast, we cannot simply use an interface valuation to satisfy a concrete formula, since the valuation is generally underspecified. Instead, we demand that there exists another valuation that includes the interface valuation, and satisfies the concrete formula. To avoid that the interface relation is unnecessarily restrictive, we require only minimal valuations to be extensible this way. Otherwise, the interface valuation could also be defined for additional atoms that only occur in the concrete formula, and in such a way that the concrete formula is not satisfied.

Definition 3.11 (Propositional interface)**.** Let φ, φ' be propositional formulas. The formula φ' *is an interface* of formula φ, denoted as $\varphi' \preceq \varphi$, if

1. $a(\varphi') \subseteq a(\varphi)$,
2. every valuation that satisfies φ also satisfies φ', and
3. for every minimal valuation α' that satisfies φ' there exists a valuation $\alpha \supseteq \alpha'$ that satisfies φ.

Note that the interface relation is stronger than entailment and equisatisfiability[5].

Example 3.10. Let $\varphi = P \wedge Q$ and $\varphi' = P$ be propositional formulas. Then $\varphi' \preceq \varphi$. By contrast, let $\psi = P \wedge (P \Rightarrow Q) \wedge (\neg Q \vee R)$ and $\psi' = \neg Q \vee R$. Then $\psi \vDash \psi'$, the formulas are equisatisfiable, and $a(\psi') \subsetneq a(\psi)$. However, $\psi' \not\preceq \psi$, since there exists a valuation α that satisfies ψ' with $\alpha(Q) = 0$, yet ψ can only be satisfied with Q bound to 1.

The following theorem is a central result of this thesis. With it, we prove that we can use interfaces instead of their concretizations to reason about formulas, if these formulas only contain atoms of the interface. In other words, the interface *is equivalent* to the concretization under this restriction. Because of the smaller set of atoms, reasoning with an interface is potentially faster compared to the concrete formula.

Theorem 3.2. *Let Φ, Φ', φ be propositional formulas such that $\Phi' \preceq \Phi$ and $a(\varphi) \subseteq a(\Phi')$. Then $\Phi' \vDash \varphi$ if and only if $\Phi \vDash \varphi$.*

Proof. We directly prove both implications of the equivalent proposition that $\Phi' \wedge \neg\varphi$ is satisfiable if and only if $\Phi \wedge \neg\varphi$ is satisfiable.

"$\Leftarrow$": Let α be a valuation that satisfies $\Phi \wedge \neg\varphi$, then α particularly satisfies Φ. Because $\Phi' \preceq \Phi$, valuation α also satisfies Φ'. Moreover, since α satisfies $\neg\varphi$, and φ only contains atoms from Φ', valuation α satisfies $\Phi' \wedge \neg\varphi$.

"$\Rightarrow$": Let α be a minimal valuation that satisfies $\Phi' \wedge \neg\varphi$. This valuation particularly satisfies Φ'. Since $a(\varphi) \subseteq a(\Phi')$, valuation α is also a minimal satisfying valuation for Φ'. Because $\Phi' \preceq \Phi$, there exists a valuation $\alpha' \supseteq \alpha$ that satisfies Φ. Since α satisfies $\neg\varphi$, valuation α' then also satisfies $\neg\varphi$ and thus satisfies $\Phi \wedge \neg\varphi$. □

The interface relation on formulas is a proper subset of the entailment relation (i.e., $\preceq \subsetneq \vDash$). As a consequence, a concrete formula absorbs its interface in conjunctions, formally $\varphi' \wedge \varphi \equiv \varphi$ [Huth and Ryan, 2004]. We prove this property in the following lemma, which we will use later in the proof of another central theorem to rearrange a formula.

[5]Two propositional formulas are *equisatisfiable* if both are either satisfiable or unsatisfiable [Harrison, 2009].

Lemma 3.1. *Let φ, φ' be propositional formulas such that $\varphi' \preceq \varphi$. Then the following relations hold:*

1. $\varphi \vDash \varphi'$.
2. $\varphi \wedge \varphi' \equiv \varphi$

Proof.
1. Since $\varphi' \preceq \varphi$, every valuation that satisfies φ also satisfies φ'. Therefore $\varphi \vDash \varphi'$.
2. Since $\varphi \vDash \varphi'$ and trivially $\varphi \vDash \varphi$, also $\varphi \vDash \varphi \wedge \varphi'$. Moreover, trivially $\varphi \wedge \varphi' \vDash \varphi$.

□

3.2.4 Properties of the slicing operation

To simplify the proofs of our other two main theorems, we first establish some properties of the slicing operation $s(\cdot, \cdot)$ from Definition 2.9 on page 17.

With the following lemma, we show that a slice satisfies the interface properties with respect to the original formula. Consequently, we can use slicing to produce abstractions of formulas for our reasoning strategies.

Lemma 3.2. *Let Φ be a propositional formula and L be a set of atoms. Then $s(\Phi, L) \preceq \Phi$.*

Proof. By induction over the size of L.

For $L = \varnothing$, we have that $s(\Phi, L) = \Phi$, and trivially $\Phi \preceq \Phi$.

By the inductive hypothesis, $s(\Phi, L) \preceq \Phi$. Now let $t \notin L$ be some atom. Then, by Definition 2.9,

$$\begin{aligned} s(\Phi, L + \{t\}) &= \exists t . s(\Phi, L) \\ &= s(\Phi, L)[t \leftarrow \bot] \vee s(\Phi, L)[t \leftarrow \top]. \end{aligned} \tag{3.1}$$

If $t \notin a(\Phi)$, then Equation 3.1 reduces to $s(\Phi, L) \vee s(\Phi, L)$, which is equivalent to $s(\Phi, L)$, and $s(\Phi, L) \preceq \Phi$ by the hypothesis.

Now we assume the contrary that $t \in a(\Phi)$ and prove all three interface properties.

1. Since, by the hypothesis, $a\left(s(\Phi, L)\right) \subseteq a(\Phi)$, and Equation 3.1 only consists of a disjunction over $s(\Phi, L)$ with t replaced by either $\bot$ or $\top$, also $a\left(s(\Phi, L + \{t\})\right) \subseteq a(\Phi)$.
2. Let α be a valuation that satisfies Φ. By the hypothesis, α also satisfies $s(\Phi, L)$. Since $t \in a(\Phi)$ and thus $t \in a\left(s(\Phi, L)\right)$, valuation α satisfies $s(\Phi, L)$ either with $\alpha(t) = 0$ or $\alpha(t) = 1$. This directly corresponds to the disjunction in Equation 3.1, where t is replaced by $\bot$ or $\top$, which represent either truth value.

3. Let α'' be a minimal valuation that satisfies $s(\Phi, L + \{t\})$. Without loss of generality, assume that α'' satisfies $s(\Phi, L)[t \leftarrow \bot]$ (i.e., the left operand of Equation 3.1). We can then construct a valuation α' that satisfies $s(\Phi, L)$ as

$$\alpha'(r) = \begin{cases} 0 & \text{if } r = t \\ \alpha''(r) & \text{otherwise.} \end{cases}$$

Since α'' is minimal, it is not defined for t. Therefore, $\alpha' \supsetneq \alpha''$. Moreover, α' is minimal with respect to $s(\Phi, L)$, and thus, by the hypothesis, can be extended to a valuation $\alpha \supseteq \alpha'$ that satisfies Φ. Consequently, also $\alpha \supseteq \alpha''$.

□

Slicing removes some atoms from a formula, while preserving a part of its semantics. However, if a formula does not contain those atoms, we cannot remove anything. Indeed, the following lemma and corollary state that we only need to apply slicing to those parts of the formula that have some common atom with the set of sliced atoms.

Lemma 3.3. *If $a(\psi)$ and L are disjoint, then $s(\varphi \wedge \psi, L) \equiv s(\varphi, L) \wedge \psi$.*

Proof. Let $L = \{t_1, \ldots, t_n\}$. Since $s(\varphi \wedge \psi, L) = \exists t_1 \ldots \exists t_n.(\varphi \wedge \psi)$, we can repeatedly apply Lemma 2.1 to pull ψ out of the scope of each quantifier, and eventually obtain

$$(\exists t_1 \ldots \exists t_n.\varphi) \wedge \psi = s(\varphi, L) \wedge \psi.$$

□

Corollary 3.2. *If $a(\varphi)$ and L are disjoint, then $s(\varphi, L) \equiv \varphi$.*

We now state a related lemma, which allows rearranging of the slicing operation even for those parts of the formula which have atoms in the slicing set. We show that the slicing set can be reduced to those atoms that actually occur in each part. However, we have to require that no two parts we want to rearrange have a common atom that is also in the slicing set.

Lemma 3.4. *Let φ, ψ be propositional formulas, and let L be a set of atoms. If $a(\varphi) \cap a(\psi) \cap L = \varnothing$, then*

$$s(\varphi \wedge \psi, L) \equiv s(\varphi, L \cap a(\varphi)) \wedge s(\psi, L \cap a(\psi)).$$

Proof. By induction over the size of L.

For $L = \varnothing$, we have

$$\begin{aligned} s(\varphi \wedge \psi, L) &= \varphi \wedge \psi \\ &\equiv s(\varphi, \varnothing) \wedge s(\psi, \varnothing) \\ &\equiv s(\varphi, L \cap a(\varphi)) \wedge s(\psi, L \cap a(\psi)). \end{aligned}$$

For the inductive step let $L' = L + \{t\}$ such that $a(\varphi) \cap a(\psi) \cap L' = \varnothing$. Then, by the inductive hypothesis,

$$\begin{aligned} s(\varphi \wedge \psi, L') &= \exists t. s(\varphi \wedge \psi, L) \\ &\equiv \exists t. \left[s(\varphi, L \cap a(\varphi)) \wedge s(\psi, L \cap a(\psi)) \right] \\ &\equiv s\left[s(\varphi, L \cap a(\varphi)) \wedge s(\psi, L \cap a(\psi)), t \right] \end{aligned} \tag{3.2}$$

If $t \notin a(\varphi)$, then $L \cap a(\varphi) = L' \cap a(\varphi)$, and a similar result holds for $t \notin a(\psi)$.

If t is in neither set of atoms, then by Corollary 3.2

$$\begin{aligned} (\text{Equation 3.2}) &\equiv s(\varphi, L \cap a(\varphi)) \wedge s(\psi, L \cap a(\varphi)) \\ &\equiv s(\varphi, L' \cap a(\varphi)) \wedge s(\psi, L' \cap a(\varphi)). \end{aligned}$$

On the contrary, without loss of generality let $t \in a(\varphi)$. Then $t \notin a(\psi)$, and we can apply Lemma 3.3 to Equation 3.2:

$$\begin{aligned} (\text{Equation 3.2}) &\equiv s\left[s(\varphi, L \cap a(\varphi)), t \right] \wedge s(\psi, L \cap a(\psi)) \\ &\equiv s(\varphi, L' \cap a(\varphi)) \wedge s(\psi, L \cap a(\psi)) \\ &\equiv s(\varphi, L' \cap a(\varphi)) \wedge s(\psi, L' \cap a(\psi)) \end{aligned}$$

□

Example 3.11. By the previous lemma, we have

$$s(P \wedge Q, \{P, Q, R\}) \equiv s(P, \{P\}) \wedge s(Q, \{Q\}).$$

That is, the atoms P and Q can be moved to the slices of their corresponding formulas, and R can be completely omitted, since it does not occur in the original formula.

Moreover, we have the following counterexample for non-disjoint sets of atoms:

$$\begin{aligned} s\left[(P \vee R) \wedge (Q \vee \neg R), \{R\} \right] &\equiv P \vee Q \\ &\not\equiv s(P \vee R, \{R\}) \wedge s(Q \vee \neg R, \{R\}) \\ &\equiv \top \end{aligned}$$

3.2.5 Properties of feature diagrams

Since our correctness proof will be partially based on the semantics of feature diagrams, we establish some properties of those semantics. In a nutshell, with these properties we can rearrange the semantic formula of a feature diagram into a conjunction of subtree diagrams and another conjunction of the remaining features.

The following lemma states that given a subtree diagram (Definition 3.4), we can use the set of edges of its surrounding feature diagram to express its semantics (instead of only the edges of the subtree).

Lemma 3.5. *Let $D = (F, E, r, \Psi)$ be a feature diagram. Then*

$$\Phi(\downarrow t) \equiv \bigwedge_{t' \in F_{\downarrow t}} \Phi_E(t') \land \Psi_{\downarrow t}$$

for every $t \in F$.

Proof. By expanding the definitions, we obtain

$$\begin{aligned} \Phi(\downarrow t) &\equiv \bigwedge_{t' \in F_{\downarrow t}} \Phi_{E_{\downarrow t'}}(t') \land \Psi_{\downarrow t} \\ &\equiv \bigwedge_{t' \in F_{\downarrow t}} \bigwedge_{(t', t'') \in E_{\downarrow t}} (t'' \Rightarrow t') \land \Psi_{\downarrow t}. \end{aligned} \tag{3.3}$$

Since $(F_{\downarrow t}, E_{\downarrow t})$ is the subtree in (F, E) with root t, the set $E_{\downarrow t}$ consists of all edges from E that have some $t' \in F_{\downarrow t}$ as source. However, the conjunction in Equation 3.3 also only ranges over edges that have their source in $F_{\downarrow t}$. Therefore, we can replace $E_{\downarrow t}$ with E in this context:

$$\begin{aligned} (\text{Equation 3.3}) &\equiv \bigwedge_{t' \in F_{\downarrow t}} \bigwedge_{(t', t'') \in E} (t'' \Rightarrow t') \land \Psi_{\downarrow t} \\ &\equiv \bigwedge_{t' \in F_{\downarrow t}} \Phi_E(t') \land \Psi_{\downarrow t} \end{aligned}$$

□

Subtree diagrams are a direct part of their surrounding diagram, and do not abstract away or add information. Therefore, intuitively we can express the semantics of the full diagram as a conjunction of subtree diagram semantics, plus the semantics of the features that are not included in any of those subtree diagrams. With the following lemma, we show that such a semantic decomposition is valid for arbitrary choices of subtree diagrams.

Lemma 3.6. *Let $D = (F, E, r, \Psi)$ be a feature diagram, and $S \subseteq F$ be a set of disjoint features. Then*

$$\Phi(D) \equiv \bigwedge_{t \in S} \Phi(\downarrow t) \land \bigwedge_{t \in F \setminus \widehat{S}} \Phi_E(t) \land \Psi_G, \tag{3.4}$$

where

$$\widehat{S} = \bigcup_{t \in S} F_{\downarrow t}$$

and

$$\Psi_G = \Psi \setminus \bigcup_{t \in S} \Psi_{\downarrow t}.$$

Proof. We rearrange the right-hand side of the equivalence until it matches the definition of $\Phi(D)$.

We first expand the $\Phi(\downarrow t)$ terms using Lemma 3.5, that is, we express the $\downarrow t$ semantics by means of E.

$$\bigwedge_{t\in S}\Phi(\downarrow t)\equiv\bigwedge_{t\in S}\bigwedge_{t'\in F_{\downarrow t}}\Phi_E(t')\wedge\Psi_{\downarrow t} \tag{3.5}$$

The conjunction in Equation 3.5 ranges over $t'\in F_{\downarrow t}$ with $t\in S$, which directly corresponds to the definition of $\widehat{S}$. Therefore, we arrive at the following equivalence.

$$(\text{Equation 3.5})\equiv\bigwedge_{t'\in\widehat{S}}\Phi_E(t')\wedge\bigwedge_{t\in S}\Psi_{\downarrow t} \tag{3.6}$$

We now substitute Equation 3.6 into the right-hand side of Equation 3.4.

$$\bigwedge_{t\in S}\Phi(\downarrow t)\wedge\bigwedge_{t\in F\setminus\widehat{S}}\Phi_E(t)\wedge\Psi_G\equiv\bigwedge_{t'\in\widehat{S}}\Phi_E(t')\wedge\bigwedge_{t\in S}\Psi_{\downarrow t}\wedge\bigwedge_{t\in F\setminus\widehat{S}}\Phi_E(t)\wedge\Psi_G \tag{3.7}$$

Since $\widehat{S}\subseteq F$ and consequently $\widehat{S}\cup(F\setminus\widehat{S})=F$, we can join both conjunctions of Φ_E terms to a single conjunction that ranges over F. Similarly, $\Psi_G\wedge\bigwedge_{t\in S}\Psi_{\downarrow t}\equiv\Psi$. We apply both equivalences and finally obtain the definition of $\Phi(D)$.

$$\begin{aligned}(\text{Equation 3.7})&\equiv\bigwedge_{t'\in F}\Phi_E(t')\wedge\bigwedge_{t\in S}\Psi_{\downarrow t}\wedge\Psi_G\\&\equiv\bigwedge_{t'\in F}\Phi_E(t')\wedge\Psi\\&=\Phi(D)\end{aligned}$$

□

Lemma 3.7. *For every set S of disjoint features, the union of local subtree features*

$$\bigcup_{t\in S}l(\downarrow t).$$

is disjoint.

Proof. Let $t,t'\in S$. Since features in S are disjoint, the sets $F_{\downarrow t}$ and $F_{\downarrow t'}$ are disjoint by definition. Furthermore, $l(t)\subseteq F_{\downarrow t}$ for every feature t by definition. Therefore, also $l(t)$ and $l(t')$ are disjoint. □

3.2.6 Correctness

Finally, we show that the subtree decomposition is correct, that is, every subtree decomposition is a selectable decomposition. We split the proof into two theorems: The first theorem states that the composition of an interface of a feature diagram with a set of subtree diagrams is still an interface. With the second theorem, we show that every atom-based selection over a set of subtree diagrams from a subtree decomposition is semantically complete. Therefore, the latter theorem establishes the connection to selectable decompositions from Section 3.1.

Informally, the following theorem states that reinstantiating a concrete subtree in an interface of a feature diagram results in another interface, but with an extended set of atoms. In other words, we can selectively reconcretize parts of a formula we have sliced out before, and obtain another, more concrete abstraction of the formula. As a consequence, we can use the result for sound and complete reasoning with respect to all of its atoms.

Theorem 3.3. *Let $D = (F, E, r, \Psi)$ be a feature diagram, and $D/S = (\Gamma, \Delta)$ be a subtree decomposition. Then $\Gamma \cup \Delta' \preceq \Phi(D)$ for every $\Delta' \subseteq \Delta$.*

Proof. By definition, $\Gamma = s(\Phi(D), L)$, where $L = \bigcup_{t \in S} l(\downarrow t)$. With Lemma 3.6, we can reformulate $\Phi(D)$, such that $\Phi(D) \equiv \bigwedge_{t \in S} \Phi(\downarrow t) \wedge \bigwedge_{t \in F \setminus \widehat{S}} \Phi_E(t) \wedge \Psi_G$.

The sets of local features are disjoint (Lemma 3.7), and occur only in formulas of their subtree diagram $\downarrow t$. Therefore, we can apply Lemma 3.4 and Lemma 3.3 to decompose the application of the slice operation:

$$\begin{aligned} s(\Phi(D), L) &\equiv s\left(\bigwedge_{t \in S} \Phi(\downarrow t) \wedge \bigwedge_{t \in F \setminus \widehat{S}} \Phi_E(t) \wedge \Psi_G, L\right) \\ &\equiv \bigwedge_{t \in S} s(\Phi(\downarrow t), l(\downarrow t)) \wedge \bigwedge_{t \in F \setminus \widehat{S}} \Phi_E(t) \wedge \Psi_G \end{aligned} \tag{3.8}$$

Recall that by Lemma 3.2, every slice is an interface of its original formula. Now consider the conjunction of $\Gamma \cup \Delta'$. Since Δ' consists of subtree diagrams induced by S, for every $\Phi(\downarrow t) \in \Delta'$ there is a corresponding interface $s(\Phi(\downarrow t), l(\downarrow t))$ in Γ. By Lemma 3.1, each interface is absorbed by its original formula. Accordingly, in the set $\Gamma \cup \Delta'$, each subtree from Δ' absorbs its interface from Γ; formally

$$\Phi(\downarrow t) \wedge s(\Phi(\downarrow t), l(\downarrow t)) \equiv \Phi(\downarrow t).$$

Therefore, we can formulate the following equivalences. Let $S' \subseteq S$ be a set of features such that $\Delta' = \{\, \Phi(\downarrow t) \mid t \in S' \,\}$. We substitute Equation 3.8 into $\Gamma \cup \Delta'$ and let each subtree absorb its interface. Consequently, the conjunction of slice operations then merely ranges over $S \setminus S'$:

$$\begin{aligned} \Gamma \cup \Delta' &\equiv s(\Phi(D), L) \wedge \bigwedge_{t \in S'} \Phi(\downarrow t) \\ &\equiv \bigwedge_{t \in S} s(\Phi(\downarrow t), l(\downarrow t)) \wedge \bigwedge_{t \in F \setminus \widehat{S}} \Phi_E(t) \wedge \Psi_G \wedge \bigwedge_{t \in S'} \Phi(\downarrow t) \\ &\equiv \bigwedge_{t \in S \setminus S'} s(\Phi(\downarrow t), l(\downarrow t)) \wedge \bigwedge_{t \in F \setminus \widehat{S}} \Phi_E(t) \wedge \Psi_G \wedge \bigwedge_{t \in S'} \Phi(\downarrow t) \end{aligned} \tag{3.9}$$

Now we apply Lemma 3.4 in reverse direction and recompose the individual slice operations into a single operation with a subset L' of L. The set L' comprises all local features of the remaining subtree diagram interfaces, thus $L' = \{\, l(\downarrow t) \mid t \in S \setminus S' \,\}$. We can include the term $\bigwedge_{t\in S'} \Phi(\downarrow t)$ in the slice operation, since the sets $S \setminus S'$ and S' are disjoint, and subtree diagrams do not share any local features (Lemma 3.7). We arrive at the following equivalence:

$$(\text{Equation 3.9}) \equiv s\left(\bigwedge_{t\in S\setminus S'} \Phi(\downarrow t) \wedge \bigwedge_{t\in F\setminus \widehat{S}} \Phi_E(t) \wedge \Psi_G \wedge \bigwedge_{t\in S'} \Phi(\downarrow t), L'\right) \tag{3.10}$$

We now rearrange the first operand of the slice operation in Equation 3.10 so that we are able to apply Lemma 3.6 in reverse direction and obtain the equivalence $\Gamma \cup \Delta' \equiv s(\Phi(D), L')$. To this end, we expand the Ψ_G term to its definition, and partially apply Lemma 3.5 to the rightmost conjunction of $\Phi(\downarrow t)$ terms.

$$\begin{aligned}
&\bigwedge_{t\in S\setminus S'} \Phi(\downarrow t) \wedge \bigwedge_{t\in F\setminus \widehat{S}} \Phi_E(t) \wedge \Psi_G \wedge \bigwedge_{t\in S'} \Phi(\downarrow t)\\
\equiv &\bigwedge_{t\in S\setminus S'} \Phi(\downarrow t) \wedge \bigwedge_{t\in F\setminus \widehat{S}} \Phi_E(t) \wedge \Psi \setminus \bigcup_{t\in S} \Psi_{\downarrow t} \wedge \bigwedge_{t\in S'} \bigwedge_{t'\in F_{\downarrow t}} \Phi_E(t') \wedge \Psi_{\downarrow t}
\end{aligned} \tag{3.11}$$

We remove the $\Phi_{\downarrow t}$ terms from the rightmost conjunction and adapt the range of the difference on Ψ accordingly[6]. Similarly, we join both conjunctions of Φ_E terms. Therefore, let $\widehat{S'} = \bigcup_{t\in S'} F_{\downarrow t}$.

$$\begin{aligned}
(\text{Equation 3.11}) &\equiv \bigwedge_{t\in S\setminus S'} \Phi(\downarrow t) \wedge \bigwedge_{t\in F\setminus \widehat{S}} \Phi_E(t) \wedge \Psi \setminus \bigcup_{t\in S\setminus S'} \Psi_{\downarrow t} \wedge \bigwedge_{t\in S'} \bigwedge_{t'\in F_{\downarrow t}} \Phi_E(t')\\
&\equiv \bigwedge_{t\in S\setminus S'} \Phi(\downarrow t) \wedge \bigwedge_{t\in F\setminus \widehat{S}} \Phi_E(t) \wedge \Psi \setminus \bigcup_{t\in S\setminus S'} \Psi_{\downarrow t} \wedge \bigwedge_{t'\in \widehat{S'}} \Phi_E(t')\\
&\equiv \bigwedge_{t\in S\setminus S'} \Phi(\downarrow t) \wedge \bigwedge_{t\in (F\setminus \widehat{S})\cup \widehat{S'}} \Phi_E(t) \wedge \Psi \setminus \bigcup_{t\in S\setminus S'} \Psi_{\downarrow t}
\end{aligned} \tag{3.12}$$

To apply Lemma 3.6, we still need to prove that $(F \setminus \widehat{S}) \cup \widehat{S'} = \bigcup_{t\in S\setminus S'} F_{\downarrow t}$.

$$\begin{aligned}
(F \setminus \widehat{S}) \cup \widehat{S'} &= (F \cup \widehat{S'}) \setminus (\widehat{S} \setminus \widehat{S'})\\
&= F \setminus (\widehat{S} \setminus \widehat{S'})\\
&= F \setminus \left(\bigcup_{t\in S} F_{\downarrow t} \setminus \bigcup_{t'\in S'} F_{\downarrow t'}\right)\\
&= F \setminus \bigcup_{t\in S\setminus S'} F_{\downarrow t}
\end{aligned} \tag{3.13}$$

Using Equation 3.13, we apply Lemma 3.6 to Equation 3.12. Therefore, Equation 3.12 is equivalent to $\Phi(D)$. Consequently, $\Gamma \cup \Delta' \equiv s(\Phi(D), L')$. Finally, using Lemma 3.2 we immediately obtain that $\Gamma \cup \Delta' \preceq \Phi(D)$. □

[6]For ease of exposition, we abuse the notation of set operations and logical connectives here. More precisely, we use sets of formulas and the conjunction of their elements interchangeably.

With the previous theorem, we have shown that we can selectively reconcretize the interface produced by a subtree decomposition. However, we still need to show that an atom-based selection on the set of subtree diagrams indeed composes a sound and complete interface for arbitrary queries. We show this with the following, final theorem.

Theorem 3.4. *Every subtree decomposition D/S of a feature diagram D is a selectable decomposition of $\Phi(D)$.*

Proof. Let $D/S = (\Gamma, \Delta)$ with $D = (F, E, r, \Psi)$, and φ be a propositional formula such that $a(\varphi) \subseteq a(\Phi(D))$. Moreover, let $\Delta' = \Delta|_{a(\varphi)}$. Since $\Delta' \subseteq \Delta$, the set $\Gamma \cup \Delta'$ is an interface of $\Phi(D)$ by Theorem 3.3. We need to show that $a(\varphi) \subseteq a(\Gamma \cup \Delta')$, then the claim follows with Theorem 3.2.

We prove that $a(\varphi) \subseteq a(\Gamma \cup \Delta')$ by contradiction. Assume that the statement does not hold, then there exists an atom $r \in a(\varphi)$ that is present neither in Γ nor in Δ'. If $r \notin a(\Gamma)$, then, by $a(\varphi) \subseteq a(\Phi(D))$ and the construction of D/S, atom r is a local feature of some subtree diagram $\Phi(\downarrow t) \in \Delta$. Since $l(\downarrow t) \subseteq F_{\downarrow t}$, also $r \in F_{\downarrow t}$. Consequently, $a(\varphi) \cap a(\Phi(\downarrow t)) \neq \varnothing$, and thus $\Phi(\downarrow t) \in \Delta'$. However, then $r \in a(\Delta')$, which is a contradiction. □

3.3 Summary

In this chapter, we have shown how to reason with decomposed feature diagrams, which enables faster SAT solving due to smaller formulas. To this end, we have first introduced a generic decomposition of propositional formulas, and corresponding strategies that recompose the relevant parts for each query, using a simple atom-based selection criterion.

Subsequently, we have defined a generic notion of feature diagrams with semantics in propositional logic. Based on this definition, we have introduced a decomposition and slicing procedure for a feature diagrams, which is guided by a set of its subtrees. Provided that we are able to choose subtrees according to implementation artifacts, we can abstract away some features that are local to such an artifact. During a family-based analysis, we are then able to selectively reconcretize parts of the abstract diagram, as needed by the particular implementation artifact. To prove correctness, we have adapted the concept of feature model interfaces to propositional formulas. We have shown that such interfaces can be used for sound and complete reasoning under a restricted set of atoms, and that slicing indeed produces such interfaces.

Finally, we have shown that the atom-based selection on our feature diagram decomposition produces a correct subset of formulas for our generic reasoning strategies. Therefore, the decomposition we have defined is correct. For now, it remains an open question how to choose an optimal set of subtrees. Possible influencing factors are the implementation articfacts, or the type of the family-based analysis.

4. Evaluation

In this chapter, we evaluate our approach for reasoning with feature model interfaces as presented in Chapter 3. In particular, we compare solver execution times of the FULL-MODEL strategy, which always uses the full model for reasoning, against our REDUCED-MODEL strategy, which composes a specific model for each solver query. We show that under certain restrictions regarding feature model decompositions and queries, REDUCED-MODEL outperforms FULL-MODEL. Consequently, we provide a threshold measure $m(\Delta)$ for the ADAPTIVE-MODEL strategy that is based on the number of CNF clauses in selected submodels.

Comparing overall execution times of a family-based analysis that uses either the FULL-MODEL or REDUCED-MODEL strategy for reasoning would yield the most precise evaluation results. For such an experiment, we need

1. a product line with source code available,
2. its feature model in feature diagram form,
3. a decomposition of the feature diagram that corresponds to the implementation structure, and
4. a family-based analysis implementation and setup for the product line.

To assess the full potential of our approach, we use the Linux kernel as the largest publicly available product line. For version 2.6.33.3, which has over 10,000 features[1], there exists a family-based type checking setup[2] for the TypeChef analysis framework[3]. Unfortunately, to the best of our knowledge there is no comprehensive feature diagram for any version of the Linux kernel, and we also do not have access to another product line with similar size. Therefore, we split our evaluation into two stages to approximate an ideal evaluation setup. In Section 4.2, we analyze

[1]in the boolean approximation of the original feature model
[2]https://github.com/ckaestne/TypeChef-LinuxAnalysis
[3]https://github.com/ckaestne/TypeChef

characteristics of solver queries that arise during typechecking of the Linux kernel. We use the results in Section 4.3 to generate realistic queries for another feature model with similar size, which is available as a feature diagram and for which we have decomposition into submodels.

4.1 Goals

We concretize the goals of this evaluation in form of the following research questions. As explained above, the first stage of our evaluation consists of analyzing practical solver queries in a family-based analysis, which we formulate in research question RQ 1.

RQ 1 What are the characteristics of SAT solver queries in the analysis of a real-world software system?

> **RQ 1.1** What is the fraction of solving satisfiability in overall analysis computation time?
>
> **RQ 1.2** How local are queries?

Since a family-based analysis not only performs SAT solving but also other computations, we determine the potential performance gain in the overall analysis runtime when using our REDUCED-MODEL approach with RQ 1.1. Moreover, in Chapter 3 we hypothesized that queries typically only contain features from few submodels. Since we anticipate some overhead caused by recomposition of the selected submodels, we are interested if such query locality actually exists. To this end, we formulate RQ 1.2.

Using the results from RQ 1, we compare reasoning times of the FULL-MODEL and REDUCED-MODEL strategies. As the REDUCED-MODEL strategy might not outperform FULL-MODEL in all cases, we are interested in properties of queries for which it does. For that purpose, we specify research question RQ 2 as follows.

RQ 2 For which kinds of queries does reasoning with interfaces outperform reasoning with the full model?

> **RQ 2.1** How large is the composition overhead?
>
> **RQ 2.2** What is the impact of locality in queries?
>
> **RQ 2.3** Is there a threshold measure $m(\Delta')$ for the composition overhead? If so, is the threshold value τ uniform accross different feature models?

As already mentioned, the overhead of recomposing models might render the REDUCED-MODEL approach inefficient in some cases. To analyze this overhead, we specify RQ 2.1. Since this overhead might be dependent on the degree of locality of a query, we analyze its impact with RQ 2.2. In Chapter 3, we have proposed a generic threshold measure to select between FULL-MODEL and REDUCED-MODEL for each query, in order to combine advantages of both strategies. With RQ 2.3, we investigate query properties as concrete candidates for such a measure. Moreover, if such a measure exists we want to know if its threshold value τ can be transferred to a different model, or if it must be determined individually.

4.2 Solver queries in family-based analysis

In this section, we answer RQ 1 by type checking a subset of the Linux kernel with TypeChef. We record all queries issued against the feature model, and their runtimes.

4.2.1 Experimental setup

We use the evaluation setup for Linux 2.6.33.3 (x86) from Liebig et al. [2013] with some modifications. Linux employs the C preprocessor in combination with the Kconfig build system to implement variability. The feature model of the given version comprises over 10,000 features. We parse and type check each kernel source file in a seperate Java Virtual Machine (JVM) with the most recent TypeChef release (version 0.3.7[4]).

As an optimization, TypeChef reasons with both *binary decision diagrams* (BDDs) [Bryant, 1986] and a SAT solver. Presence conditions are converted to BDDs and mostly checked only against each other without incorporating the feature model. Only when final results (i.e., type errors) are about to be reported, TypeChef invokes the SAT solver with a condition and the feature model to prune false positives. Moreover, results of recurring queries are cached.

We instrument TypeChef to record the formulas of all queries issued against the solver, since only those queries involve the feature model. We use `System.nanoTime()`[5] to measure solver computation time for each query. However, we do not record queries that are retrieved from the cache. In addition, we measure times of the parsing and type checking phases for each file.

Since type checking all kernel files would take several weeks, we only measure a random subset of files. For the same reason, we do not take multiple samples of each time measurement to reduce noise caused by garbage collection or just-in-time optimizations [Georges et al., 2007]. Therefore, the results should be considered as an estimate.

We perform all measurements in a 64-bit Debian 8.6[6] virtual machine with 12GB of memory and four cores. We run a type checking task with 2.5GB JVM heap space on each core. The host system is equipped with an Intel Core i7-4800MQ with four cores and 16GB of memory.

4.2.2 Results

We have randomly chosen and type checked 1,363 out of 7,760 files from the x86 subset of the kernel, which issued a total of 173,845 queries against the feature model. Out of those queries, 61 percent were syntactically distinct. On average, the solver took 90 milliseconds to process a query. Per file, this amounts to 12 seconds of solver computation time for 128 queries, while the overall file processing time in TypeChef is 24 minutes. Thus, 0.7 percent of the overall analysis time is consumed

[4]https://github.com/ckaestne/TypeChef/tree/v0.3.7
[5]http://docs.oracle.com/javase/8/docs/api/java/lang/System.html#nanoTime--
[6]https://www.debian.org

by SAT solving. Considering only the type checking phase, which takes 19 seconds on average and issues most of the queries, approximately 60 percent of the time are used for reasoning.

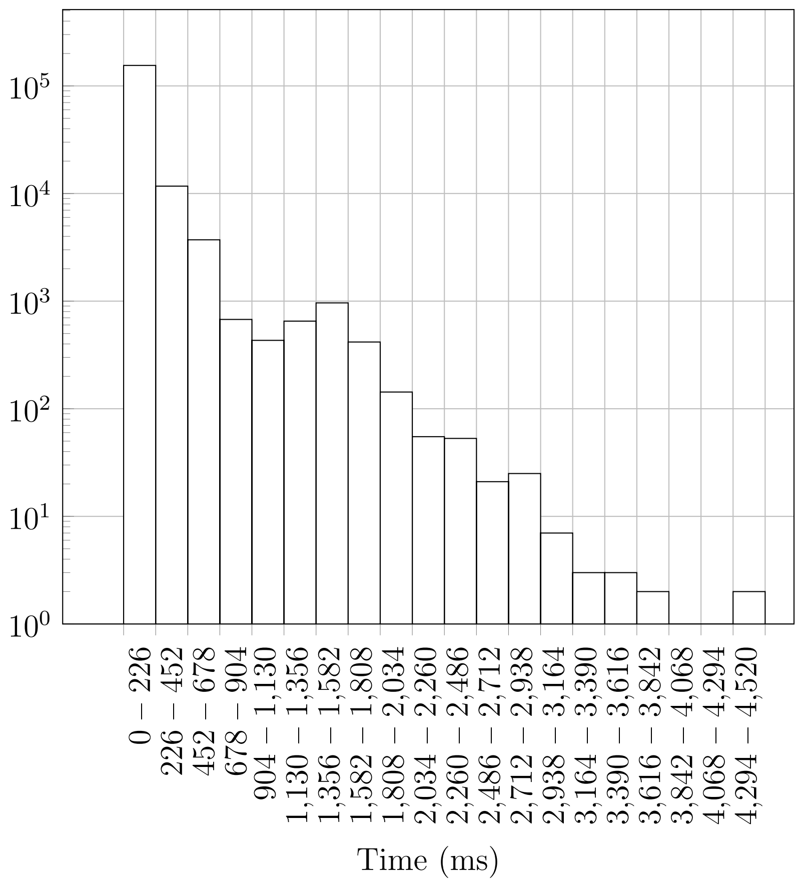


Figure 4.1: Distribution of solver times per query

In Figure 4.1, we show the distribution of solver times on a logarithmic scale. We see that more than half of the queries take less than 226 milliseconds. The median lies at 20 milliseconds. Nevertheless, there are also queries that take over 4 seconds to compute.

To assess the locality of queries (RQ 1.2), we analyze the number of literals and the number of distinct features in each query. Literals are atomic propositions or their negations, thus the number of distinct features is always smaller or equal to the number of literals. Since we do not have a feature diagram decomposition for Linux, we cannot determine the actual locality (i.e., number of submodels involved in the query). Instead, we examine the distributions of literals and distinct features to obtain an estimate.

We show the distribution of literals per query in Figure 4.2. Almost half of the queries contain only up to 28 literals, but there are also queries with up to 529 literals, with a median of 26 literals. The distribution is roughly exponential. By comparison, we show the distribution of distinct features per query in Figure 4.3. The diamond symbol represents the average at 8.8 features, which is close to the median at 9 features. There are three outlier queries at 18, 20, and 27 features.

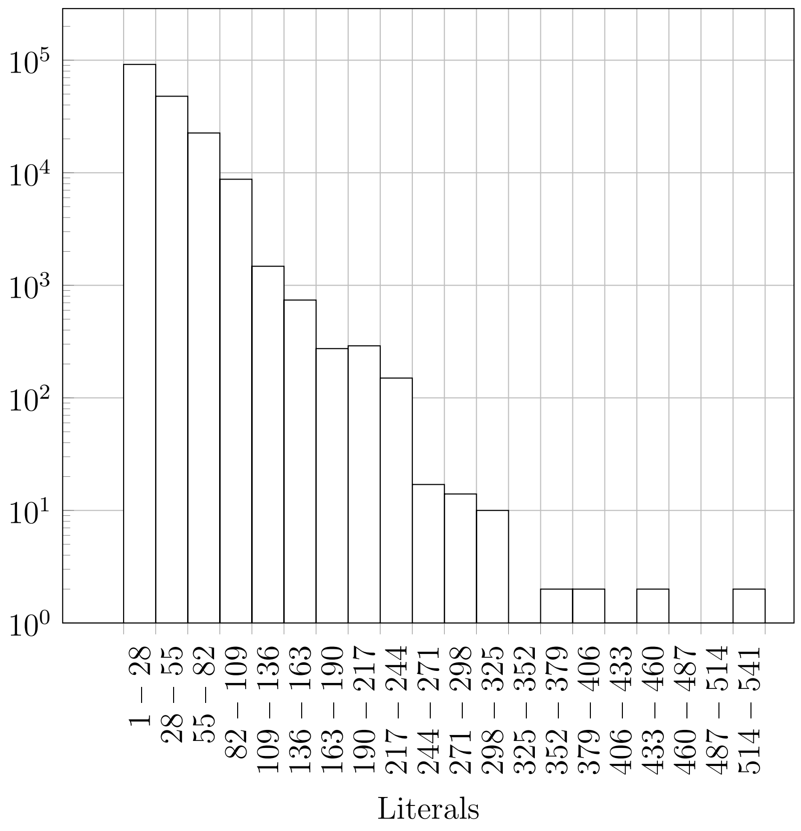


Figure 4.2: Distribution of literals per query

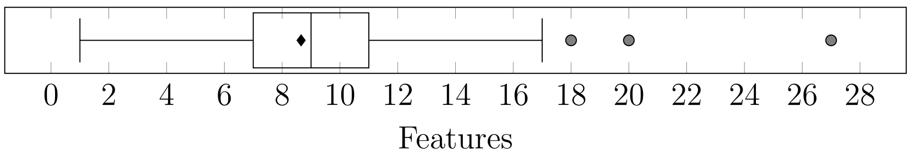


Figure 4.3: Distribution of distinct features per query

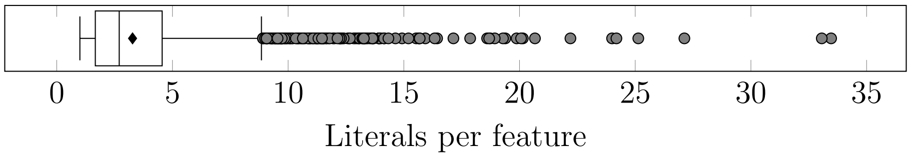


Figure 4.4: Distribution of literal to feature ratios

In Figure 4.4, we show the distribution of literals per feature for each query. The median is at a ratio of 2.7, and slightly differs from the average at 3.3. Most queries have a ratio of less than 8.8. Because of the large data set, there are nearly 5,000 outliers, with a maximum ratio of 33.5.

4.2.3 Discussion

Our time measurements reveal that in our particular setup, reasoning is by far not the dominant cost when considering the overall analysis time. Rather, variability-aware parsing accounts for over 99 percent of the computation time. We suspect that this is due to complexity of parsing C preprocessor annotations together with the C source code. Since in general, such annotations and the C language constructs do not align (there may be so-called *undisciplined* annotations [Liebig et al., 2011]), a variability-aware C parser must split the analysis on a per-token basis, which causes computational overhead compared to a disciplined variability mechanism. We refer the reader to the work of Kästner et al. [2011] for further details. However, TypeChef allows to save and reuse the parsing result for different family-based analyses of the same file.

If we assume a disciplined variability mechanism such as feature-oriented programming, parsing becomes simpler, as variability only requires additional language constructs in the grammar. Therefore, we may relate the reasoning time to the actual analysis time alone (i.e., the type checking phase in our case). Under this assumption, we can potentially improve over half of the analysis time with our approach. However, for precise results we need to perform another evaluation using such those variability mechanisms, which we leave to future work.

Contrary to our expectations, even for such a large model with over 10,000 features a typical query takes only 20 milliseconds to solve, and the majority requires under 300 milliseconds, as shown in Figure 4.1. Nevertheless, there are also more complicated queries that take several orders of magnitude longer.

In summary, we answer RQ 1.1 as follows. In general, our approach may affect over 60 percent of the pure analysis time. Although we may save up to 11 seconds of reasoning time per file in our experiments, our approach does not improve the analysis of product lines that are based on the C preprocessor, since parsing is the dominant cost and should be optimized first. As an additional result, we observed that at least 39 percent of the queries are potentially redundant, which allows for further optimization, for example through caching across several TypeChef runs.

Regarding query locality (RQ 1.2), we conclude from the distribution in Figure 4.3 that a typical query contains 9 distinct features. Thus, for combinatorial reasons these features belong to at most 9 different submodels. Unfortunately, without a decomposed feature model for the considered Linux version (e.g., with kernel modules as components), we cannot determine sharper bounds.

Although queries may contain over 500 literals (see Figure 4.2), they have a maximum of 27 distinct features out of over 10,000 available features. However, the average literal to feature ratio is lower (3.3), as shown in Figure 4.4. Thus, for queries that are large in terms of literals, we identify a higher locality than for queries with fewer literals.

4.3 Performance of reasoning with interfaces

In the previous section, we have analyzed queries from a practical family-based analysis. To answer RQ 2, in this section we compare the FULL-MODEL and REDUCED-MODEL strategies using the structure of the recorded queries together with four evolutionary snapshots of a feature diagram from the automotive domain.

4.3.1 Experimental setup

We have developed a prototypical benchmark implementation of the REDUCED-MODEL strategy on top of the FeatureIDE framework for feature-oriented software development [Thüm et al., 2014b]. To realize the FULL-MODEL strategy, we use existing capabilities of FeatureIDE for reasoning with feature diagrams. From the queries recorded in Section 4.2, we generate a set of benchmark queries by replacing all Linux features with random features from the automotive snapshots. Then, we run each benchmark query using both the FULL-MODEL and the REDUCED-MODEL implementation, and measure computation times for comparison.

Implementation

Our implementation is based on version 3.0.1 of FeatureIDE[7]. FeatureIDE provides procedures for loading and manipulating feature diagrams, as well as their conversion to propositional formulas and subsequent reasoning. This version uses the SAT solver library Sat4j[8] 2.3.5.

To initialize our implementation, we load a feature diagram (i.e., the full model) as well as a set of extracted subdiagrams and their corresponding precomputed interfaces from XML files. For the FULL-MODEL strategy, we simply use the loaded diagram itself. By contrast, for the REDUCED-MODEL strategy we need a selectable composition (Γ, Δ), where Γ represents a reduced feature model that we can selectively refine with submodels from Δ (see Algorithm 3.2 on page 26). To construct Γ, we modify in the full model in two steps. First, we insert the loaded interface trees in place of their corresponding concrete subtrees. Second, we remove all constraints which contain only features that are local to some replaced subdiagram (see Definition 3.6 on page 32). The set Δ simply consists of the loaded subdiagrams. To prepare the models and submodels for reasoning with Sat4j, we convert them to their propositional formulas in conjunctive normal form. Finally, we create two SAT solver instances, one configured with the full model, and another one configured with the reduced model Γ.

We implement the REDUCED-MODEL strategy as follows to process a query φ. As the sliced model Γ of a subtree decomposition is an interface of the original model, we do not need to include any concrete submodels if the query only consists of features from Γ. We use this as a slight optimization of the initial definition of REDUCED-MODEL that is independent of slicing. For each feature in φ that is not present in Γ, we select the subdiagram CNFs from Δ that contain this feature. Then, we concatenate all selected CNFs with the CNF of $\neg\varphi$. Finally, we invoke the reduced model solver with the resulting formula.

[7]https://github.com/FeatureIDE/FeatureIDE/releases/tag/v3.0.1
[8]http://www.sat4j.org/

Feature models

For this evaluation, we use the *Automotive_02* feature diagram from the FeatureIDE example models[9]. This model originates from practical development in the automotive industry [Schröter et al., 2016]. It has been created by integrating several smaller models into a comprehensive model. However, the submodels are still available as separate files. Furthermore, there is also a precomputed interface for each submodel, which omits precisely all local features. Therefore, we can construct a selectable decomposition as previously described.

Snapshot	Features	Constraints	Clauses	Submodels
1	14,010	666	237,706	44
2	17,742	914	342,935	45
3	18,434	1,300	347,557	46
4	18,616	1,369	350,287	46

Table 4.1: Statistics of the automotive model snapshots

The automotive model is available in four different snapshots, which capture its evolution over time. We conduct independent measurements on each snapshot. In Table 4.1, we show the statistics of each snapshot. The snapshots have between 14,010 and 18,616 features, and between 44 and 46 submodels. Thus, even the smallest snapshot has more features than the Linux model we have used in Section 4.2. We also specify the number of CNF clauses as computed by our implementation. Additionally, we show the distribution of the number of features in each submodel in Figure 4.5 for each snapshot.

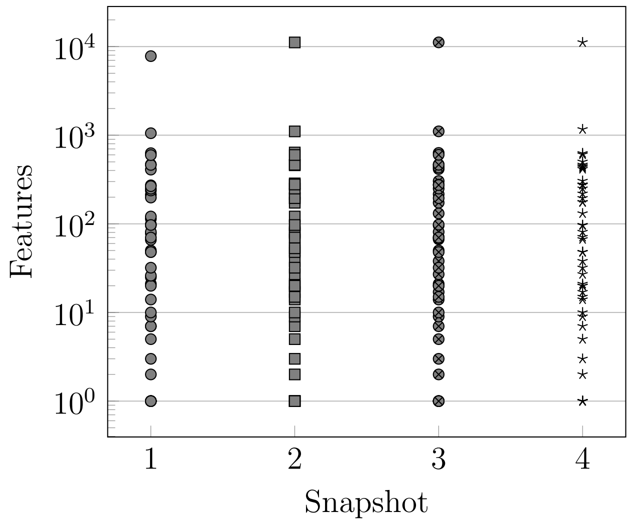


Figure 4.5: Features in submodels

Query generation

We use the queries recorded from the Linux evaluation in Section 4.2 as templates to generate semi-random queries for the automotive snapshots. For each distinct

[9]https://github.com/FeatureIDE/FeatureIDE/tree/develop/plugins/de.ovgu.featureide.examples/featureide_examples/FeatureModels

feature in the original query, we randomly select a feature from the snapshot model. As a result, we obtain a one-to-one a mapping from Linux features to automotive features on a per-query basis. Finally, we replace the Linux features in all literals of the query using this mapping.

For example, in the query

$$(\mathit{CONFIG\_X86\_64} \wedge \neg \mathit{CONFIG\_PARAVIRT} \wedge \neg \mathit{CONFIG\_X86\_32}) \vee \mathit{CONFIG\_PARAVIRT}$$

we have three different Linux features. To create a query for our evaluation from this query, we randomly choose a feature from the automotive model for each of these features (e.g., *F_4AA1EA10E8E2*, *F_8B531C609895*, and *F_D86F03821B9B*), and substitute it into the formula. Thus, we obtain the formula

$$(\mathit{F\_4AA1EA10E8E2} \wedge \neg \mathit{F\_8B531C609895} \wedge \neg \mathit{F\_D86F03821B9B}) \vee \mathit{F\_8B531C609895},$$

which we then check for satisfiability against the particular automotive model. We use a single set of such generated queries for both reasoning strategies.

With this method, we generate queries that are more realistic compared to completely random formulas. The substition operation preserves properties such as formula structure, amount of literals and distinct features, and satisfiability of the formula itself (without considering the feature model). In addition, randomly generated formulas are often harder to solve than formulas that arise from practical applications [Liang et al., 2015].

Measurements

Performance measurements on the Java Virtual Machine are nondeterministic due to garbage collection and just-in-time optimizations [Georges et al., 2007]. Measuring execution time of a code fragment multiple times could yield results that differ by orders of magnitude, for example because the code was not optimized at the first run, or garbage collection was accidently triggered during measurement. If such perturbations are not detected and eliminated, the resulting data could lead to wrong conclusions.

To improve the precision of our measurements, we use *ScalaMeter*[10] version 0.7, which is a performance measuring framework for the JVM. ScalaMeter implements several techniques that reduce the amount of misleading measurements. To measure each reasoning strategy, we use the following ScalaMeter configuration. For a set of generated queries, we consecutively run nine JVM invokations. Before the actual measurements, we repeatedly run the strategy with a random query until execution time stabilizes (so called *warm-up* runs). This reduces the effects of class loading and just-in-time compilation. Then, we take four measurements of the execution time of each query. If there is an outlier in these measurements (e.g., due to garbage collection) we replace it with another measurement. Thus, in total we measure each query 36 times. We use the ScalaMeter standard parameters for steady-state detection of the warm-up phase and outlier elimination.

[10]https://scalameter.github.io/

Unless noted otherwise, for the FULL-MODEL strategy we measure how long the initialized solver takes to decide the satisfiability of each query[11]. When measuring the REDUCED-MODEL strategy, we additionally include the time to select and concatenate the relevant submodel CNFs.

We conduct all measurements on a Windows 7 system with an Intel Core i5-3570 and 8GB of memory, and run all JVM invokations with 4GB of heap space.

4.3.2 Optimization

In early experiments, we found that our REDUCED-MODEL strategy was slower than the FULL-MODEL strategy by orders of magnitude. We did not expect such an adverse effect, since the REDUCED-MODEL solver processes smaller formulas on average. Therefore, we investigated the cause and optimized the Sat4j implementation based on our findings.

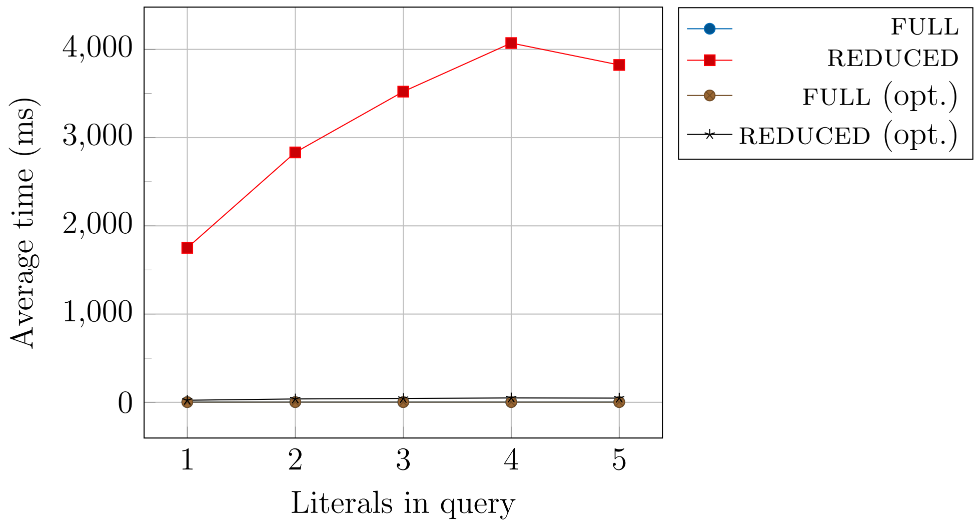
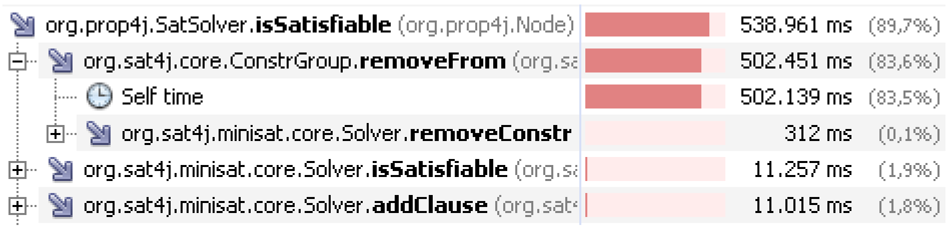


Figure 4.6: Comparison of optimized and unoptimized solver

In Figure 4.6, we compare optimized and unoptimized versions of both strategies. For each data point, we took the average of 100 randomly generated queries. We see that even for a single literal in the query, the unoptimized version of REDUCED-MODEL is over 700 times slower than the unoptimized FULL-MODEL.



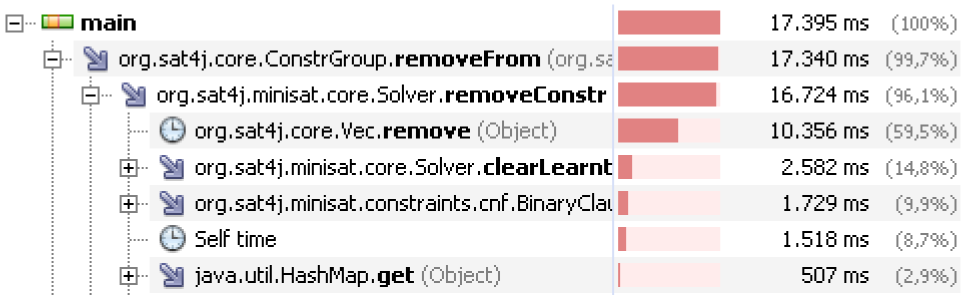
```
org.prop4j.SatSolver.isSatisfiable (org.prop4j.Node)      538.961 ms  (89,7%)
  org.sat4j.core.ConstrGroup.removeFrom (org.sa           502.451 ms  (83,6%)
    Self time                                             502.139 ms  (83,5%)
    org.sat4j.minisat.core.Solver.removeConstr                312 ms   (0,1%)
  org.sat4j.minisat.core.Solver.isSatisfiable (org.sa      11.257 ms   (1,9%)
  org.sat4j.minisat.core.Solver.addClause (org.sat         11.015 ms   (1,8%)
```

Figure 4.7: Profiling session of initial benchmark implementation

To track down the issue, we profiled the REDUCED-MODEL implementation with the NetBeans IDE[12] profiler, version 8.1. We ran the implementation in a loop with

[11]i.e., method `org.prop4j.SatSolver.isSatisfiable(Node)`
[12]https://netbeans.org/

random queries, and measured execution times of all methods to discover possible bottlenecks. In Figure 4.7, we show a screenshot of the profiling session. The `removeFrom` method of Sat4j class `ConstrGroup` causes over 80 percent of the total computation time.



```
main                                               17.395 ms  (100%)
  org.sat4j.core.ConstrGroup.removeFrom (org.sa    17.340 ms  (99,7%)
    org.sat4j.minisat.core.Solver.removeConstr     16.724 ms  (96,1%)
      org.sat4j.core.Vec.remove (Object)           10.356 ms  (59,5%)
      org.sat4j.minisat.core.Solver.clearLearnt     2.582 ms  (14,8%)
      org.sat4j.minisat.constraints.cnf.BinaryClau  1.729 ms  (9,9%)
      Self time                                     1.518 ms  (8,7%)
      java.util.HashMap.get (Object)                  507 ms  (2,9%)
```

Figure 4.8: Detailed profiling of `removeFrom` method

We suspected that the bottleneck was located in this method, and thus analyzed it in more detail. We show the results in Figure 4.8. The `remove` method of class `Vec` contributes most of the computation time.

With our findings, we investigated the semantics of `ConstrGroup.removeFrom`, and especially the role of `Vec.remove` using the source code of Sat4j. In a nutshell, the Sat4j solver implementation maintains a global array of clauses. This array is encapsulated by the `Vec` class, which roughly resembles the implementation of a standard Java `ArrayList`[13]. In our REDUCED-MODEL use case, this array initially contains the reduced model Γ. When a query is issued, Sat4j temporarily appends all query clauses to the array, solves its satisfiability, and then removes the clauses again with a call to `Vec.remove` for each clause. This method linearly searches the particular clause in the array, and then removes it by shifting all following clauses one position to the left. Thus, we have a time complexity of $O(n^2)$ for the complete removal of a query, where n is the number of clauses in the query. The larger a query gets, the more performance degrades.

Since the REDUCED-MODEL strategy includes several models with possibly thousands of clauses in the query, the `Vec.remove` implementation becomes a bottleneck. By contrast, the FULL-MODEL approach generally issues smaller queries, since it does not add any models to the original query.

To remedy the problem, we exploit the fact that query clauses are always *appended* to the array, and precisely those appended clauses are removed afterwards. The array allocated in the `Vec` implementation may be larger than the actual number of elements. Therefore, the class maintains a pointer to the end of the array (variable `nbelem`), and ignores all elements beyond this pointer. Before each query, we save the pointer, and let Sat4j add the clauses. After the query has been processed, we simply restore the old pointer value. This way, we effectively achieve a removal operation in constant time without any array traversal or shifting of clauses. In other words, the time complexity decreases from $O(n^2)$ to $O(1)$.

As shown in the comparison in Figure 4.6, the optimization has an effect on REDUCED-MODEL, but leaves the performance of FULL-MODEL unchanged. We suppose that

[13]http://docs.oracle.com/javase/8/docs/api/java/util/ArrayList.html

our particular way of using Sat4j was not expected by its developers. Therefore, we have deployed this optimization on all measurements in the remainder of this chapter.

4.3.3 Results and discussion

With the optimization from the previous section, we conduct the measurements to compare the FULL-MODEL and REDUCED-MODEL strategies. To this end, we generate a lists with 1,000 to 5,000 queries and measure the runtime of each query in both approaches. We also record other query parameters, such as number of relevant submodels, and the sum clauses added by the REDUCED-MODEL strategy.

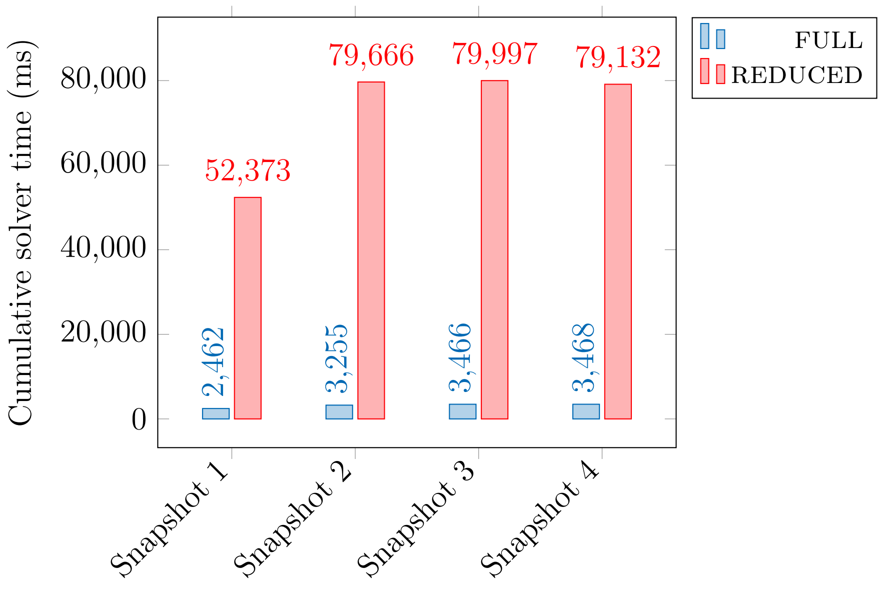


Figure 4.9: Comparison of cumulative solver times (1000 queries)

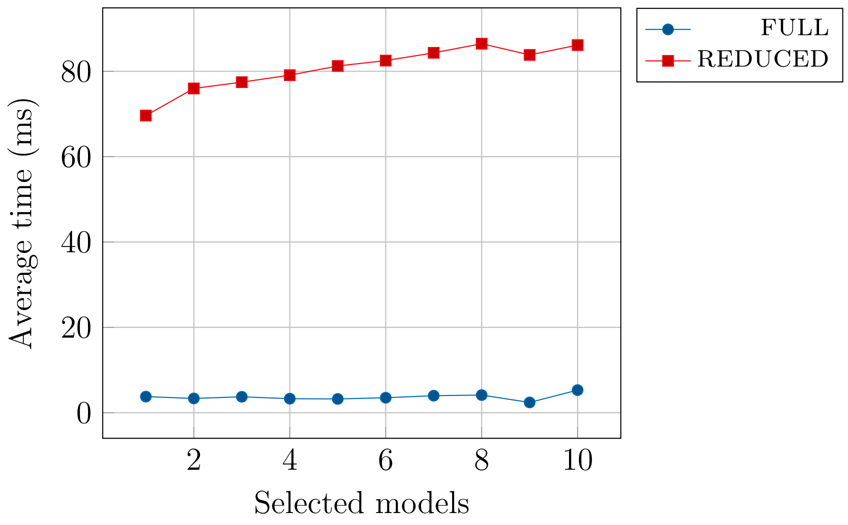


Figure 4.10: Average solver times against number of selected models (Snapshot 4)

Granularity of decompositions

In Figure 4.9, we compare cumulative times of the strategies for all snapshots. Despite the optimization, REDUCED-MODEL is between 21 (Snapshot 1) and 24 (Snapshot 2) times slower than FULL-MODEL. To assess the impact of locality, we show average solver times plotted against number of models selected by REDUCED-MODEL in Figure 4.10 for Snapshot 4. We see that REDUCED-MODEL is generally slower than FULL-MODEL for all measured cases. Note that the FULL-MODEL approach does not actually select any models. Instead, its x values merely reflect the number of submodels involved in the query.

As the performance of FULL-MODEL remains almost constant, locality does not have an impact for this strategy. By contrast, REDUCED-MODEL tends to perform worse the more models are selected. Since we were again surprised that REDUCED-MODEL was over 20 times slower, we aimed to find the cause for the discrepancy. From our experience with the optimization in the previous section, we suspected that the performance was still dependent on the number of added submodel clauses. For the following analysis, we exemplary use Snapshot 4, as the other snapshots exhibit similar characteristics. We provide the results for Snapshots 1–3 in Chapter A.

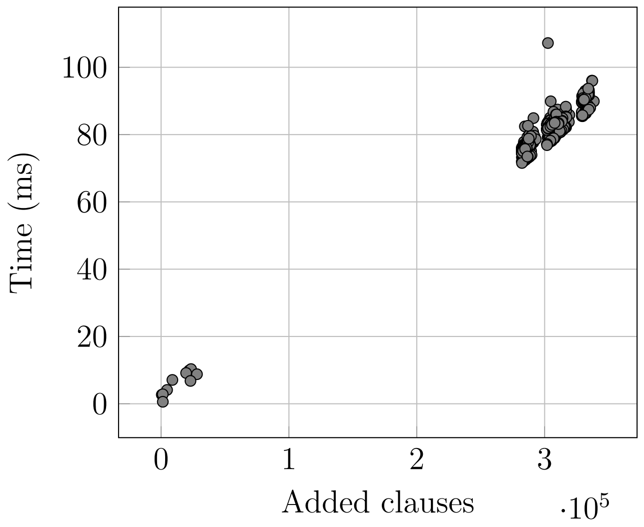


Figure 4.11: Solver times against number of added clauses, using REDUCED-MODEL

In Figure 4.11, we show the time of each query plotted against number of added clauses for REDUCED-MODEL. Most data points gather in the upper right corner. However, there are also a few data points in the lower left corner with up to 32,000 submodel clauses. Therefore, most queries require the strategy to add over 280,000 submodel clauses. Between both clusters, there is an horizontal gap with no queries. In the upper right point cluster, there are also two smaller gaps.

For comparison, we show the same type of plot for the FULL-MODEL strategy in Figure 4.12. There are two clusters in the lower left and lower right corners. However, compared to REDUCED-MODEL, there are more outliers near the right cluster. Since we used the same set of queries, the gaps remain the same. There is also a vertical gap that separates the points in both clusters.

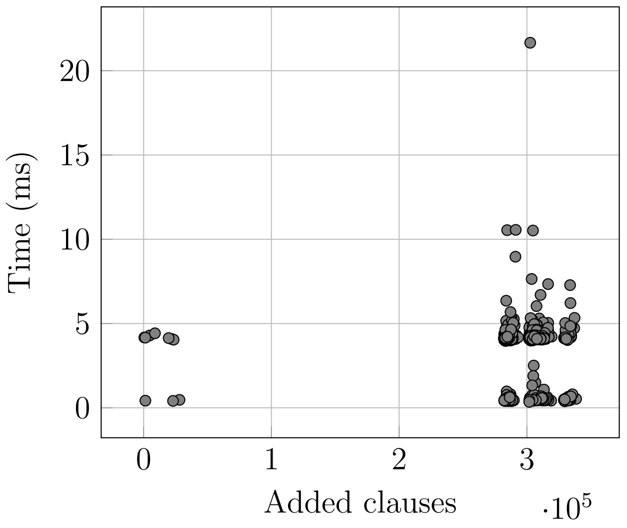


Figure 4.12: Solver times against number of added clauses, using FULL-MODEL

Figure 4.11 suggests a linear performance dependency on the number of added clauses for the REDUCED-MODEL approach. Thus, due to the large gap in the number of clauses, most queries are slower than their corresponding FULL-MODEL measurements, which again have mostly constant performance despite some outliers. Furthermore, we suppose that the vertical gap in the FULL-MODEL plot is caused by CNF simplification steps in the Sat4j implementation. In particular, if the CNF itself already can be transformed to a contradiction or tautology, the actual solver algorithm is not executed. However, we have not analyzed this in more detail.

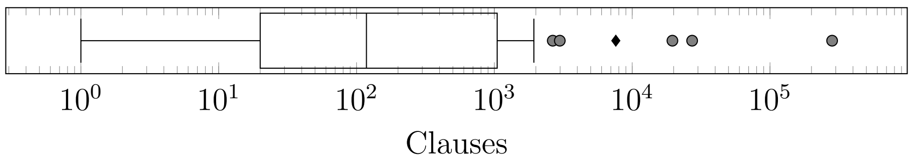


Figure 4.13: Distribution of number of clauses in submodels

The horizontal gap in the number of clauses suggests a large (clause-wise) submodel that also has a high probability to be relevant for a query. Therefore, we analyzed the distribution of clauses in all submodels. We show the results in Figure 4.13. The median lies at 118 clauses, and most models have up to 2,000 clauses. There are five outliers with 2,653, 2,985, 19,587, 27,229, and 282,268 clauses.

The largest submodel has roughly ten times more clauses than the next-largest submodel, which is still two orders of magnitude larger than the typical submodel. Furthermore, the largest submodel has 111,151 features, and has therefore an over 50 percent chance to be relevant already to a query with a single literal. We therefore conclude that the REDUCED-MODEL approach is ineffective if the decomposition contains single submodels with too many clauses.

Excluding large submodels

To simulate more fine-grained decompositions, in the following we excluded all local features of such outlier submodels from query generation. Therefore, the REDUCED-

MODEL does not include such models anymore. We also raised the number of generated queries to 5,000 to obtain more precision. Again, we show results of Snapshot 4 unless stated otherwise.

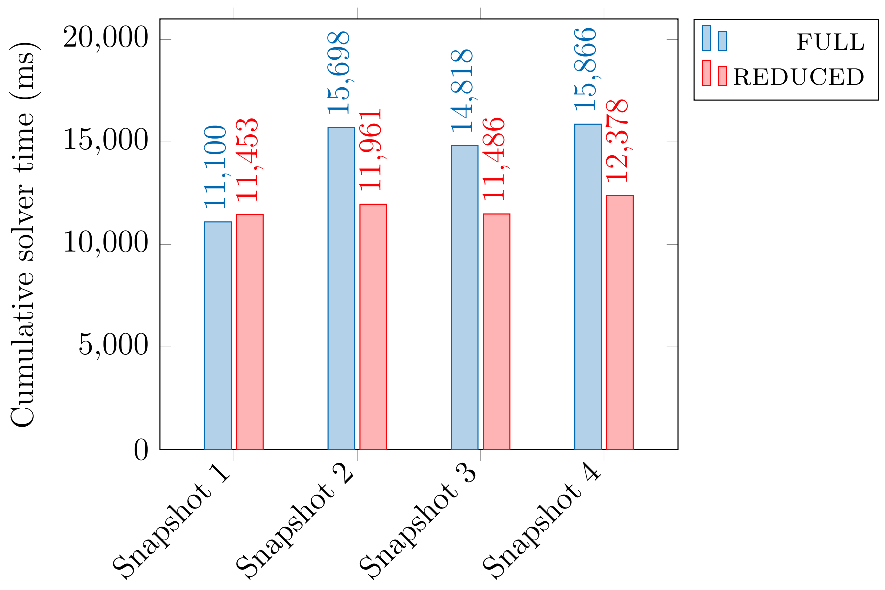


Figure 4.14: Comparison of cumulative solver times without large models (5,000 queries)

In Figure 4.14, we again compare cumulative times of the strategies, this time with large submodels excluded. For Snapshot 1, both strategies have almost the same performance, although FULL-MODEL is faster. By contrast, the REDUCED-MODEL approach is between 24 and 22 percent faster for Snapshots 2–4.

With a sufficiently fine-grained decomposition, our REDUCED-MODEL approach therefore allows for up to 24 percent of performance gain. However, the tie at Snapshot 1 shows that we cannot generally conclude that using REDUCED-MODEL is advantageous.

Impact of locality

We also again analyzed the impact of locality on average solver time in Figure 4.15. We see that for up to eight selected models, REDUCED-MODEL outperforms FULL-MODEL. This holds for all snapshots except Snapshot 1, which is only faster for up to six submodels.

Thus, we answer RQ 2.2 as follows. The average performance of queries in the REDUCED-MODEL strategy depends on the number of selected models. This strategy is therefore more effective for queries with higher locality. Conversely, all snapshots reach a point where REDUCED-MODEL becomes ineffective due to insufficient locality.

Threshold measures

The locality analysis shows that there still is some threshold beyond which the REDUCED-MODEL approach is slower due to its overhead. As we have discovered

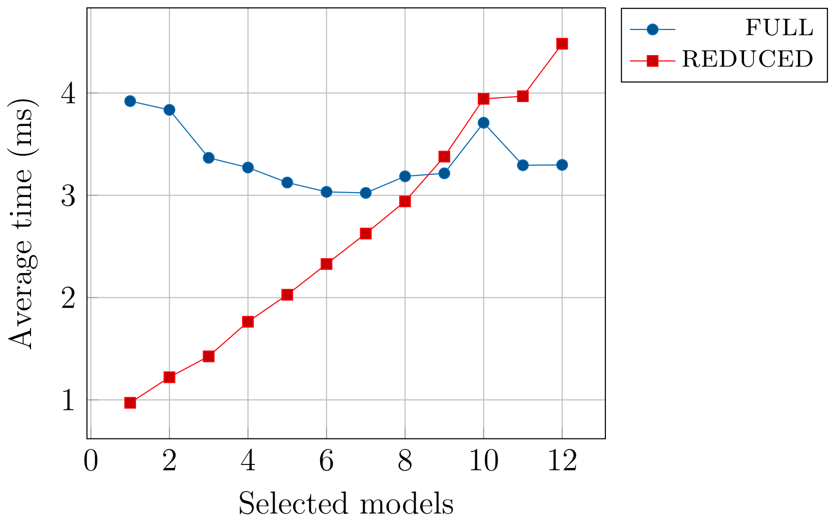


Figure 4.15: Average solver times against number of selected models, with large submodels excluded

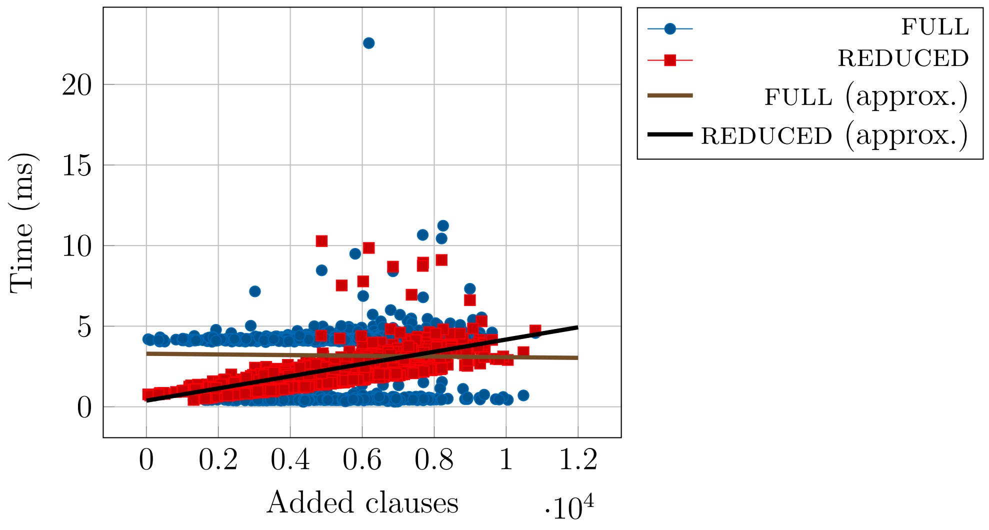


Figure 4.16: Solver times against added clauses, with large submodels excluded

that performance depends on the number of added clauses for REDUCED-MODEL in Figure 4.11, we now analyze this correlation with large submodels excluded to answer RQ 2.1 and RQ 2.3. Using the number of clauses allows for a more precise measure than using the number of relevant submodels, since the latter aggregates all queries into only 12 different data points.

In Figure 4.16, we compare solver times against the number of added clauses for both strategies. The performance of FULL-MODEL roughly alternates between two constant values. By contrast, the performance of REDUCED-MODEL increases as the number of added clauses increases. Compared to Figure 4.11, there are no apparent gaps.

The plot indicates a linear correlation for REDUCED-MODEL, however with an increasing variation. By contrast, the performance of FULL-MODEL appears to depend on some hidden property of each query, which we did not identify in this work. However, the data points concentrate at two constant values. We therefore propose a linear regression method [Rawlings et al., 1998] to model the average performance of queries by the number of submodel clauses.

In particular, we investigate the number of clauses in relevant submodels as a threshold measure $m(\Delta')$. To this end, we define the *clause measure* $m_c(\Delta')$ as the number of clauses in Δ'. To determine the threshold τ, beyond which REDUCED-MODEL becomes ineffective, we approximate the average time cost per submodel clause for each strategy as a linear function. The τ value is then the intersection of both functions.

Snapshot	Time / 10000 clauses (ms)	τ	$\widehat{\tau}$
1	3.72	5170	2.2%
2	3.55	7581	2.2%
3	3.69	7269	2.1%
4	3.79	7266	2.1%

Table 4.2: Approximated costs per clause, and thresholds for the ADAPTIVE-MODEL strategy

In Table 4.2, we show the approximated time costs per added submodel clause, and the threshold we have computed using the described method for each snapshot. All snapshots exhibit similar cost factors. While thresholds for Snapshots 2–4 are also similar, the threshold of Snapshot 1 is lower. Additionally, we show the approximated cost functions for Snapshot 4 in Figure 4.16.

The results show that the composition costs of the REDUCED-MODEL strategy are independent of the subject model. The smaller threshold of Snapshot 1 results from its smaller FULL-MODEL computation time. When comparing the thresholds with the overall number of model clauses from Table 4.1, we see that Snapshots 2–4 also have similar CNF sizes, but Snapshot 1 is smaller, which could explain the threshold difference. Therefore, we normalized each τ value by dividing it by the number of clauses in the corresponding full model. The resulting *normalized threshold* reflects the ratio between the number of submodel clauses and the number of clauses in the full model. We show this *normalized threshold* $\widehat{\tau}$ in Table 4.2. It is similar for all snapshots.

As a consequence, we are able to specify a normalized threshold measure with a uniform threshold value. We define the *normalized* measure $m'(\Delta')$ as the clause measure $m_c(\Delta')$ divided by the number of clauses n_f in the full model. Formally, $m'(\Delta') = m_c(\Delta)/n_f$. Such a model-independent threshold is favorable, since we can potentially transfer it to unknown models without prior measurements or estimates.

Threshold evaluation

Finally, we evaluate the performance gain when using the normalized threshold. To this end, we determine cumulative solver times for the ADAPTIVE-MODEL strategy. In particular, we sum up all data points from REDUCED-MODEL that are *below* the threshold, and add all data points from FULL-MODEL starting *from* the threshold.

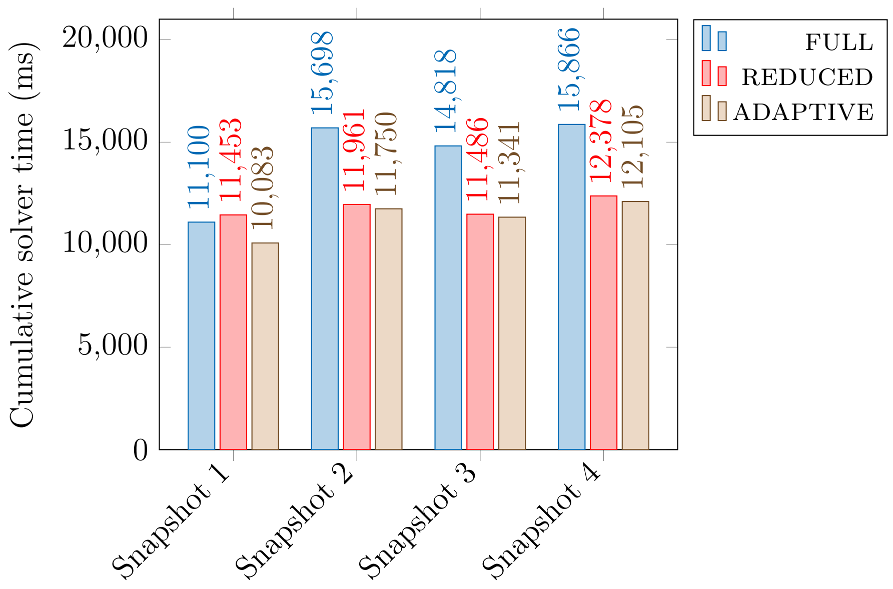


Figure 4.17: Comparison of all three strategies, using normalized threshold

We compare the performance of all three strategies in Figure 4.17, using 2.1 percent as the smaller one of the both determined threshold values. For Snapshot 1, we obtain a 10 percent improvement compared to FULL-MODEL. Moreover, as the REDUCED-MODEL strategy is slower than FULL-MODEL in this case, reasoning with interfaces is only effective with ADAPTIVE-MODEL. The other snapshots only slightly benefit from using ADAPTIVE-MODEL compared to REDUCED-MODEL.

In summary, we answer the two remaining research questions as follows. Regarding RQ 2.1, we have found a varying overhead that is roughly linearly dependent on the number of clauses in the submodels to be composed. The average penalty per additional clause is similar across different models. For small enough queries, the REDUCED-MODEL strategy is more efficient than FULL-MODEL despite the overhead. However, the overhead can become so large that it renders the REDUCED-MODEL approach inefficient by an order of magnitude, especially if the decomposition contains submodels with too many clauses.

To answer RQ 2.3, we have investigated an absolute and a relative threshold measure. We have used a linear regression to determine the threshold values. With normalizing, we are able to obtain a uniform threshold value for all analyzed models. For Snapshots 2–4, using the threshold with the ADAPTIVE-MODEL strategy shows no significant improvement. By contrast, reasoning with interfaces in Snapshot 1, which is the smallest model, is only effective when using the threshold. Thus, there is no definite answer to this research question. We suppose that the benefit of the ADAPTIVE-MODEL strategy would be greater if our evaluation subjects contained either smaller models or queries with less locality. Furthermore, we suggest to explore other, possibly more precise techniques to determine threshold measures and values in future work.

4.4 Threats to validity

In this section, we identify possible threats to the validity of our experiments. We distinguish between *internal validity*, which concerns the design of our experiments, and *external validity*, which affects the generalizability of our results.

Internal validity

When measuring solver performance in a family-based analysis of the Linux kernel in Section 4.2, we did not repeat any measurements due to the long duration of the experiment. Therefore, particular measurements may be distorted by accidental garbage collection, class loading, or just-in-time compilation. However, we measured over 120,000 data points. Thus, we still expect a sufficiently precise estimate of the average query duration.

Furthermore, we analyzed only about 1/6 of all files of the Linux kernel due to time constraints. Therefore, we might have missed extreme cases regarding reasoning time or locality. To prevent bias in the results caused by some accidental order, all measured files were randomly chosen.

The functional correctness of TypeChef is checked by unit tests. We upgraded the original experimental setup to the latest TypeChef version. Nevertheless, we cannot guarantee soundness of the recorded analysis results.

In our experiments for comparing the three reasoning strategies in Section 4.3, there may also be measurement errors due to internal JVM operations such as garbage collection. As a countermeasure, we repeated each solver query several times in different JVM instances, and replaced outliers with additional measurements. Therefore, we expect those measurements to be reliable and repeatable.

We performed the measurements for all strategies in an isolated benchmark implementation to capture the raw computation time. However, such a setup could be unrealistic, since in a real family-based analysis, reasoning and actual analysis operations are interleaved. For example, the REDUCED-MODEL strategy could occupy more JVM heap memory, which in turn causes the garbage collector to be triggered more often, such that the overall analysis becomes slower.

Our concept or implementation might contain errors, and could therefore accidently perform better than any correct implementation. However, we proved conceptual correctness in Chapter 3. Furthermore, we cross-checked satisfiability results between all three strategies to ensure functional correctness of the implementation. We also ensured that the time measurements are fair between different strategies. In particular, we took care not to include auxiliary operations[14] in the measurement for one strategy that we had not measured for a another strategy.

External validity

In Section 4.2, we gathered data from a single product line. The results might not apply to other product lines, as the Linux kernel is the largest one available. Especially the parsing times might be lower for other software systems. However, this would increase the potential of the approach presented in this work.

Furthermore, we only conducted one type of family-based analyis. Queries that arise during other analyses (e.g., data-flow analysis) might have different characteristics with regard to solver time or locality.

To compare the reasoning strategies, we derived semi-random queries from the Linux analysis data, and issued them against an unrelated model. While these queries are more realistic than randomly generated formulas, we lose the connection between feature dependencies in the query, and feature dependencies in the query. Therefore, an evaluation in which queries and models are related might yield different results. Therefore, we consider our evaluation as a first step to assess the potential performance gain of our approach.

Since we used four snapshots of a single model for our evaluation, we have less variance in the input data compared to using four unrelated models. Thus, this choice might not be representative, especially regarding the decomposition into submodels and the sets of local features. As the snapshots have obfuscated feature names, we cannot further judge the semantic quality of the model. Since we have no access to other models of similar size, we consider using only a single model in different versions as a necessary trade-off.

We used only a single SAT solver implementation for our measurements. Therefore, the results might be different for other solver algorithms and implementations, especially concerning the composition overhead we have determined.

4.5 Summary

In this chapter, we have evaluated the potential of reasoning with feature model interfaces in a family-based analysis. To this end, we have conducted a two-step evaluation.

First, we have analyzed properties of queries in a practical family-based analysis. We have type checked a subset of the Linux kernel, and recorded all SAT solver queries. Although variability-aware parsing took most of the processing time on average, we found that reasoning about presence conditions accounts for up to 60 percent of the actual analysis phase. Thus, we can potentially improve over half of

[14] e.g., computation of a query CNF

the analysis time with our approach. Furthermore, queries are local to some degree. A typical query contains nine distinct features out of over 10,000 available features, and even queries with over 500 literals only contain up to 27 distinct features.

In the second step, we have compared the performance of reasoning with interfaces against the conventional approach that uses the full feature model. To this end, we have used the queries recorded in the first evaluation step as templates to generate realistic queries for four versions of another feature model, for which we have a decomposition into submodels, and precomputed interfaces. As a first result, we found that reasoning with interfaces is not efficient if the decomposition contains submodels with too many clauses. Subsequently, we have excluded the problematic models to simulate a more fine-grained decomposition. We have shown that the interface approach can improve overall reasoning time by up to 24 percent despite the linear overhead introduced by the composition of submodels. Moreover, query locality is important. If a query is not local enough, reasoning with interfaces is inefficient. To further improve performance, we have investigated a threshold measure to select between both reasoning approaches, based on the number of submodels relevant to the query. Although the measure permits a uniform threshold for all model versions, applying the threshold in a combined reasoning strategy is only beneficial for the smallest model. By contrast, the other models show no improvement.

In summary, we are able to improve the time of a family-based analysis by up to 14 percent using the approach presented in this thesis. However, we need more realistic evaluation setups, where we process practical queries during a family-based analysis run using our approach. Furthermore, it remains an open question how a feature model has to be decomposed to achieve optimal performance.

5. Related work

In this chapter, we survey related work. We focus on feature model abstractions, and techniques to accelerate reasoning about feature models and presence conditions.

Schröter et al. [2016] propose *feature model interfaces* (FMIs) for compositional reasoning with feature models. The authors show that FMIs are sound and complete for several feature model analyses (e.g., core features), if one is interested in only interested in the interface features. Since FMIs are typically smaller than their corresponding concrete models, they can be used to accelerate such analyses. Furthermore, in an evolutionary scenario where a feature model is composed of several smaller models, FMIs prevent reanalyzing the composed model under certain restrictions, if one of the smaller models changes. We have based the approach presented in this work on FMIs. However, our theory is based on propositional logic rather than configuration semantics.

Acher et al. [2011] propose a slicing operation for feature models to remove a selection of features without affecting the relationships between the remaining features. They use slicing to achieve a seperation of concerns for large feature models by constructing different views [Acher et al., 2012]. The authors specify the slicing operation in terms of configuration semantics, and additionally as an existential quantification operation on propositional formulas. In this work, we have proven that both definitions are equivalent. In other words, we can use existential quantification to obtain abstractions of propositional formulas that are sound and complete for a subset of atoms. Moreover, Schröter et al. [2016] use feature model slicing to construct FMIs.

Krieter et al. [2016] propose an algorithm for feature model slicing that is based on logical resolution. In their evaluation, they compare different heuristics for the order of feature removal to achieve optimal performance.

Liebig et al. [2013] employ multiple techniques to improve the performance of presence condition reasoning in several family-based analyses, such as type checking and data-flow analysis. For example, they avoid reasoning with the feature model, and instead reason only using the presence conditions themselves when possible. They mostly use the feature model only to prune false positives from the final analysis results. Their TypeChef analysis tool[1] is able to use different feature models for different analysis phases. In particular, their variability-aware C parser uses a simplified model, whereas type checking is performed with the full model. In addition, TypeChef caches satisfiability results to avoid solving the same query multiple times. The technique we have proposed in this work could be a complementary optimization.

Liang et al. [2015] show in their experiments that propositional simplification rules reduce the formulas sizes of practical feature models by several orders of magnitude. They conclude that for this reason, SAT solving on feature models with thousands of features is still tractable, as SAT solvers internally perform such simplifications.

Finally, Mendonça et al. [2008] evaluate the use of binary decision diagrams (BDDs) for reasoning about feature models, as often BDDs are more efficient than other reasoning techniques. As size and performance of the BDD construction depend on the chosen variable ordering, they investigate existing heuristics to find such variable orderings. They show that these heuristics do not produce BDDs of tractable sizes. As a remedy, they propose new heuristics that exhibit a better performance when used for feature models.

[1]https://github.com/ckaestne/TypeChef/tree/v0.3.7

6. Conclusion and future work

Family-based analysis is a technique to handle the complexity of software product lines in a sound and complete way. In this thesis, we have presented an approach to accelerate the inherent feature model reasoning in such analyses by using the notion of feature model interfaces. In particular, we take advantage of the hierarchical structure of feature diagrams to produce abstractions of feature models, which can be selectively refined for individual reasoning tasks. Since these abstractions are typically smaller the original feature model, they allow for faster reasoning. We have demonstrated that our approach accelerates family-based analyses by up to 14 percent in our experimental setup.

We have formalized our approach using a propositional semantics for feature diagrams. First, we have defined a generic decomposition for propositional formulas. We have specified three reasoning strategies, which decide propositional entailments of the form $\Phi \vDash \varphi$, where Φ is some feature model formula with a precomputed decomposition, and φ is some presence condition from a family-based analysis. Instead of using the original model Φ, two of the strategies compose a smaller abstraction of Φ using the decomposition, with only elements that are relevant to the query φ. To this end, we have adapted the notion of feature model interfaces to propositional logic, resulting in the definition of a *propositional interface*. A propositional interface is an abstraction of a propositional formula that is equivalent to the original formula in all cases where only the interface atoms are involved.

Under the assumption that feature diagrams present hierarchical refinements of product-line functionality, and thus the diagram structure corresponds to the implementation structure of the particular product line, we have specified a decomposition operation for feature diagrams. Given a selection of subtrees of a feature diagram, we produce a feature model interface for the diagram by slicing out all features that are only relevant within a particular subtree. This interface can then be selectively reconcretized by adding individual submodels from the selection. We have proven that our decomposition operation is correct, that is, its result can be used by the reasoning strategies we have defined.

We have evaluated our approach to assess its potential and the limits of its applicability. To this end, we have first analyzed practical presence conditions in a family-based analysis of the Linux kernel. We have found that reasoning contributes up to 60 percent of the analysis time. Furthermore, although queries may contain over 500 literals, they typically involve only nine distinct features. Such query locality makes our approach more efficient, since it leads to fewer composition operations in the strategies.

In a second step, we have compared the performance of the three reasoning strategies using four versions of a feature model from the automotive domain, for which pre-computed submodels and interfaces are available. We have found that the efficiency of our approach depends on the overall number of CNF clauses in the submodels to be composed. Therefore, the granularity of a decomposition influences efficiency, and consequently decompositions with smaller submodels (with respect to the number of clauses) should be preferred. Contrariwise, decompositions with too large submodels even make our approach completely inefficient. With such outlier submodels excluded, our approach outperformed conventional reasoning with the full feature model for all model versions by up to 24 percent. However, reasoning with the smallest model was only faster when using the adaptive reasoning strategy, which employs the conventional approach beyond some threshold value of the composition size.

Future work

This thesis gives rise to several directions of future work. The optimal structure of feature diagram decompositions, for example with respect to submodel granularity, is an immediate open research question. To achieve optimal performance, we suppose that implementation artifacts must correspond to the diagram hierarchy and the selection of subtrees for the decomposition, as this leads to more local queries. In this context, it would also be interesting to know how well this correspondence is for existing product lines, and if we can analyze implementation artifacts to select an optimal set of submodels.

Furthermore, a particular weakness of our evaluation is the use of previously recorded queries on an unrelated feature model. To conduct a more realistic evaluation, we would ideally integrate our approach into a family-based analysis tool such as TypeChef, and measure its performance during a real analysis run. However, for such an evaluation setup we need a sound and complete feature diagram of the product line that reflects domain knowledge. Unfortunately, feature models are often encoded in build systems such as makefiles or Kconfig files. We need techniques to reverse engineer a precise and correct feature diagram from such artifacts, which also retains domain knowledge in its structure. In particular, we are interested in a comprehensive feature diagram of the Linux kernel.

We have only used a single SAT solver implementation (i.e., Sat4j) to analyze the potential of our approach. The Sat4j optimization we have performed during our evaluation (see Section 4.3.2 on page 52) shows that implementation peculiarities significantly influence the performance of our approach. Therefore, we suggest a more thorough survey and analysis of SAT solving algorithms with regard to our way of solver utilization, in order to select candidates for a detailed evaluation. Particular techniques could be BDD solvers or, as feature model interfaces can be computed by existential quantification, solvers for quantified boolean formulas (QBFs) [Büning and Bubeck, 2009].

Finally, there are also other possibilities to use feature model interfaces for reasoning. For example, instead of having a reduced model Γ that connects and integrates the submodels, we could employ *strong* interfaces of submodels, which are already an interface of the full model.

A. Evaluation results

In this chapter, we present detailed evaluation results for Snapshots 1–3 of the automotive feature model used in Chapter 4.

A.1 Snapshot 1

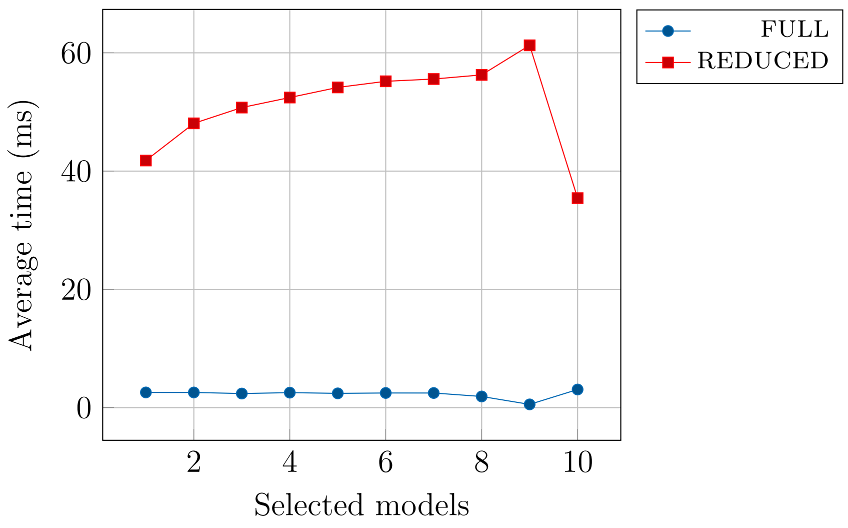


Figure A.1: Average solver times against number of selected models, without excluding submodels (Snapshot 1)

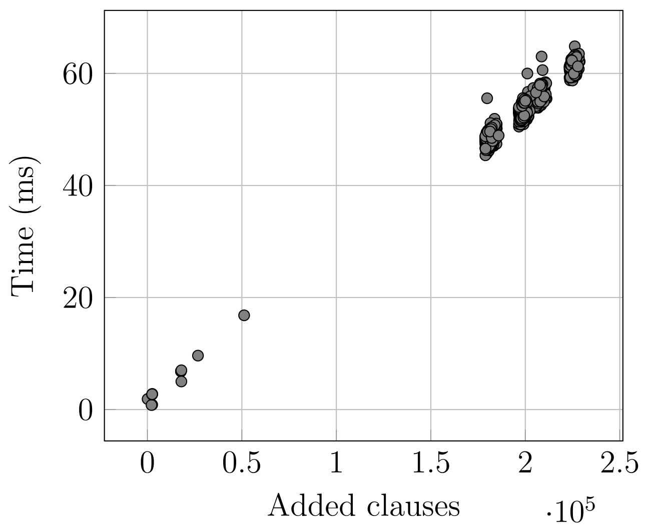


Figure A.2: Solver times against number of added clauses, using REDUCED-MODEL, without excluding submodels (Snapshot 1)

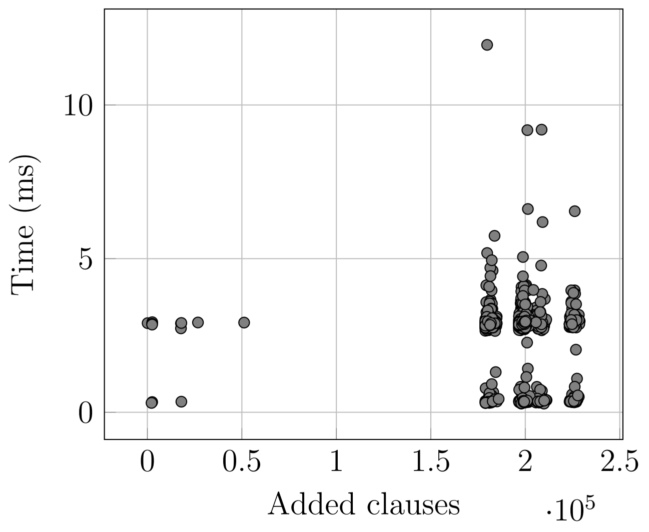


Figure A.3: Solver times against number of added clauses, using FULL-MODEL, without excluding submodels (Snapshot 1)

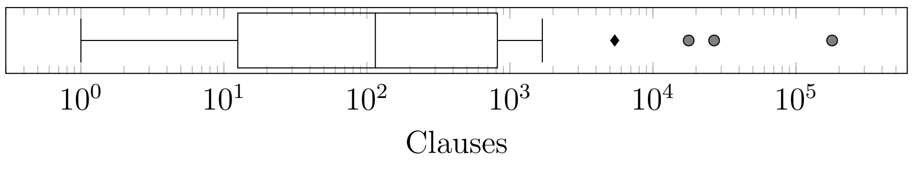


Figure A.4: Distribution of number of clauses in submodels (Snapshot 1)

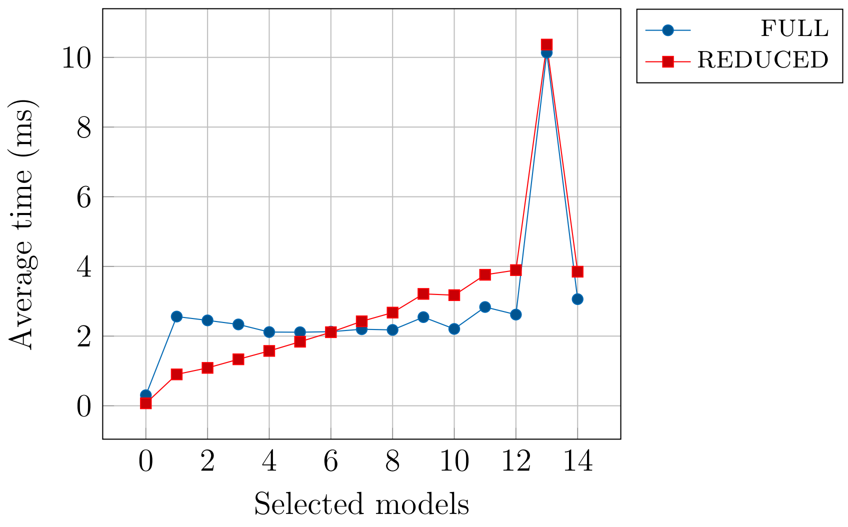


Figure A.5: Average solver times against number of selected models, with large submodels excluded (Snapshot 1)

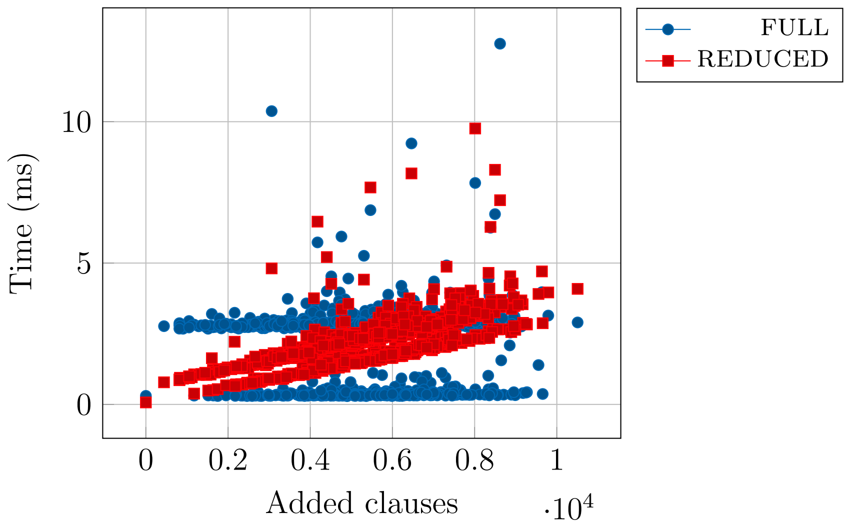


Figure A.6: Solver times against added clauses, with large submodels excluded (Snapshot 1)

A.2 Snapshot 2

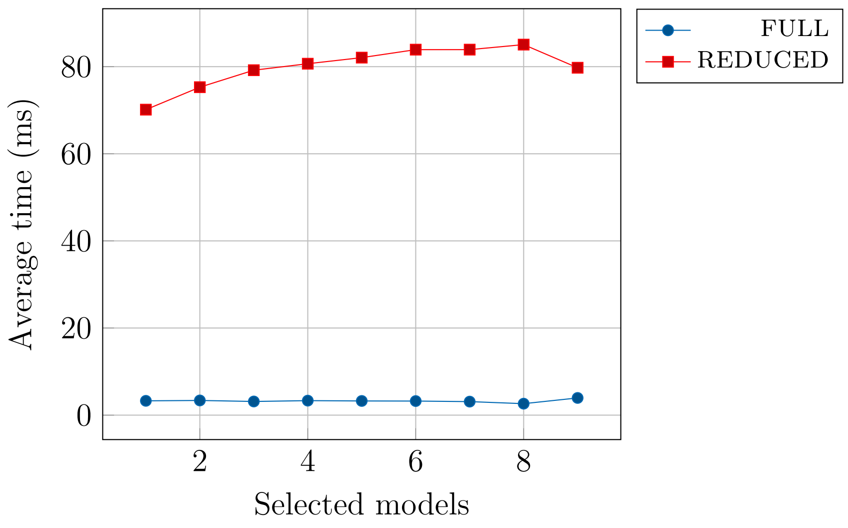


Figure A.7: Average solver times against number of selected models, without excluding submodels (Snapshot 2)

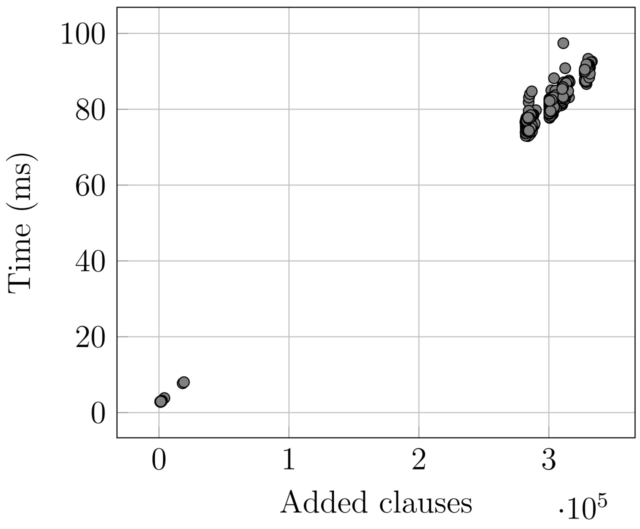


Figure A.8: Solver times against number of added clauses, using REDUCED-MODEL, without excluding submodels (Snapshot 2)

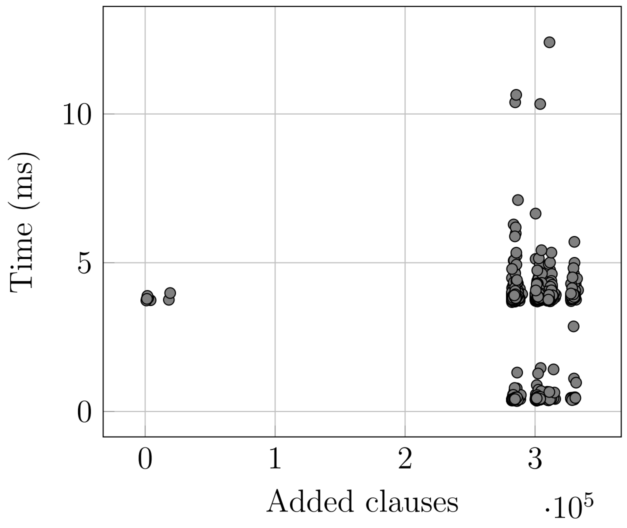


Figure A.9: Solver times against number of added clauses, using FULL-MODEL, without excluding submodels (Snapshot 2)

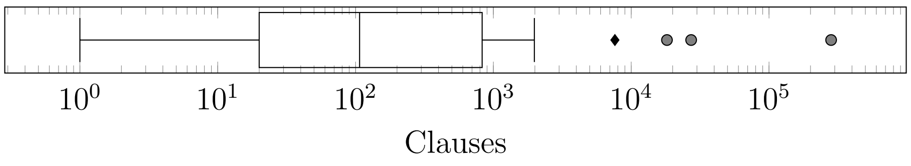


Figure A.10: Distribution of number of clauses in submodels (Snapshot 2)

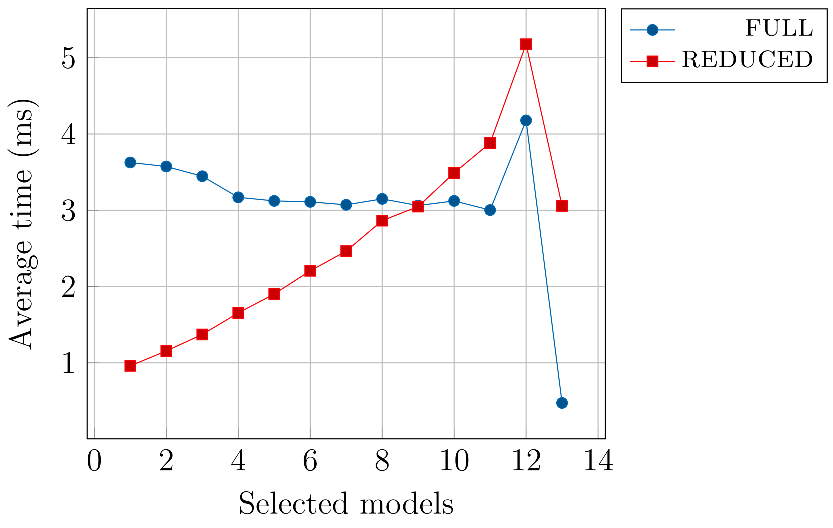


Figure A.11: Average solver times against number of selected models, with large submodels excluded (Snapshot 2)

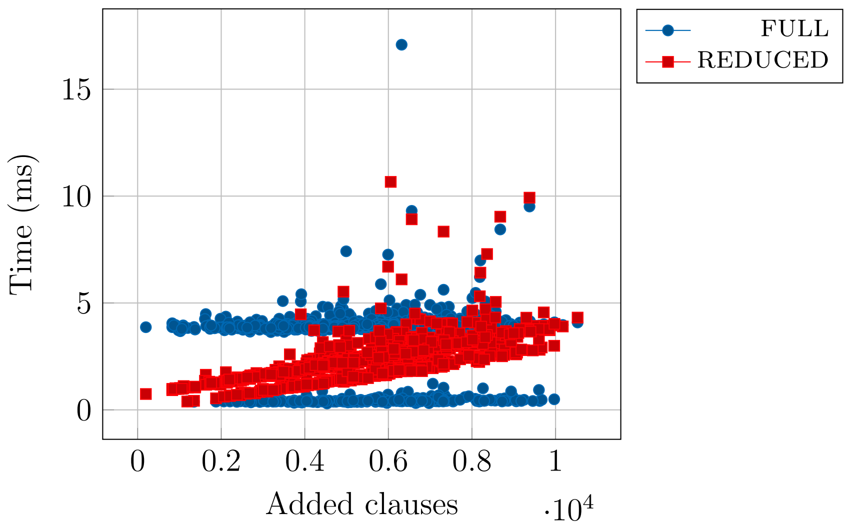


Figure A.12: Solver times against added clauses, with large submodels excluded (Snapshot 2)

A.3 Snapshot 3

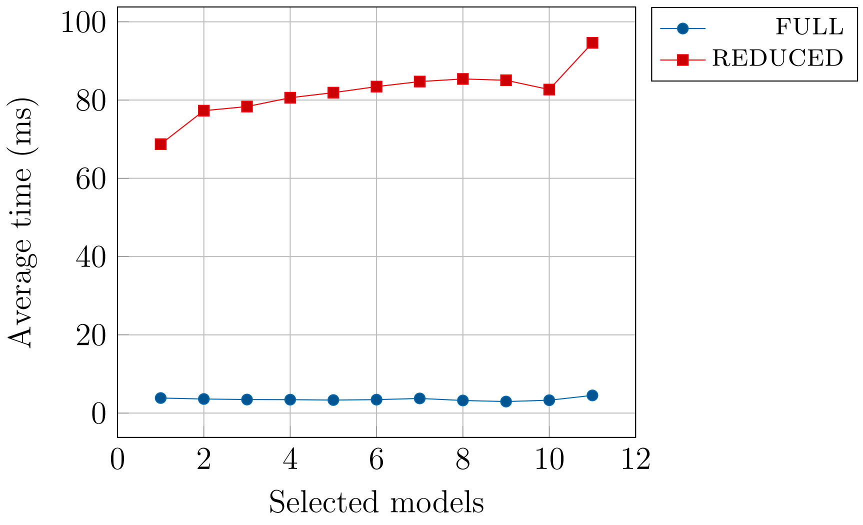


Figure A.13: Average solver times against number of selected models, without excluding submodels (Snapshot 3)

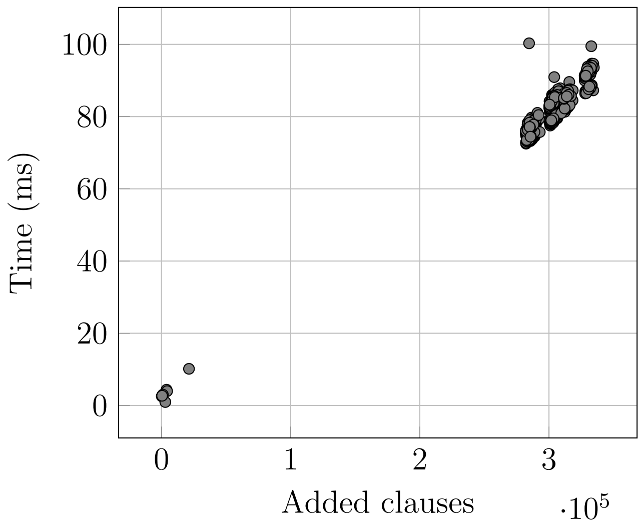


Figure A.14: Solver times against number of added clauses, using REDUCED-MODEL, without excluding submodels (Snapshot 3)

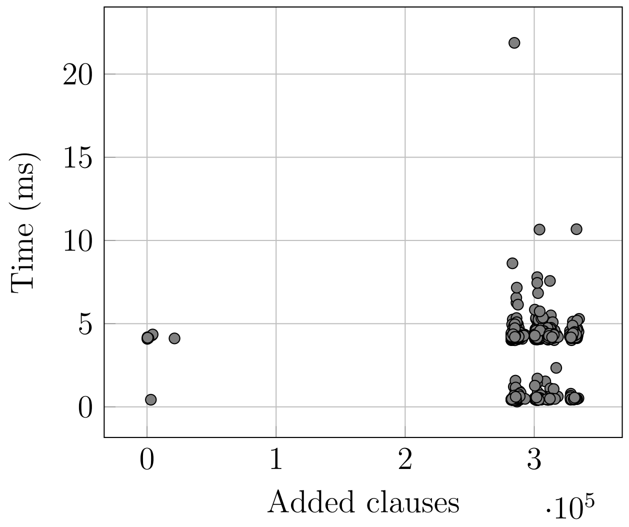


Figure A.15: Solver times against number of added clauses, using FULL-MODEL, without excluding submodels (Snapshot 3)

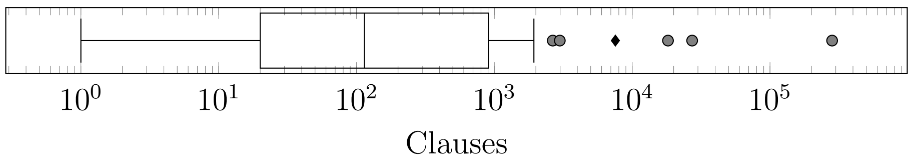


Figure A.16: Distribution of number of clauses in submodels (Snapshot 3)

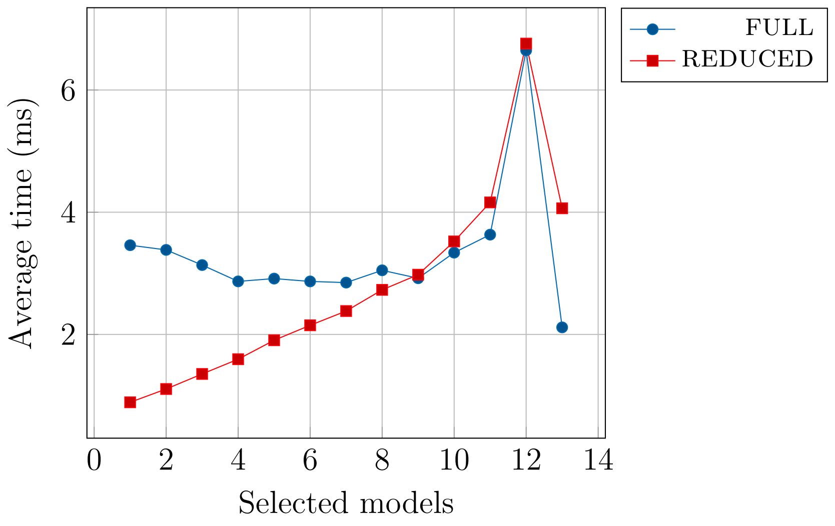


Figure A.17: Average solver times against number of selected models, with large submodels excluded (Snapshot 3)

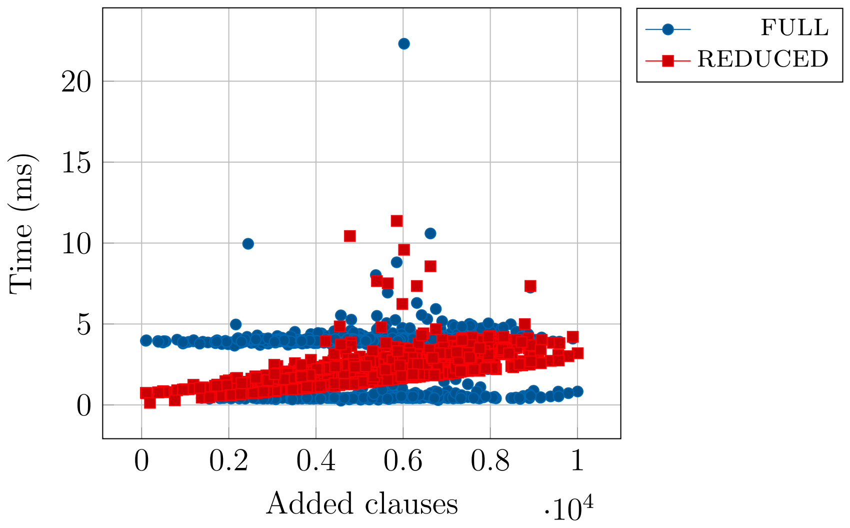


Figure A.18: Solver times against added clauses, with large submodels excluded (Snapshot 3)

Bibliography

Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing feature models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 424–427. IEEE, 2011. (cited on Page 3, 9, 16, and 65)

Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Separation of concerns in feature modeling: Support and applications. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 1–12. ACM, 2012. (cited on Page 65)

Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines - Concepts and Implementation*. Springer, 2013a. (cited on Page 1, 9, and 19)

Sven Apel, Christian Kästner, and Christian Lengauer. Language-independent and automated software composition: The FeatureHouse experience. *IEEE Transactions on Software Engineering (TSE)*, 39(1):63–79, 2013b. (cited on Page 18 and 19)

Sven Apel, Sergiy S. Kolesnikov, Norbert Siegmund, Christian Kästner, and Brady Garvin. Exploring feature interactions in the wild: the new feature-interaction challenge. In *Proceedings of the International Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8. ACM, 2013c. (cited on Page 19)

Don S. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005. (cited on Page 2 and 12)

Don S. Batory, Jacob Neal Sarvela, and Axel Rauschmayer. Scaling step-wise refinement. *IEEE Transactions on Software Engineering (TSE)*, 30(6):355–371, 2004. (cited on Page 18 and 19)

David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated analysis of feature models 20 years later: A literature review. *Information Systems*, 35(6): 615–636, 2010. (cited on Page 2, 12, 13, and 14)

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wasowski, and Krzysztof Czarnecki. Variability modeling in the real: a perspective from the operating systems domain. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 73–82, 2010. (cited on Page 10 and 18)

Daniel Le Berre and Anne Parrain. The Sat4j library, release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–6, 2010. (cited on Page 8)

Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986. (cited on Page 45)

Hans Kleine Büning and Uwe Bubeck. Theory of quantified boolean formulas. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, chapter 23, pages 735–759. IOS Press, 2009. (cited on Page 17 and 69)

Paul C. Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (cited on Page 1 and 9)

Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM, 2000. (cited on Page 1 and 11)

Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically rigorous Java performance evaluation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 57–76. ACM, 2007. (cited on Page 45 and 51)

John Harrison. *Handbook of practical logic and automated reasoning*. Cambridge University Press, 2009. (cited on Page 2, 7, 8, and 34)

Claus Hunsen, Bo Zhang, Janet Siegmund, Christian Kästner, Olaf Leßenich, Martin Becker, and Sven Apel. Preprocessor-based variability in open-source and industrial software systems: An empirical study. *Empirical Software Engineering*, pages 1–34, 2015. (cited on Page 1 and 18)

Michael Huth and Mark Ryan. *Logic in computer science*. Cambridge University Press, second edition, 2004. (cited on Page 5, 8, 9, 16, 17, and 34)

Kyo C. Kang, Sholom G. Cohen, James A. Hess, William E. Novak, and A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, 1990. (cited on Page 1 and 10)

Christian Kästner, Paolo G. Giarrusso, Tillmann Rendel, Sebastian Erdweg, Klaus Ostermann, and Thorsten Berger. Variability-aware parsing in the presence of lexical macros and conditional compilation. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 805–824. ACM, 2011. (cited on Page 1, 2, and 48)

Christian Kästner, Klaus Ostermann, and Sebastian Erdweg. A variability-aware module system. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 773–792. ACM, 2012. (cited on Page 1 and 21)

Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice-Hall, second edition, 1988. (cited on Page 18)

Jonathan Koscielny, Sönke Holthusen, Ina Schaefer, Sandro Schulze, Lorenzo Bettini, and Ferruccio Damiani. DeltaJ 1.5: Delta-oriented programming for Java 1.5. In *Proceedings of the International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages and Tools (PPPJ)*, pages 63–74. ACM, 2014. (cited on Page 19)

Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing algorithms for efficient feature-model slicing. In *Proceedings of the International Systems and Software Product Line Conference (SPLC)*, pages 60–64. ACM, 2016. (cited on Page 17 and 65)

Jia Hui Liang, Vijay Ganesh, Krzysztof Czarnecki, and Venkatesh Raman. SAT-based analysis of large real-world feature models is easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 91–100. ACM, 2015. (cited on Page 2, 8, 24, 51, and 66)

Jörg Liebig, Christian Kästner, and Sven Apel. Analyzing the discipline of preprocessor annotations in 30 million lines of C code. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 191–202. ACM, 2011. (cited on Page 48)

Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable analysis of variable software. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)*, pages 81–91. ACM, 2013. (cited on Page 2, 45, and 65)

Roberto E. Lopez-Herrejon and Don Batory. A standard problem for evaluating product-line methodologies. In *Proceedings of the International Conference on Generative and Component-Based Software Engineering (GCSE)*, pages 10–24. Springer, 2001. (cited on Page 9)

Marcílio Mendonça, Andrzej Wasowski, Krzysztof Czarnecki, and Donald D. Cowan. Efficient compilation techniques for large scale feature models. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 13–22. ACM, 2008. (cited on Page 66)

Marcilio Mendonca, Andrzej Wąsowski, and Krzysztof Czarnecki. SAT-based analysis of feature models is easy. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 231–240. Carnegie Mellon University, 2009. (cited on Page 2)

Leonardo Passos, Marko Novakovic, Yingfei Xiong, Thorsten Berger, Krzysztof Czarnecki, and Andrzej Wąsowski. A study of non-boolean constraints in variability models of an embedded operating system. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 2:1–2:8. ACM, 2011. (cited on Page 12)

John O. Rawlings, Sastry G. Pantula, and David A. Dickey. *Applied Regression Analysis: A Research Tool*. Springer, second edition, 1998. (cited on Page 59)

Ina Schaefer, Lorenzo Bettini, Viviana Bono, Ferruccio Damiani, and Nico Tanzarella. Delta-oriented programming of software product lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 77–91. Springer, 2010. (cited on Page 19)

Pierre-Yves Schobbens, Patrick Heymans, Jean-Christophe Trigaux, and Yves Bontemps. Generic semantics of feature diagrams. *Computer Networks*, 51(2):456–479, 2007. (cited on Page 10)

Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-model interfaces: the highway to compositional analyses of highly-configurable systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 667–678. ACM, 2016. (cited on Page 2, 3, 13, 14, 27, 50, and 65)

Reinhard Tartler, Daniel Lohmann, Julio Sincero, and Wolfgang Schröder-Preikschat. Feature consistency in compile-time-configurable system software: facing the Linux 10,000 feature problem. In *Proceedings of the European Conference on Computer Systems*, pages 47–60. ACM, 2011. (cited on Page 22)

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A classification and survey of analysis strategies for software product lines. *ACM Computing Surveys*, 47(1):6:1–6:45, 2014a. (cited on Page 1 and 20)

Thomas Thüm, Christian Kästner, Fabian Benduhn, Jens Meinicke, Gunter Saake, and Thomas Leich. FeatureIDE: An extensible framework for feature-oriented software development. *Science of Computer Programming*, 79:70–85, 2014b. (cited on Page 3 and 49)

All URLs cited in this work have been visited on December 6, 2016.

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Braunschweig, den 7. Dezember 2016