University of Magdeburg

School of Computer Science

Master's Thesis

Efficient Configuration of Large-Scale Feature Models Using Extended Implication Graphs

Author:

Sebastian Krieter

19.10.2015

Advisors:

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. Reimar Schröter

University of Magdeburg - School of Computer Science

Dr.-Ing. Thomas Thüm

TU Braunschweig - Institute of Software Engineering and Automotive Informatics

Krieter, Sebastian:
Efficient Configuration of Large-Scale Feature Models Using Extended Implication Graphs
Master's Thesis, University of Magdeburg, 2015.

Acknowledgments

I would like to thank my advisors Prof. Gunter Saake, Thomas Thüm, and Reimar Schröter for giving me the possibility of writing this master's thesis. A special thanks to Reimar for his fast and constructive feedback.

I also thank my friends and family and everybody else who supported me during the creation of this thesis.

Contents

List of Figures

List of Tables

1. Introduction

Software product line engineering (SPLE) has become an important concept to develop software and software-intensive systems. It enables developers to efficiently create customized software for various customers in terms of development time and costs [PBvdL05, CN01]. In SPLE, developers provide single software artifacts instead of complete software products. By composing a subset of all available software artifacts, with respect to their mutual dependencies, developers are able to build individual, coherent software products. Therefore, developers are able to efficiently develop and maintain variable and common source code parts for all their products [PBvdL05]. In our thesis, we use the common term *feature* to refer to software artifacts.

In order to build specific products in SPLE, one has to specify the set of included features. This vital aspect of SPLE is called the *configuration process*, which results in a *configuration* that specifies the included features of one software product [CN01]. Naturally, not all features can be freely composed together, due to certain dependencies and interactions among each other. To define the possible, valid combinations, developers have to provide a feature model that specifies the relationships between all features [CE00, ABKS13]. Thereby, the developers only allow feasible combinations of features that can be composed to a correctly working product. It is part of the configuration process to test whether the defined set of features is in accordance with the dependencies given by the feature model [PBvdL05]. By nature, the test for validity of a given combination can be done in polynomial time, however the task of finding a valid combination (i.e., the configuration process) is NP-complete [Coo71].

In most cases, the configuration process is sequential, with the developers deciding the inclusion of each feature, step-by-step [CHE05]. In each configuration step, the developers decide whether they include or exclude a given feature from the product that they want to build. Due to the feature's interdependencies, the decision for one specific feature can lead to the forced inclusion or exclusion of other depending features. There are two conceptually differing methods to handle this situation by either ignoring or determining the implications resulting from the latest decision. The first method ignores the potential implications in each configuration step and later checks whether any dependencies of the feature model are violated [WSB+08]. If so, the developers have to resolve the problem by revoking some of their decisions. The second method determines all implications and updates the current set of features after each configuration step. This leads to an interactive configuration process, in which the developers receive feedback about the implications of their last made decision [HSJ+04]. For an interactive configuration process, it is necessary to automatically determine all implications of a decision, which is called *decision propagation*. While the first method requires less computational effort than the second one, in some cases, it may lead to a frustrating configuration process for the developer, due to a high amount of revocations. Hence, if the developer's system has sufficient computational recourses, then the second method is more preferable.

An interactive configuration relies on decision propagation for each configuration step. Consequently, whenever a configuration step is performed, an algorithm has to determine whether the current step implies the in- or exclusion of other features. However, determining the configuration status of another feature, given a set of arbitrary dependencies, is an NP-complete problem and, thus, in general its execution time grows exponentially with an increasing number of features. Thus, straight-forward algorithms for decision propagation are unable to handle the configuration of large product lines in a feasible amount of time. Especially for large feature models with $10,000$ or more features (e.g., a model of the Linux kernel [TLD+11]), such an approach may require several minutes to finish one single configuration step, which is highly impracticable. Still, there is evidence that points out that most real-world feature models do not contain highly complex feature dependencies [MWC09]. Therefore, it is likely that for most real-world product lines, there are efficient ways to apply decision propagation for the configuration process. Based on this assumption, we want to find a decision propagation algorithm that performs more efficiently, regarding computation time, for large-scale feature models, which are used in industry today.

In our thesis, we propose a new approach that is based on implication graphs, which are known from the domain of boolean algebra. By expressing all dependencies of a feature model as an implication graph, the problem of decision propagation becomes easy to solve [APT79]. In detail, the decision propagation is reduced to solving multiple 2-satisfiability problems which are known to be P-complete [HJ90]. However, most feature models cannot entirely be expressed as an implication graph, due to their complex dependencies. Nevertheless, we try to utilize its advantages by expressing simple dependencies as a partial implication graph and storing additional information about the remaining complex dependencies. For this, we extend ordinary implication graphs to suit our needs and call the resulting data structure *feature graph*. Our new proposed approach, the *configuration assistant*, uses feature graphs to reduce the amount of computational effort for the decision propagation and, thus, achieves a faster performance for this process.

From a scientific point of view, we want to answer the following research questions.

RQ1: *Does the usage of a feature graph significantly reduce the required computational effort for decision propagation?*

RQ2: *Does the performance improvement dependent on the used feature model and if so, which kinds of feature models are most suited for our approach?*

RQ3: *How is the overall performance of the feature graph, regarding construction time and memory consumption?*

Goals and Contribution

In accordance to our scientific research questions, we infer that our main objective is to investigate the efficiency of our new approach, as part of our evaluation, and to determine feature-model structures for which our approach is most suitable. Aside from the scientific investigation, we make the following contributions.

- We introduce our new approach the configuration assistant.
- We implement the configuration assistant as part of the FeatureIDE framework.
- We compare our approach with other state-of-the-art configuration tools.

In addition to a fast performance for decision propagation, we require certain secondary conditions for our new approach. In particular, we design our approach to have the following properties.

1. Our configuration assistant can operate on arbitrary feature models and always provides a complete and correct result.
2. The computations to determine the features' configuration status are independent from each other.

These secondary conditions result from technical requirements and certain functionalities that we want to support with our approach. In detail, we want to integrate our approach in an existing framework, which relies on an exact result of the decision propagation. There exist efficient decision-propagation methods that only work on certain feature model structures [Men09]. Unlike these methods, we require that we can apply decision propagation to any kind of feature model and receive a correct result in every case (cf. Condition 1). In addition, we want to use several implementation techniques such as multi-threading to further improve the performance of our approach. In order to use these techniques, we have to be able to determine the configuration status of each feature in an arbitrary order or even in parallel. Thus, we must be able to compute each feature's configuration status independently of each other (cf. Condition 2).

Outline

In order to related to our new approach, we provide background information on SPLE in Chapter 2, with a particular focus on the configuration process. In Chapter 3, we introduce our new approach, the configuration assistant, and present its core concept, the feature graph. Moreover, in Chapter 4, we state details of the configuration assistant's implementation that we used for our evaluation. In Chapter 5, we describe our evaluation concept for answering our research questions and present the evaluation results. Subsequently, in Chapter 6, we talk about similar approaches and related topics. In Chapter 7, we summarize all our findings and draw a conclusion. Finally, we talk about possible future work in Chapter 8.

2. Background

In this chapter, we give all necessary information to comprehend to our new approach for an interactive configuration process. We explain the concept of software product line engineering, where we especially focus on feature modeling and product-line configuration. In detail, we show two different feature-model representations, feature diagrams and propositional formulas. Furthermore, we describe the general concept and challenges of the interactive configuration process. Finally, we review relevant feature-model analyses, which are necessary for our approach.

2.1 Software Product Line Engineering

At first, we define *software product line engineering* (SPLE) in accordance to Pohl et al. as "a paradigm to develop software applications (software-intensive systems and software products) using [...] mass customization" [PBvdL05]. In SPLE, we achieve mass customization by implementing *reusable software artifacts* that we can individually combine to build certain customized products. For this, we develop *common* and *variable* software artifacts and embed them in one *software product line* (SPL). Thus, an SPL represents multiple customized software products that share a common source-code basis [CN01, CE00].

2.1.1 Applications of SPLE

The main advantage to choose SPLE over conventional software development is the efficiency increase, when fulfilling the requirements of multiple customers. The development of reusable artifacts introduces some development overhead, compared to the development of a single product. However, when we develop multiple products, based on an SPL, we do not have to implement every new product from scratch. Hence, we save development time for new products, which we depict in Figure 2.1. Similarly, the initial development costs of an SPL amortize over time, due to smaller costs for single

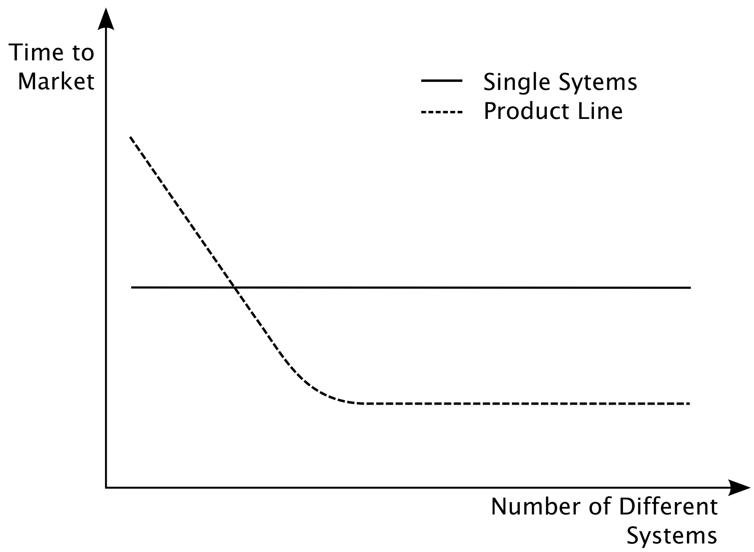

Figure 2.1: Comparison of development time between product lines and single systems. [PBvdL05]

software products, which we depict in Figure 2.2. Additionally, SPLE eases the maintenance of the derived products. When we need to modify source code that is common in multiple products, we only have to edit the corresponding software artifacts, instead of maintaining every product on its own. Therefore, we save development time when extending or debugging already existing source code.

There are several frameworks and tools that can be used for SPLE. In our thesis, we use FeatureIDE as basis for our approach. FeatureIDE is a framework for SPLE that allows us to develop, configure, and analyze SPLs [TKB+14].

2.1.2 Domain and Application Engineering

SPLE can be divided into two consecutive tasks, *domain engineering* and *application engineering*. As both are relevant for our approach, we describe them briefly in the following.

Domain Engineering

In domain engineering, the developers define all common and variable artifacts of a software product line [CE00]. Additionally, in order to manage the commonality and variability of a software product line, developers define variability models, which specify the dependencies between all artifacts of the product line. In our thesis, we focus on variability models based on *features* that are organized in a *feature model* to manage variability. Feature models map all artifacts of an SPL onto a set of features and describes the dependencies between these features.

Domain engineering also includes the implementation of the single features. However, we do not consider the actual implementation in this thesis, since it is independent from

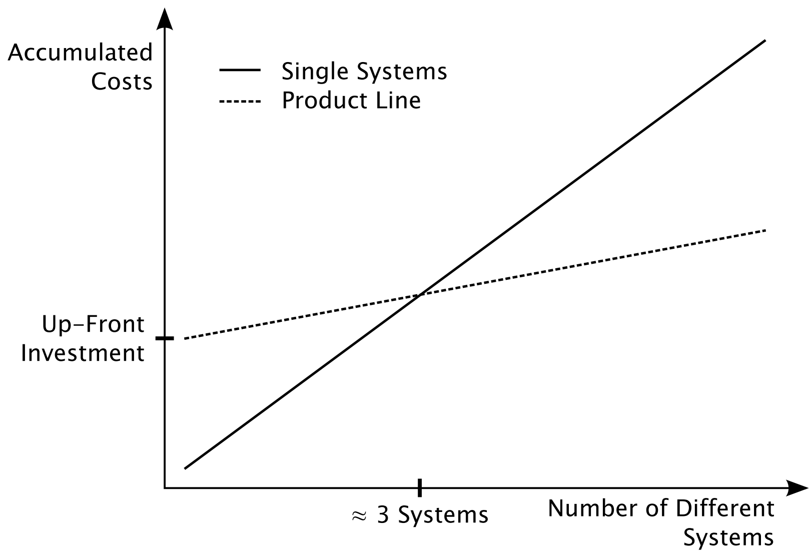

Figure 2.2: Comparison of development costs between product lines and single systems. [PBvdL05]

the pure feature-modeling and configuration process. We examine feature modeling further in the separate Section 2.2. Furthermore, we describe the analysis of feature models in Section 2.4.

Application Engineering

In application engineering, the developers derive the final products of an SPL by composing its single features with respect to the dependencies of the SPL's feature model [PBvdL05]. One aspect of application engineering is the decision of which features are composed to a final product. This decision is called the *configuration process*, which is the objective of our thesis. In Section 2.3, we describe the configuration process in more detail.

There exist many different implementation techniques for the actual composition to final software products [ABKS13]. These implementation techniques specify the generation mechanism and by this determine the final source code for single products. For instance, there are preprocessors [Käs10], aspect-oriented programming (AOP) [KLM+97], and feature-oriented programming (FOP) [ABKS13, Pre97, AKL13, Bat06]. Though, we do not need to consider these different techniques for our approach, since we are working with feature models and their specified dependencies. Feature models are on a more abstract level and, thus, independent of the chosen implementation technique. Hence, in this work, we focus on the configuration of an SPL, rather than the actual implementation.

2.2 Feature Modeling

In literature, we find several definitions for a feature of an SPL. We decided to define a feature in conformity with Kang et al. as "a prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems" [KCH+90].

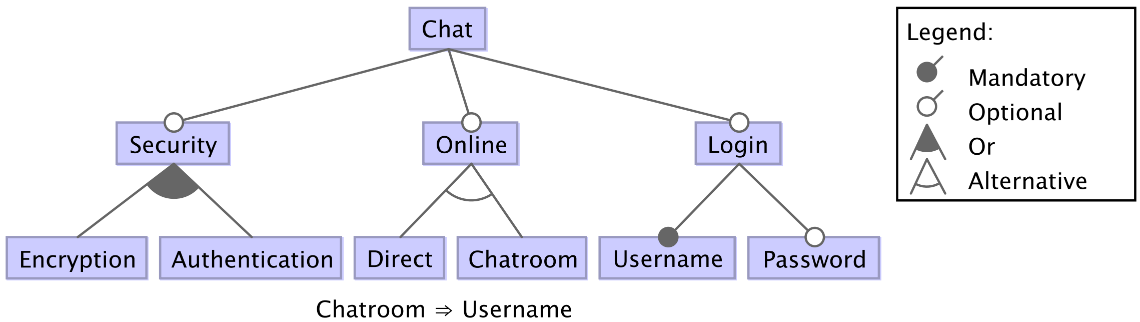

Figure 2.3: Example of a feature diagram, representing a small Chat product line.

In most cases, features are not independent of one another, but have certain interdependencies that must be considered in order to derive a correctly working product. All dependencies between different features are represented by a feature model [CE00]. For example, features can be mutually exclusive, such as features that include source code for different operating systems. Furthermore, features can be dependent on another, such as a feature that changes the appearance of an application relies on a feature that implements a graphical user interface.

There exist multiple representations for feature models, which have their individual advantages [CE00]. In this thesis, we consider the two most popular representations, feature diagrams and propositional formulas, and describe them in more detail in the next sections.

2.2.1 Feature Diagram

A popular, graphical representation for feature models is a *feature diagram*. A feature diagram consists of a hierarchical tree structure with one root feature at the top [CE00, KCH+90]. Feature dependencies are modeled by the arrangement of features within the tree structure, special edge types, and additional cross-tree constraints. Through the feature diagram's hierarchical structure, feature diagrams offer a proper overview of all features in a feature model and their dependencies. Thus, they provide a good readability to humans. We show an example of a feature diagram in Figure 2.3. Here, we illustrate the feature model of a small Chat product line, which represents multiple variants of a basic chat client. In sum, the model consists of ten features `Chat`, `Security`, `Encryption`, `Authentication`, `Online`, `Direct`, `Chatroom`, `Login`, `Username`, and `Password`. The feature diagram consists of all basic constructs that can be used to express dependencies between features. These constructs are parent-child relationships, optional features, mandatory features, feature groups, and cross-tree constraints. Parent-child relationships are the fundamental construct for the hierarchical tree structure. Each feature relies on its parent and, thus, can only be part of a product, if its parent is there as well. For instance, all products that contain the feature `Password`, also contain its parent feature `Login` and, therefore, also `Login`'s parent feature `Chat`. The feature `Username` is mandatory, which means that if its parent feature `Login` is part of a product, then `Username` is also contained in it. By contrast, the optional features `Security`, `Online`, `Login`, and `Password` have no such relationships to their corresponding parent features. The features `Encryption` and `Authentication` are part of an OR-group, which means that if their parent feature `Security` is part of a product, then at least one of them must be in the product as well. The features `Direct` and `Chatroom` are part of an alternative-group. If their parent feature `Online` is contained in a product, then exactly one of them must also be present. Another element of feature diagrams are cross-tree constraints, which specify additional constraints that cannot be represented by the current tree structure. For instance, the cross-tree constraint `Chatroom` ⇒ `Username` is depicted at the bottom of the diagram.

2.2.2 Propositional Formula

Every feature model can be represented by a propositional formula [Bat05]. In a propositional formula, each feature is represented by one boolean variable. Feature dependencies are modeled by connecting the variables with different logical operators. Propositional formulas are often used as input for algorithms that modify or analyze feature models, because most tasks can be reduced to well-known problems in boolean algebra.

A feature model represented by a feature diagram can always be transformed into a propositional formula. Our example feature model in Figure 2.3 can be written as the formula given in Figure 2.4.

$$
\begin{aligned}
Chat \wedge &\qquad \textit{root feature} && (1.1)\\
(Encryption \vee Authentication \Rightarrow Security) \wedge &\qquad \textit{parent-child dependency} && (1.2)\\
(Encryption \vee Authentication \vee \neg Security) \wedge &\qquad \textit{OR-group} && (1.3)\\
(Direct \vee Chatroom \Rightarrow Online) \wedge &\qquad \textit{parent-child dependency} && (1.4)\\
(Direct \vee Chatroom \vee \neg Online) \wedge &\qquad \textit{alternative-group} && (1.5)\\
(\neg Direct \vee \neg Chatroom) \wedge & && \\
(Username \vee Password \Rightarrow Login) \wedge &\qquad \textit{parent-child dependency} && (1.6)\\
(Login \Rightarrow Username) \wedge &\qquad \textit{mandatory feature} && (1.7)\\
(Chatroom \Rightarrow Username) &\qquad \textit{cross-tree constraint} && (1.8)
\end{aligned}
$$

Figure 2.4: Propositional formula of the Chat feature model.

All parent-child relationships can be expressed in a propositional formula with an implication (e.g., Equation 1.2). Consequently, mandatory features can be expressed as an implication as well (e.g., Equation 1.7). As the name suggests, OR-groups represent a logical OR (i.e., a disjunction) between all features in the group, hence they can be expressed using disjunctions and negations (e.g., Equation 1.3). Alternative-groups are similar to OR-groups, but with one additional rule, all children exclude each other, which can be written as set of pairwise disjunctions (e.g., Equation 1.5).

Often, applications require certain representations of propositional formulas. For instance, the formula given above, in Figure 2.4, contains multiple logical operators, such as implication ($\Rightarrow$), disjunction ($\vee$), conjunction ($\wedge$), and negation ($\neg$). However, often, algorithms that work on feature models require the *conjunctive normal form* (CNF) of a propositional formula. Another useful representation is an implication graph, which is one way to combine the domains of boolean algebra and graph theory. In our thesis, we rely on both, CNFs and implication graphs. Thus, we now describe the two concepts in more detail.

Conjunctive Normal Form

A CNF contains only the logical operators disjunction, conjunction, and negation in a certain order. It consists of a conjunction of clauses that consists of a disjunction of single positive or negative variables. Negation is only allowed on single variables and not for whole clauses or the entire formula. Every propositional formula can be written in CNF [Das05]. For instance, the constraint $Chatroom \Rightarrow Username$ can also be written as $\neg Chatroom \vee Username$.

In most cases, CNFs are easy to create from feature diagrams, since a CNF simply resembles a collection (conjunction) of constraints (clauses) that must be fulfilled. However, transforming complex cross-tree constraints can be a time consuming task, since this is, in general, an NP-complete problem. When we transform the complete propositional formula given in Figure 2.4, we get the CNF depicted in Figure 2.5.

$$Chat \wedge \tag{2.1}$$
$$(\neg Encryption \vee Security) \wedge (\neg Authentication \vee Security) \wedge \tag{2.2}$$
$$(Encryption \vee Authentication \vee \neg Security) \wedge \tag{2.3}$$
$$(\neg Direct \vee Online) \wedge (\neg Chatroom \vee Online) \wedge \tag{2.4}$$
$$(Direct \vee Chatroom \vee \neg Online) \wedge \tag{2.5}$$
$$(\neg Direct \vee \neg Chatroom) \wedge \tag{2.6}$$
$$(\neg Username \vee Login) \wedge (\neg Password \vee Login) \wedge \tag{2.7}$$
$$(\neg Login \vee Username) \wedge \tag{2.8}$$
$$(\neg Chatroom \vee Username) \tag{2.9}$$

Figure 2.5: Propositional formula of the Chat feature model in CNF.

Implication Graph

An implication graph is a special data structure to represent propositional formulas [APT79]. It is a directed graph, whose nodes represent the variables of a formula and each edge an implication from one variable to another. Every variable is mapped to exactly two nodes. The first node represents the positive and the second one the negative form of a variable. Hence, the number of nodes in an implication graph is twice the amount of variables in the formula. If the value of one variable implies a certain value for another variable, this is represented by an edge.

To express a propositional formula as an implication graph, it must be transformable into a 2-CNF, which is a formula in CNF, where all clauses consist of at most two variables. All clauses in a 2-CNF are equivalent to a logical implication, as we demonstrated above. Thereby, the entire formula can be written as a set of implications, which can be mapped to edges in the implication graph. By contrast, a constraint with more than two variables, such as $(Direct \vee Chatroom \vee \neg Online)$, cannot be expressed as set of implications between only two variables and, thus, it is not possible to create a corresponding 2-CNF. Therefore, not every propositional formula can be expressed by an implication graph.

There already exists extension to implications graphs that allow the usage of arbitrary propositional formulas, such as the inclusion of conjunction nodes [TGH97] or the expansion to hypergraphs [CW07]. However, in our thesis, we focus on ordinary implication graphs and propose an own extension that suits our needs best.

2.3 Product-Line Configuration

In general, a product line consists of multiple features that can be part of a product. The process of configuring a product of an SPL is the decision of which features are part of a certain product with respect to the feature dependencies specified by the SPL's feature model [PBvdL05]. A *configuration* is the (intermediate) result of the configuration process and specifies for each feature in the SPL whether it is included or not. A configuration is called *valid*, if it satisfies all dependencies of a given feature model. By contrast, a configuration that contradicts at least one dependency is called *invalid*.

A straight-forward approach to configure a product line is to manually define a configuration, which specifies all features that are part of one product. We then provide a configuration for each product that we want to derive. With this approach we configure all features of the feature model at once. Of course, this approach can lead to invalid configurations, since a manually defined configuration is likely to violate at least one feature dependency. Thus, a manual configuration process for all features is unreasonable for large product lines. An alternative approach is a stepwise configuration process, where we configure each feature one-at-a-time. In the following, we describe the procedure of a stepwise configuration process in detail. Furthermore, in the subsequent section, we explain the concept of an interactive configuration process, which includes the propagation of decision implications.

2.3.1 Stepwise Configuration Process

In the stepwise configuration process, we configure all features of an SPL in succession. This is strongly related to *staged configurations*, which is the process of specializing a feature model in consecutive stages to derive a final configuration [CHE05]. Similar to staged configuration, the stepwise configuration process reduces the number of possible decisions with each step and, thus, limits the configuration space.

During the stepwise configuration process, a feature can have one of three possible selection states, *positive* (the feature is selected), *negative* (the feature is deselected), or *undefined* (there is no decision for this feature yet). At the beginning, the selection states of all features are set to undefined. Step by step, we set the selection state of each feature to either positive, which means it is included in the product, or negative, which means it is excluded from the product. The stepwise configuration process is finished if there remains no undefined feature. In addition, we can finish the stepwise configuration process at any given point by assigning a default selection state, such as negative, to all remaining undefined features.

After each configuration step, we get a *partial configuration* that specifies the selection states of all features of the product line. In a partial configuration some features can have an undefined selection state. By contrast, when we finished the configuration process, we get a *full configuration*, in which all features are either selected or deselected. Thus, a full configuration can be considered as a special case of a partial configuration. Contrary to most implementations, we do not omit deselected features, but include both, selected and deselected features, in a full configuration. In the remainder of our thesis, we use the short term configuration to refer to a partial configuration.

A stepwise configuration process may lead to an invalid configuration that does not meet all constraints specified by the feature model. For example, consider our Chat feature model from Figure 2.3. If we select the feature `Direct`, then it is not possible to select `Chatroom` without introducing a conflict in the configuration. However, we might not be aware of that fact and are still able to select `Chatroom`. Not before we test the current configuration for validity, we know about the resulting conflict. In this case, we have to undo the last configuration steps until the introduction of the conflict. Of course a feature selection or deselection can introduce multiple conflicts in the current configuration. Thus, we might need to undo more than one configuration step. To avoid the revocation of configuration steps in the first place, we can use the concept of the interactive configuration process, which we describe in the following.

2.3.2 Interactive Configuration Process

An interactive configuration process means that at no time during a stepwise configuration process the resulting partial configuration is invalid [Men09]. To enforce a valid partial configuration, we propagate every selection state that is implied from the last configuration step. This process is called decision propagation [MBC09, TKB+14]. As a consequence, we never have to undo a configuration step, because it contradicts with the feature model's dependencies. Hence, the resulting configuration process is backtracking-free.

Using an interactive configuration process, we exemplary configure a product of our Chat product line (see Figure 2.3). At first, we deselect the feature `Security`, because we do not need a secure chat application. Through decision propagation the features `Encryption` and `Authentication` are deselected, since it is not possible to select them if their parent feature is already deselected. Next, we select the feature `Chatroom`, because we want to chat with more than one person simultaneously. The selection of `Chatroom` affects many other features. Its parent feature `Online` is selected as well. By contrast, its sibling `Direct` is deselected, due to the alternative-group's constraint. The cross-tree constraint `Chatroom` $\Rightarrow$ `Username` infers the selection of `Username` and, consequently, its parent `Login`. Now, `Password` is the only feature left with an undefined selection state. In this example, we deselect `Password`. In the end, after three configuration steps, we have a full configuration (i.e., no undefined features) with the selected features `Chat`, `Online`, `Chatroom`, `Login`, and `Username`.

A way to realize decision propagation is the application of the feature-model dependency analysis. The results of this analysis are equal the outcome of decision propagation. In the following section, we explain the analysis, among others, in more detail.

2.4 Feature-Model Analysis

Our concept for decision propagation relies on certain properties of a feature model. In order to determine these properties, we use several automated feature-model analyses, which we want to describe in this section. Thus, we present the analyses of *void* feature models, *variant* features, and *atomic sets* and the *dependency analysis* [BSRC10]. Moreover, we briefly present an implementation concept for each these analyses.

We use the feature-model representation of propositional formulas to explain the feature-model analyses and their implementation concepts. Thereby, we are able to reduce all analyses to one or more instances of the *satisfiability problem*. The satisfiability problem (SAT) represents the question whether there is a variable assignment that satisfies a given propositional formula. For example, consider the following propositional formula $Chat \wedge (Chat \Rightarrow Login)$. This formula is satisfiable, because it has the satisfying variable assignment, $(Chat = true, Login = true)$. By contrast, the propositional $Chat \wedge (Chat \Rightarrow Login) \wedge \neg Login$ has no satisfying variable assignment. In terms of product-line configuration, a satisfying variable assignment represents a valid, full configuration of a product line.

The general satisfiability problem is NP-complete and, therefore, likely not be solved in polynomial time [Coo71]. However, there exist algorithms designed for solving instances of the satisfiability problem by using certain heuristics to find a solution in reasonable time for most cases. These algorithms are called satisfiability solvers. Additionally, Mendonça et al. point out that the satisfiability problem does scale well for most feature models [MWC09]. Since we reduce the presented feature-model analyses to SAT, we are able to use satisfiability solvers in the actual implementation of all shown analyses.

2.4.1 Void Feature Model

An important question is whether a given feature model is valid, which means that it represents at least one valid product. By contrast, we call a feature model *void*, if it represents no product [Bat05]. A feature model can be void if it either has no features or if it contains a contradiction in its feature dependencies. Consider we would add the constraint $\neg Chat$ to our feature model Chat (cf. Figure 2.3). Since the feature `Chat` must be contained in every valid product, we would create a contradiction within the feature model and, thus, it would be void.

The void feature-model analysis is of high importance, since we cannot use void feature models in the application-engineering process. Furthermore, due to their definitions, all of the following analyses can only be performed on non-void feature models [STSS13].

If we use a propositional formula as feature-model representation, we can easily test for validity by solving the corresponding satisfiability problem. If there is a satisfying variable assignment for the propositional formula, the feature model represents at least one product and, thus, is not void. Otherwise, if there is no variable assignment that satisfies the feature dependencies, the feature model is void.

2.4.2 Variant Features

In general, in a full configuration, a feature can be selected or deselected in a certain product. Features that can configured both ways are called *variant features* [BSRC10]. In contrast, there are *dead features* and *core features*, which have only one possible selection state. A features is called core, if and only if it is part of every possible product of an SPL [BSRC10, TRC09]. Thereby, its only possible selection state is positive. Contrarily, a feature that is part of no product is called dead [BSRC10, TBC06]. Thus, it can only have the selection state negative.

In our Chat feature model (cf. Figure 2.3), all features but `Chat` are variant features. `Chat` is the root feature of the given feature diagram and, thus, contained in every product. Hence, `Chat` is a core feature by default. Hypothetically, if we would add the constraint $Chat \Rightarrow Chatroom$, the analysis would identify five core features, `Chat`, `Login`, `Username`, `Online`, and `Chatroom`, and one dead feature, `Direct`.

The knowledge of variant, core, and dead features can help to enhance the configuration process and to detect flaws in a feature model. Dead features can always be seen as defect, since they have no conceivable purpose. On the other hand, core features can naturally occur in a feature model. The implementation of core features can be used to provide the common source code base for all possible products. For instance, the root feature of a feature diagram is always a core feature. Furthermore, in order for our approach to work correctly, we need to know all variant features of a feature model.

If the feature model is given in form of a propositional formula, all dead and core feature can be determined by using the following approach. For each feature, we set the truth value of the corresponding variable to $false$. If the formula is not satisfiable, then the feature is core. Analogous, if the variable's value was set to *true* and the formula cannot be satisfied, then the feature is dead. After determining the dead and core features of a feature model, all remaining features must be variant features. This analysis only applies if the formula was satisfiable in the first place (i.e., the feature model is not void). In other words, if a feature model represents no products, it is not feasible to ask, whether a feature is part of *every* product.

2.4.3 Dependency Analysis

The dependency analysis is the most important analysis for our approach, since it is the basis for decision propagation. The dependency analysis can be seen as a generalization of the core and dead feature analysis. Additionally to a feature model, this analysis also takes a partial configuration as input. Then, the analysis determines all selection states that are implied by the given partial configuration and updates the partial configuration accordingly [BSRC10]. We call the corresponding features of determine selection states *conditionally dead* and *conditionally core* features [BSRC10].

For example, using our Chat feature model (cf. Figure 2.3), the feature `Chatroom` is conditionally dead if the given partial configuration defines the feature `Direct` as selected. In addition, the parent feature `Online` would be conditionally core.

The implementation of the dependency analysis using satisfiability solvers is similar to the core and dead analysis, but with one exception. Before testing every feature, we assign the corresponding truth values to all variables in the propositional formula, whose of corresponding features are selected or deselected in the given partial configuration. Afterwards, we perform the same procedure as if determining core and dead features. For each undefined feature, we set the truth value of its corresponding variable to either $false$ or *true* and check whether the propositional formula is still satisfiable.

2.4.4 Atomic Sets

An atomic set is a maximal set of features that fulfills the following condition. In each valid, full configuration all feature in the set are either all selected or all deselected. For certain algorithms and analyses, features in an atomic set can be treated as a single unit [BSRC10].

Regarding our Chat feature model (cf. Figure 2.3), an example for an atomic set are the features `Login` and `Username`. It is not possible to only include one of them in a valid product. They are either both present or both absent.

Since features in an atomic set have an equal selection state for each configuration, it is possible to combine all of them in one feature that represents all of them at once. Thereby, we effectively decrease the number of features in a feature model and, thus, limit the configuration space. Therefore, we can use atomic sets to reduce the complexity of certain analyses and the configuration process [Seg08, ZZM04]. We further discuss this extension in Chapter 8.

A simple implementation concept of this analysis using SAT is the following, which tests each pair of features, whether they are in an atomic set or not. For each pair, we set the truth value of one variable to $true$ and the truth value of the other variable to $false$, if the propositional formula is still satisfiable under this condition, the features cannot be in an atomic set. Otherwise, we repeat the test with inverted truth values. If the formula is not satisfiable in the second test as well, then the features must be in the same atomic set. Since each pair of features is tested, this approach needs to solve about n^2 satisfiability problems, where n is the number of features. Hence, its computational effort for large feature models is quite high.

2.5 Summary

In this chapter, we presented the concept of software product line engineering and focused on two vital aspects, namely feature modeling and configuration. In addition, we presented certain feature-model analyses that are related to our approach. We described feature modeling as the process of creating a feature model that represents all features of an SPL and their interdependencies. Additionally, we presented feature diagrams and propositional formulas as representation for feature models and explained how implication graphs can be used to express simple feature dependencies. We introduced the stepwise configuration process and based on that the interactive configuration process, which relies on decision propagation to update the resulting partial configuration for each step. Finally, we explained several feature-model analyses, such as void feature models, variant features, dependency analysis, and atomic sets, and demonstrated implementation concepts for each analysis by applying the satisfiability problem.

3. Concept

In this chapter, we introduce the *configuration assistant*, our new approach for automated decision propagation during an interactive configuration process. For this, we propose an extension for implication graphs that we call *feature graph* and use it to express the dependencies of a feature model. During the decision propagation, our configuration assistant traverses a feature graph to efficiently determine all forced selection states for the current partial configuration. First of all, we give an overview of our approach and explain the general idea behind it. Then, we describe our new data structure, the feature graph, and demonstrate its application to the automated decision propagation during the interactive configuration process. Finally, we explain the feature graph's construction process.

3.1 Overview of the Configuration Assistant

Our new approach, the configuration assistant, is designed for an interactive configuration process with the main goal of reducing the computation time of the automated decision propagation as much as possible. For this, we try to avoid using the *complex propagation test* for determining each feature's selection state during the automated decision propagation. The complex propagation test refers to an arbitrary implementation of the dependency analysis presented in Chapter 2 using satisfiability solvers. Although we assume that the complex propagation test always finds the correct solution, it can be very time consuming, since it is solving multiple NP-complete problems. In detail, if there are n undefined features in the current partial configuration, the complex propagation test has to solve $2 \cdot n$ problems.

In the following, we describe the basic principle of our approach and why it can be useful for improving the performance of the interactive configuration process. Furthermore, we explain the usage of implication graphs to express feature dependencies and our associated extension, the feature graph.

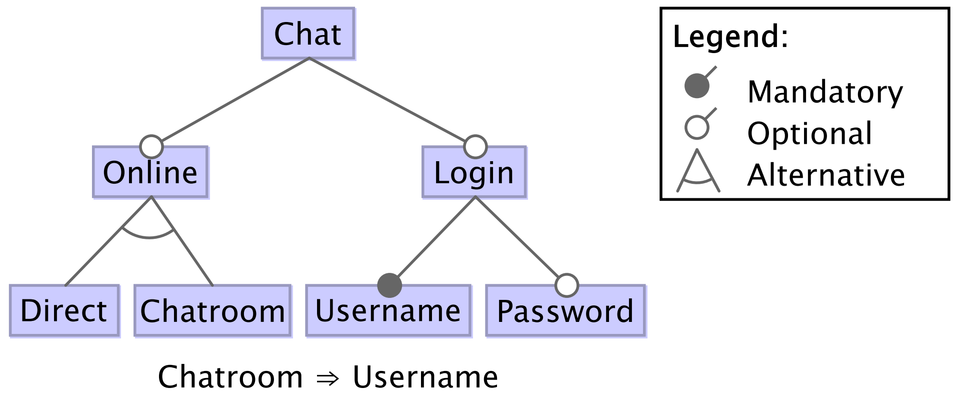

Figure 3.1: Reduced feature model of the Chat product line.

3.1.1 Basic Principle

Our approach is based on two observations on the interactive configuration process. First, many forced selection states that are found during the decision propagation originate from simple feature dependencies (i.e., feature dependencies that can be expressed in 2-CNF). Second, many features that are not affected by the decision propagation are independent of the currently configured feature (i.e., there are no dependencies between them). A good example for these observations are feature dependencies that originate from a feature diagram's tree structure. Parent-child dependencies, as well as mandatory features, can be expressed with a logical implication between two features. We consider logical implications as simple feature dependencies, since they can be easily evaluated during the decision propagation. Moreover, features in different subtrees are always independent of each other if there exist no cross-tree constraints connecting both trees.

In each configuration step of the interactive configuration process, we make a decision that changes the selection state of one feature. Afterwards, the automated decision propagation is used to update the remaining undefined features of the current partial configuration. By using the two observations mentioned above, we are able to categorize the potential selection states of all undefined features dependent on the made decision and divide them into one of three groups. The selection state of an undefined feature is either *directly dependent*, *indirectly dependent*, or completely *independent* on the latest decision. A direct dependency means that we can derive a feature's selection state directly from the latest decision, because both features are connected via one or more logical implications (i.e., simple feature dependencies). By contrast, an indirectly dependent selection state cannot be determined without considering the selection state of other features besides the one configured in the latest configuration step. An independent selection state is not affected by the latest decision step at all.

To exemplify our statements in this chapter, we use a smaller version of the Chat feature model from Chapter 2. We depict the corresponding feature diagram in Figure 3.1. By examining a configuration step involving the feature `Chatroom`, we can show all three mentioned categories of selection-state dependencies. Assume, we start an interactive configuration process and, as first step, we assign a positive selection state to `Chatroom` (i.e., select it). We can see that some selection states of other features are directly dependent on this decision. These are the positive selection states of `Online`, `Username`, and `Login` and the negative selection state of `Direct`. A negative selection state is implied for `Direct`, due to the alternative-group with `Chatroom`. In addition, a positive selection state is implied for `Online` (parent of `Chatroom`), `Username` (via a cross-tree constraint), and `Login` (parent of `Username`). We can derive each of these selection states directly from the positive selection state of `Chatroom` without considering the selection state of other features. Furthermore, we can see that the positive and negative selection states of feature `Password` are independent from our decision. By contrast, if we start the interactive configuration process by assigning a negative selection state to `Chatroom` (i.e., deselecting it), we can see indirect dependencies for other selection states. In total, there are six selection states that are indirectly dependent on this decision, the positive selection states of `Direct`, `Online`, `Username`, and `Login` and the negative selection states of `Direct` and `Online`. All these selection states might be implied after our made decision, but they do not directly dependent on the deselection of `Chatroom`. For instance, a negative selection state of `Chatroom`'s parent feature `Online` is forced, if `Online`'s other child, `Direct`, is also deselected in the current partial configuration. Likewise, a positive selection state of `Direct` is forced if `Online` is selected. However, we cannot determine these selection states without considering at least one other feature besides the currently configured one (e.g., `Chatroom`).

For automated decision propagation, we can use the categorization of other features' selection states to reduce the amount of complex-propagation-test applications. Both, directly dependent and independent selection states of features can be determined by just considering the currently configured feature. Only the indirectly dependent selection states require more extensive computations. Therefore, a categorization of the possible selection states of all undefined features, based on the current configuration step, means that we are able to save computational effort and, hence, improve the overall performance of the configuration process.

3.1.2 Usage of Implication Graphs

A first approach to realize the categorization described above is to model the feature dependencies as an implication graph. As we stated in Chapter 2, all propositional formulas that are convertible into 2-CNF can also be written as an implication graph. However, most feature models contain feature groups or complex cross-tree constraint, which prevents us from representing the entire feature model as a 2-CNF propositional formula. Therefore, in most cases, we cannot use ordinary implication graphs to express the dependencies of a feature model. However, we can exclude those parts of the feature model that cannot be written in 2-CNF and use the remaining constraints to build a reduced implication graph. Except for feature groups and complex cross-tree constraints, we can convert every construct of a feature diagram into a 2-CNF statement.

For instance, if we only use the 2-CNF clauses of a CNF, we could create a partial implication graph. From this partial graph, we are able to derive certain information that are useful for the decision propagation. Naturally, this graph does not fully represent the original model, since we excluded all other clauses. However, in Chapter 1, we specified secondary conditions for our approach that demand an exact and complete result of the decision propagation (cf. condition 1). Therefore, we also need the remaining dependencies of the feature model for a complete decision propagation and, thus, we propose an extension for implication graphs that is able to hold the necessary information. We call the resulting data structure a *feature graph*.

A feature graph is based on an ordinary implication graph and, thus, it is also a directed graph with nodes that represent the selection states of single features. The difference between our graph and an ordinary implication graph is that we use two different kinds of edges, which we call *strong connections* and *weak connections*. Strong connections represent a direct dependency from one node to another. By contrast, weak connections represents an indirect dependency. Furthermore, if we traverse through the feature graph, starting from node A and are not able to reach a certain node B, then these two nodes, A and B, are independent of one another. Thus, the feature graph exactly holds those information that are required by our configuration assistant.

We can divide our approach into two consecutive phases, the *initialization phase*, where the feature graph is constructed and the *configuration phase*, where the feature graph is used for the automated decisions propagation. In the following section, we explain both phases in detail. At first, we demonstrate how we utilize the information, represented by our feature graph, to improve the decisions propagation in the configuration phase. Afterwards, we present the initialization phase of our approach, which consist of constructing a feature graph based on a given feature model.

3.2 Configuration Phase

We now demonstrate how our feature graph is used during the interactive configuration process. To comprehend to our approach, in Figure 3.2, we depict a complete feature graph for our small Chat product line (see Figure 3.1). The feature graph consists of two nodes for each variant feature in the feature model. Each node represents either the positive or negative selection state of the corresponding feature (e.g., $Chatroom$ and $\neg Chatroom$). The dependencies between the nodes (i.e., selection states) are represented by the graph's strong and weak connections.

The interactive configuration process consists of consecutive configuration steps with subsequent decisions propagation. In our approach, we realize the decision propagation with a *selection algorithm* that uses the information of our feature graph. For each configuration step, our selection algorithm traverses through the feature graph to determine selection states of yet undefined features. The decision, made in one configuration step, can be mapped to the corresponding node in the feature graph. For instance, when we deselect the feature `Chatroom`, this decision is mapped to the feature-graph node that represents the negative selection state of `Chatroom` (i.e., $\neg Chatroom$). This node represents the starting point of the following traversal. By performing a depth-first search

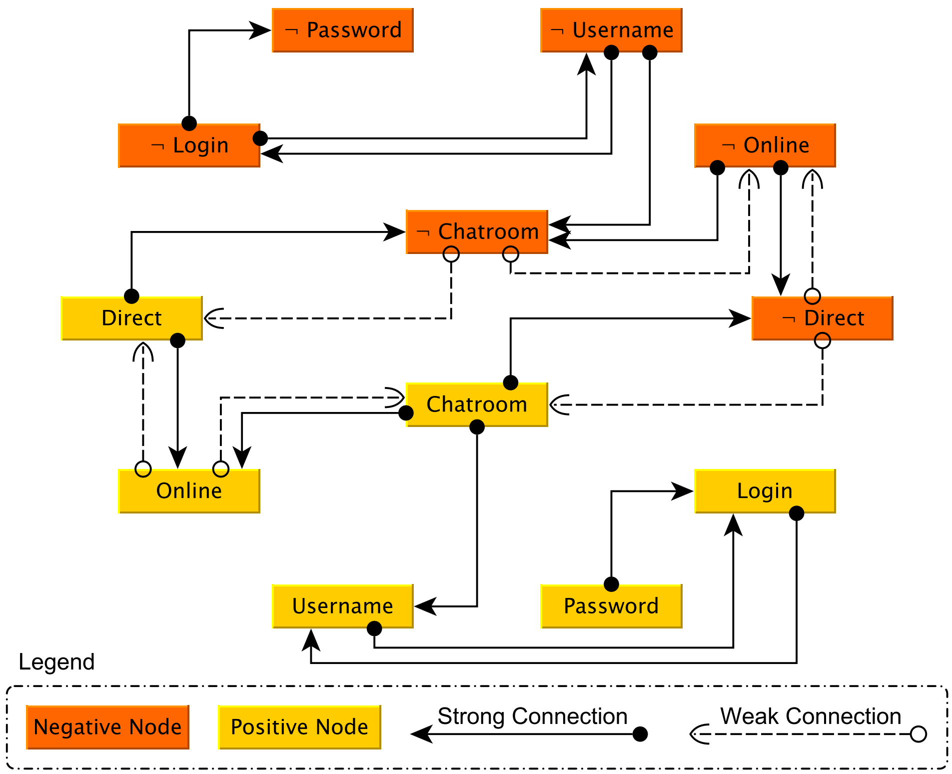

Figure 3.2: Feature graph for the Chat feature model (cf. Figure 3.1).

(DFS), our selection algorithm visits every node that can be reached from the starting node via one or more connections. For each reached node, the algorithm examines the connection types in the path from the starting node. If the path only consists of strong connections (i.e., a *strong path*), our algorithm immediately knows the selection state of the corresponding feature. By contrast, if the path contains at least one weak connection (i.e., a *weak path*), our algorithm has to determine the selection state with the complex propagation test. For all nodes that are not connected to the starting node, the algorithm has to do nothing.

In Algorithm 1, we show pseudo source code for the general selection algorithm. This algorithm realizes the feature-graph traversal recursively. The algorithm starts with the procedure `decisionPropagation` (Line 1). As parameters, the algorithm passes the feature graph and information from the latest configuration step, which feature was configured and which selection state (positive or negative) was set. At first, the algorithm retrieves the node in the feature graph that maps to the latest decision (Line 2). Then, it traverses along all strong paths (Lines 3, 6–15) and sets the corresponding selection states (Lines 11, 28–34). After that, it traverses along the weak paths (Lines 4, 16–27) and tests the found selection states via the complex propagation test (Line 21). If the complex propagation test is successful, the algorithm sets the corresponding selection state (Lines 22, 28–34). In Section 3.3.2 we introduce the concept of transitive closure, which adds all transitive edges to the feature graph. Without anticipating too much, we can say that this approach limits the DFS' search depth to one level, i.e.,

Algorithm 1 Configuration Assistant - General Selection Algorithm: After each configuration step, `decisionPropagation` is called with according parameters.

```
procedure DECISIONPROPAGATION(featureGraph, feature, selectionState)
    node ← featureGraph.GETNODE(feature, selectionState)
    TRAVERSESTRONG(node, ∅)
    TRAVERSEWEAK(node, ∅)
end procedure

procedure TRAVERSESTRONG(node_start, nodes_visited)
    nodes_visited ← nodes_visited ∪ {node_start}
    nodes_adjacent ← node_start.strongNeighbors \ nodes_visited
    for all node_neighbor ∈ nodes_adjacent do
        if node_neighbor.feature.selectionState = UNDEFINED then
            CONFIGURE(node_neighbor)
        end if
        TRAVERSESTRONG(node_neighbor, nodes_visited)
    end for
end procedure

procedure TRAVERSEWEAK(node_start, nodes_visited)
    nodes_visited ← nodes_visited ∪ {node_start}
    nodes_adjacent ← node_start.allNeighbors \ nodes_visited
    for all node_neighbor ∈ nodes_adjacent do
        if node_neighbor.feature.selectionState = UNDEFINED then
            if COMPLEXTEST(node_neighbor) then
                CONFIGURE(node_neighbor)
            end if
        end if
        TRAVERSEWEAK(node_neighbor, nodes_visited)
    end for
end procedure

procedure CONFIGURE(node)
    if node.isPositive then
        node.feature.selectionState ← POSITIVE
    else
        node.feature.selectionState ← NEGATIVE
    end if
end procedure
```

it only has to visit the direct neighbors of the starting node. Thereby, we are able to simplify the selection algorithm. We present the corresponding pseudo source code in

Algorithm 2 Configuration Assistant - Simplified Selection Algorithm.

```
procedure decisionPropagation(featureGraph, feature, selectionState)
    node ← featureGraph.getNode(feature, selectionState)
    traverseStrong(node)
    traverseWeak(node)
end procedure

procedure traverseStrong(node_start)
    for all node_neighbor ∈ node_start.strongNeighbors do
        if node_neighbor.feature.selectionState = UNDEFINED then
            configure(node_neighbor)
        end if
    end for
end procedure

procedure traverseWeak(node_start)
    for all node_neighbor ∈ node_start.weakNeighbors do
        if node_neighbor.feature.selectionState = UNDEFINED then
            if complexTest(node_neighbor) then
                configure(node_neighbor)
            end if
        end if
    end for
end procedure
```

Algorithm 2. Nevertheless, in Section 3.3.2, we also introduce an alternative concept, transitive reduction, which relies on the general selection algorithm.

We exemplify the functionality of the simplified selection algorithm (i.e., Algorithm 2) with the help of our Chat feature model (see Figure 3.1). In order to use the simplified selection algorithm, we have to apply transitive closure to the feature graph depicted in Figure 3.2. We later explain the procedure of transitive closure in more detail (see Section 3.3.2), for now, we just consider the resulting feature graph, which we visualize in Figure 3.3. As first configuration step, we manually select the feature `Online`. We now look at all nodes that can be reached from the starting node *Online*, which represents the positive selection state of feature `Online`. We can find connections in the graph that lead to the nodes *Direct*, *Chatroom*, *Username*, *Login*, $\neg$*Direct*, and $\neg$*Chatroom*. Thus, we have to determine the corresponding selection states. Note that we do not need to consider other nodes such as *Password* or $\neg$*Login*, nor any other nodes that cannot be reached from *Online*. Since there are weak connections on all found paths, we need to use the complex propagation test to compute all forced selection states. When we apply the complex propagation test, we find out that there is no other feature that has to be selected or deselected in this configuration

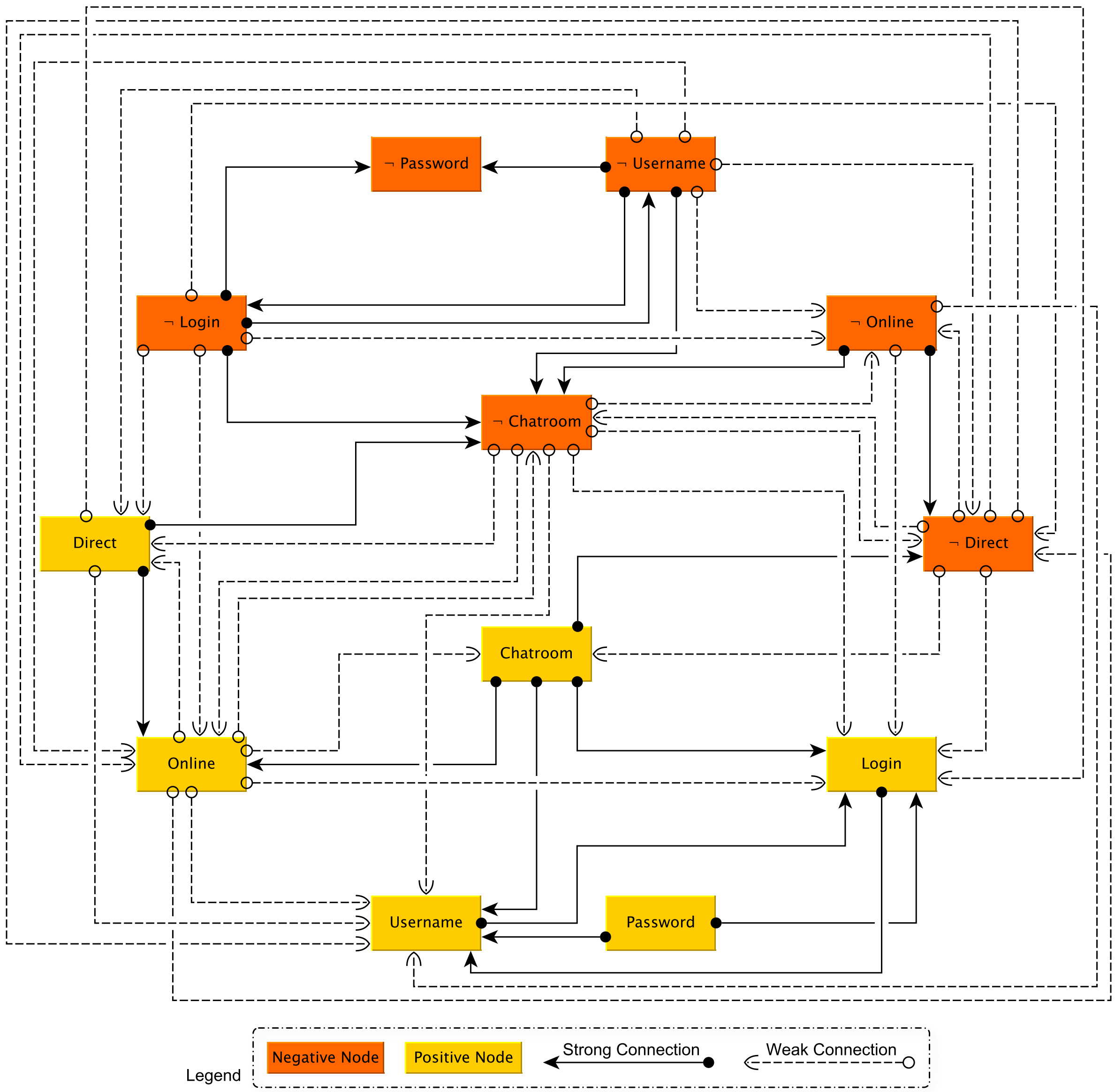

Figure 3.3: Feature graph for the Chat feature model (cf. Figure 3.1) after feature-graph restructuring (using transitive closure).

step. In the next configuration step, we manually deselect the feature `Login`. When we look at the feature graph, we see that we have strong connections from $\neg Login$ to $\neg Password$, $\neg Username$, and $\neg Chatroom$. In addition, we have weak connections to $Direct$, $\neg Direct$, $Online$, and $\neg Online$. However, since we already know the selection states for the features `Online`, `Login`, `Chatroom`, `Username`, and `Password`, we only have to compute whether `Direct` has to be selected, deselected, or stays undefined. By using the complex propagation test, we find out that we have to select `Direct`. We now have a full and valid configuration of our example product line, which includes the features `Online` and `Direct`.

3.3 Initialization Phase

Before we can use our new approach to configure a software product line, we need to build a feature graph by extracting feature dependencies of the product line's feature model. Our approach creates a feature graph for a given feature model in its initialization phase, which consists of three major steps. The first step is the computation of all variant features of the given feature model, which form the basis for the feature graph's nodes. In the second step, all feature dependencies from the feature model are converted into edges between the nodes of our feature graph. As last step, the created feature graph is restructured by either removing or adding transitive edges. The intention behind the last step is to increase the feature graph's efficiency either in terms of memory-space consumption or computational effort during the automated decision propagation. In the following, we explain each step in more detail.

3.3.1 Feature-Graph Construction

The first step of building the feature graph consists of finding all variant features, i.e., all non-core, non-dead features of the given feature model (cf. Section 2.4.2). Since the selection states of core and dead features are fixed, we do not need to consider these features in the automated decision propagation. By reducing the total number of features contained in the graph, we are able to save memory space. Additionally, we might be able to derive several strong connections if non-variant features are contained in a feature group. Moreover, including dead or core features in the feature graph would cause problems later on, when we are determining transitive connections.

In particular, we calculate all core and dead features and remove them from the total set of features. All remaining features are variant features and are used to create the nodes of our feature graph. Each feature is converted into two nodes, where the first node represents the positive and the second node the negative selection state. Considering our example feature model shown in Figure 3.1, we now have a feature graph with 12 nodes and no connections, which we display in Figure 3.4 (Note that the depicted graph is centrally symmetric to provide an easy orientation).

In the second step of building the feature graph, our approach converts all dependencies, specified by the feature model, to connections in the feature graph. The most general way of converting all dependencies is to translate them into a CNF and transform each clause into the corresponding connections. In fact, we use this method for feature models given as a propositional formula and for complex cross-tree constraints in feature diagrams. However, if the feature model is given in form of a feature diagram, we are able to analyze its tree structure to identify dependencies without translating it into a CNF. Moreover, many feature-diagram structures can be written as logical implications and, thus, are converted to strong connections.

For the mapping of structural information from feature diagrams to connections of our feature graph, we use a set of mapping rules. We list the mapping rule for each structure in a feature diagram in Table 3.1 and explain it in the following, in more detail. In total,

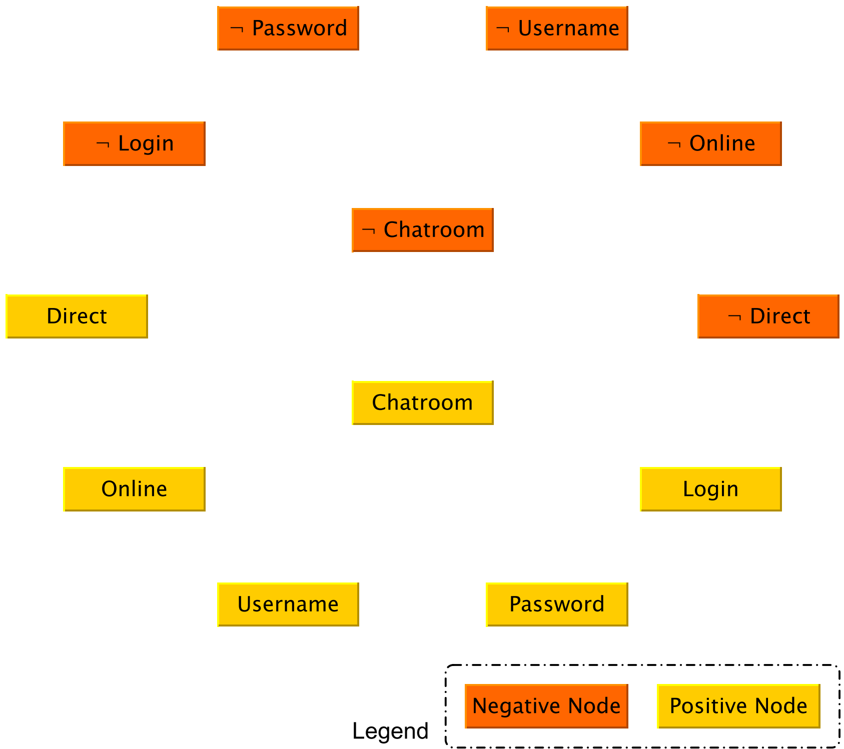

Figure 3.4: Incomplete feature graph for the Chat feature model (cf. Figure 3.1) after determining variant features (containing nodes only).

there are six structures in a feature diagram that we need to consider. We can derive feature dependencies from parent-child relationships, mandatory features, alternative-groups, OR-groups, and complex and simple cross-tree constraints. Note that any type of connection is only added to the feature graph if both involved features are neither dead or core features, because these are not contained in the graph.

At first, we consider the most frequent structure of a feature graph, the parent-child relationship. This structure can be represented by a logical implication from the child to its parent. Hence, they are mapped to a strong connection from the positive node of the child feature to the positive node of its parent. Since an implication $A \Rightarrow B$ is equivalent to the expression $\neg B \Rightarrow \neg A$, we also add a strong connection from the negative parent node to the negative child node. In Figure 3.5, we visualize our example feature graph with all strong connections that result from parent-child relationships (highlighted in blue color).

Next, we add strong connections for all mandatory features in the feature diagram. Similar to parent-child relationships, mandatory features can be represented by a logical implication between parent and child feature. Though, the implication is inverted compared to the parent-child relationship. Thus, we add two strong connections to the feature graph, from the positive node of the parent to the positive node of the child and from the negative node of the child to the negative node of the parent. Considering

Structure (Features)	**Strong Connections**	**Weak Connections**
Parent-Child Relationship (Parent, Child)	$Parent \rightarrow Child$ $\neg Child \rightarrow \neg Parent$	
Mandatory Feature (Parent, Child)	$Child \rightarrow Parent$ $\neg Parent \rightarrow \neg Child$	
Alternative-Group (Parent, Child1, Child2)	$Child1 \rightarrow \neg Child2$ $Child2 \rightarrow \neg Child1$	$Parent \rightarrow Child1$ $Parent \rightarrow Child2$ $\neg Child1 \rightarrow Child2$ $\neg Child1 \rightarrow \neg Parent$ $\neg Child2 \rightarrow Child1$ $\neg Child2 \rightarrow \neg Parent$
OR-Group (Parent, Child1, Child2)		$Parent \rightarrow Child1$ $Parent \rightarrow Child2$ $\neg Child1 \rightarrow Child2$ $\neg Child1 \rightarrow \neg Parent$ $\neg Child2 \rightarrow Child1$ $\neg Child2 \rightarrow \neg Parent$
2-CNF Cross-Tree Constraint ($A \vee B$)	$\neg A \rightarrow B$ $\neg B \rightarrow A$	
Complex Cross-Tree Constraint ($A \vee B \vee C$)		$\neg A \rightarrow B$ $\neg A \rightarrow C$ $\neg B \rightarrow A$ $\neg B \rightarrow C$ $\neg C \rightarrow A$ $\neg C \rightarrow B$

Table 3.1: Rules for mapping feature-diagram structures to connections of a feature graph.

our example feature graph, we add both strong connections for the mandatory feature `Username` and depict the result in Figure 3.6.

Contrary to parent-child relationships and mandatory features, feature groups add weak connections to the graph, since they involve more than two features. For OR-groups, we add a weak connection from the positive node of the parent feature to the positive node of each child feature in the group. In addition, we add a weak connection from the negative node of each child to the positive node of every other child. Moreover, we add a weak connection from the negative node of each child to the positive node of its parent. Alternative-groups are converted exactly like OR-group, but with one extension. For each alternative feature we add a strong connection from its positive node to each negative node of other features in the group. Note that, for both feature groups, we do not need to add strong connections from child to parent nodes, since

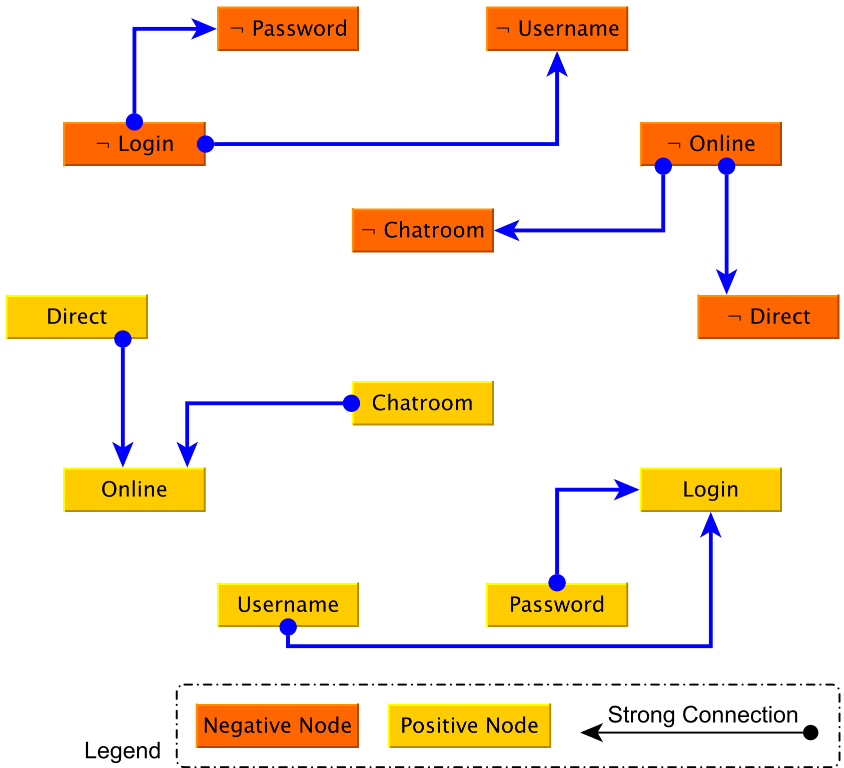

Figure 3.5: Incomplete feature graph for the Chat feature model (cf. Figure 3.1) during feature-graph construction (only parent-child relationships).

these connections were already added via the conversion of parent-child relationships. We display the updated feature graph of our running example in Figure 3.7.

In some special cases, it is possible to identify more strong connections within feature groups or at least reduce the amount of weak connections. As we mentioned above, such a situation can occur if a group contains non-variant features. If a core feature is part of an OR-group, it makes all other variant features in this group optional. Hence, we do not need to add weak connections for this particular group. Another important point is that each dead feature within any feature group can be neglected. Therefore, we count all non-dead features of a feature group. Assuming the parent feature of the group is not dead, there are two different situations in which we are able to add strong connections instead of weak ones. First, if any feature group only contains one variant feature, it can be treated as ordinary mandatory feature. Second, if an alternative-group contains exactly two variant features and the parent feature is core, then, instead of weak, we can add strong connections between both features of the group. Although these special cases seem rather odd and ill-designed, they can actually be found in industrial feature models, since those models often evolve over time and are not completely redesigned.

Finally, we add connections to the graph that result from cross-tree constraints. For cross-tree constraints, as well as for feature models given as propositional formula, we use the method mentioned above. In particular, we translate the whole constraint or formula into a CNF and investigate each clause on its own. For us, the relevant property

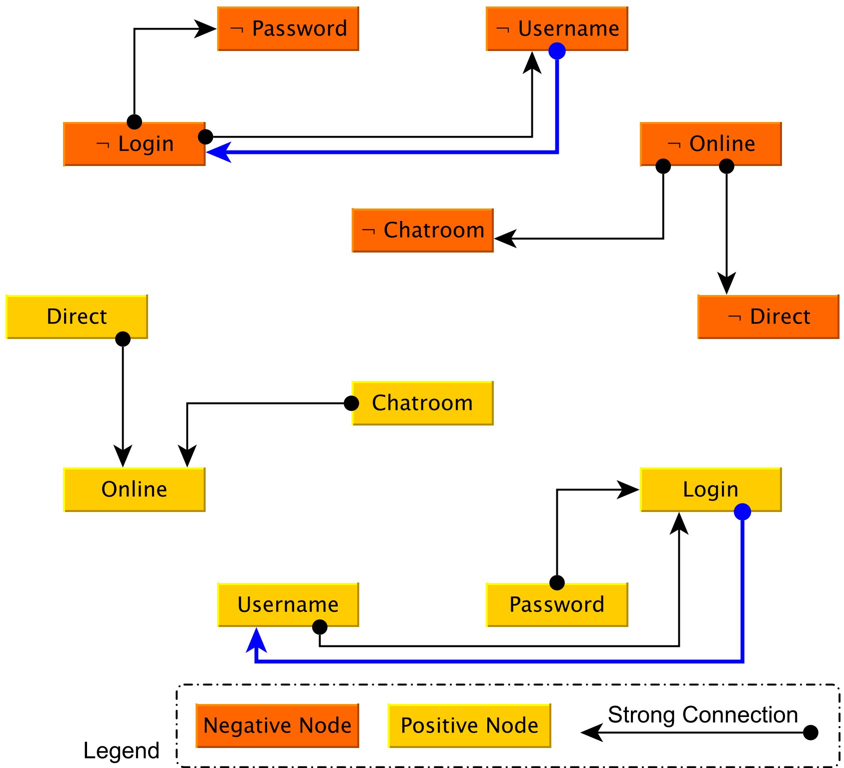

Figure 3.6: Incomplete feature graph for the Chat feature model (cf. Figure 3.1) during feature-graph construction (only parent-child relationships and mandatory features).

is the number of different variables contained in the current clause. If a clause contains exactly two variables it can be written as an implication and, thus, is converted into strong connections in our feature graph. Furthermore, a clause with only one variable represents a core or dead feature and since these features are not part of the feature graph, we ignore those clauses. In contrast, a clause with three or more variables is converted into weak connections. We add a weak connection from the negative to the positive node for each variable 2-tuple in the constraint. If a variable in a clause is present in its negated form, we respectively use the opposite node in the graph. We display the complete example feature graph in Figure 3.8.

Naturally, we try to identify as many strong connections as possible to avoid adding weak connections to our feature graph. In this work, we use a rather simple approach to convert the dependencies and, thus, may not find the maximum amount of strong connections. As we demonstrated in Section 3.2, only weak connections lead to extensive computations. Therefore, we can infer that the fewer weak connections are contained in a feature graph the better is the performance of the automated decision propagation. Feature-diagram structures that lead to weak connections are OR-groups, alternative-groups, and complex constraints (i.e., constraints that cannot be written in 2-CNF). Hence, we assume that these structures have a negative impact on the overall performance.

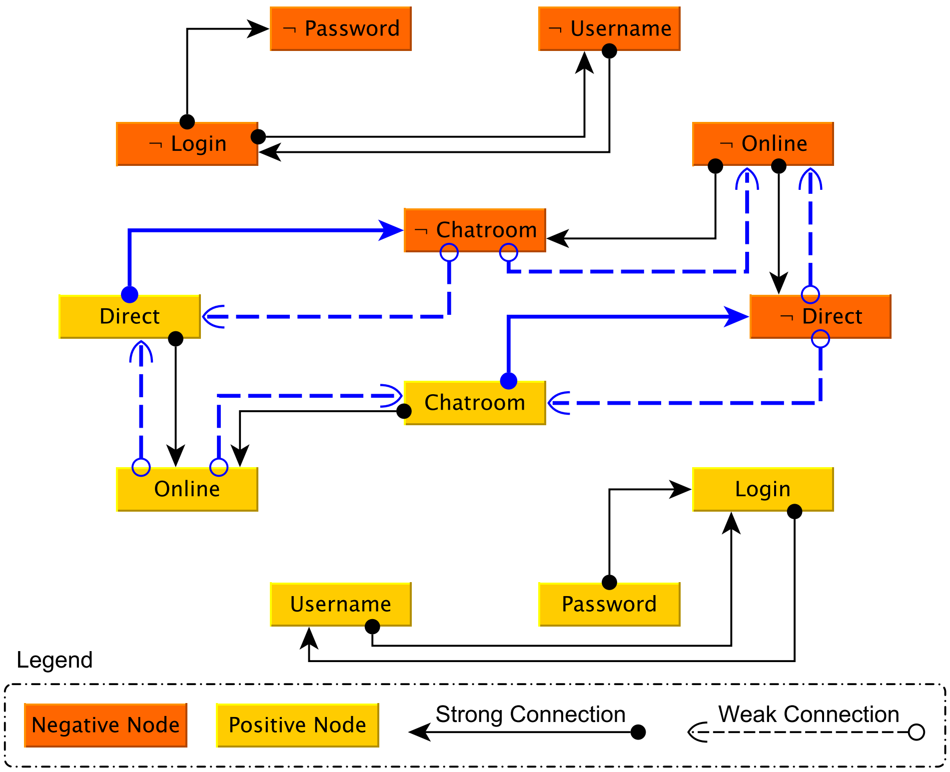

Figure 3.7: Incomplete feature graph for the Chat feature model (cf. Figure 3.1) during feature-graph construction (excluding cross-tree constraints).

3.3.2 Feature-Graph Restructuring

As a last step of the initialization phase, we apply one of two contrary strategies, *transitive closure* or *transitive reduction*, to restructure the current feature graph. That is, we are either adding all possible transitive connections to the graph or reducing them to a minimum. Although these strategies are not mandatory in order for the selection algorithm to work, each strategy has individual advantages and disadvantages regarding memory-space consumption of the graph and computational effort of the initialization and configuration phase. In addition, the chosen strategy has an influence on the selection algorithm, which we already addressed in Section 3.2. In our implementation and, thus, also in our evaluation, we use the first strategy, transitive closure, for various reason, which we explain in the next section.

Transitive Closure

Transitive closure adds all transitive connections to the feature graph. The main advantage of this method is reduction of computational effort during the configuration phase. Since all transitive connections are already contained in the feature graph, there is no need for a complete search in the graph during the decision propagation to find all affected features. It is sufficient to just consider the direct neighbors of the starting node. Therefore, the selection algorithm becomes easier to implement, as we already presented in Section 3.2. This advantage comes at the cost of more computational effort

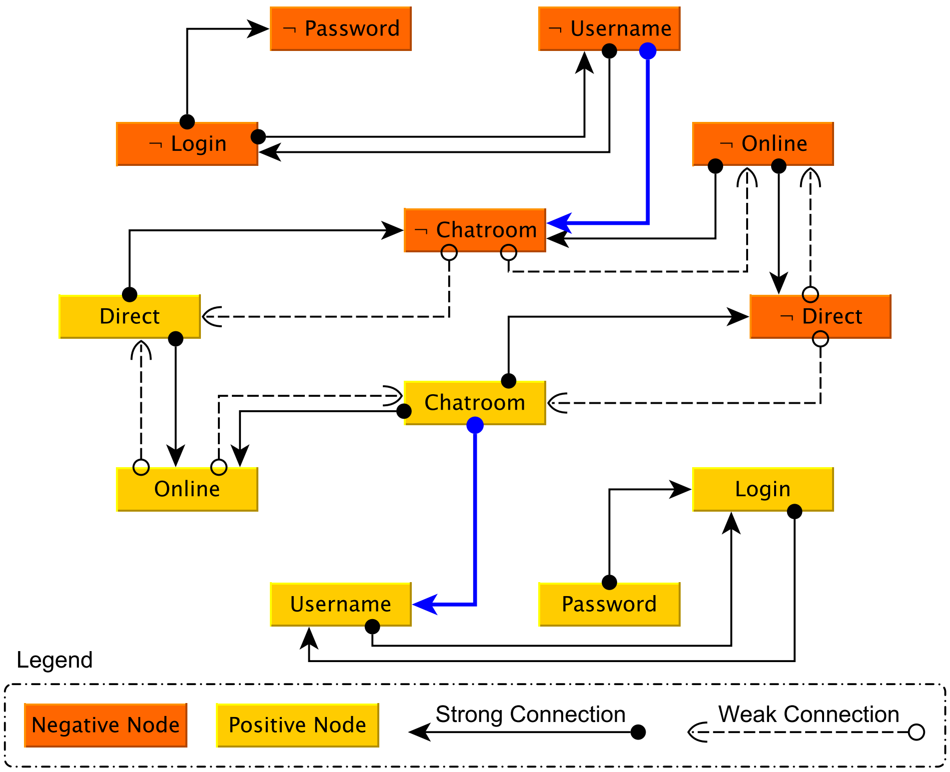

Figure 3.8: Complete feature graph for the Chat feature model (cf. Figure 3.1) after feature-graph construction.

for constructing the feature graph, because the search must be performed during the initialization phase.

To find all transitive connections, we use a *search algorithm* that is based on a DFS. We show the general approach of the search algorithm as pseudo code in Algorithm 3. For each node in the graph, the search algorithm performs a DFS and adds a connection for every found path. At first, the search algorithm only considers strong paths (Lines 5, 15–22). For each found strong path, the search algorithm adds a strong connection to the feature graph (Line 19). Note that the used procedure `addStrongConnection` overrides existing weak connections. Afterwards, the search algorithm adds transitive connections for the remaining weak paths (Lines 11, 23–30). Unlike the previous procedure `addStrongConnection`, the procedure `addWeakConnection` (Line 27) does not override any strong connection. In order to avoid searching the same subgraph twice, the algorithm keeps track of all nodes where the DFS was already performed (Lines 2, 6, 8, 12). However, since we perform a DFS for each feature, we end up with a complexity of $\mathcal{O}(n^3)$, where n is the number of nodes in the graph.

As example, we visualize the results of transitive closure on the feature graph depicted in Figure 3.2. In Figure 3.9, we show all transitive strong connections that can be found using our search algorithm. We depict the complete result, containing all transitive connections, in Figure 3.3.

Algorithm 3 Search Algorithm for Transitive Closure of a Feature Graph.

procedure TRANSITIVECLOSURE($featureGraph$)
  $nodes_{visited} \leftarrow \emptyset$
  for all $node \in featureGraph.nodes$ **do**
    $nodes_{visitedCopy} \leftarrow nodes_{visited}$
    SEARCHSTRONG($node$, $node$, $nodes_{visitedCopy}$)
    $nodes_{visited} \leftarrow nodes_{visited} \cup \{node\}$
  end for
  $nodes_{visited} \leftarrow \emptyset$
  for all $node \in featureGraph.nodes$ **do**
    $nodes_{visitedCopy} \leftarrow nodes_{visited}$
    SEARCHWEAK($node$, $nodes_{visitedCopy}$)
    $nodes_{visited} \leftarrow nodes_{visited} \cup \{node\}$
  end for
end procedure
procedure SEARCHSTRONG($node_{start}$, $node_{current}$, $nodes_{visited}$)
  $nodes_{visited} \leftarrow nodes_{visited} \cup \{node_{current}\}$
  $nodes_{adjacent} \leftarrow node_{current}.strongNeighbors \setminus nodes_{visited}$
  for all $node_{neighbor} \in nodes_{adjacent}$ **do**
    ADDSTRONGCONNECTION($node_{start}$, $node_{neighbor}$)
    SEARCHSTRONG($node_{start}$, $node_{neighbor}$, $nodes_{visited}$)
  end for
end procedure
procedure SEARCHWEAK($node_{start}$, $node_{current}$, $nodes_{visited}$)
  $nodes_{visited} \leftarrow nodes_{visited} \cup \{node_{current}\}$
  $nodes_{adjacent} \leftarrow node_{start}.allNeighbors \setminus nodes_{visited}$
  for all $node_{neighbor} \in nodes_{adjacent}$ **do**
    ADDWEAKCONNECTION($node_{start}$, $node_{neighbor}$)
    SEARCHWEAK($node_{start}$, $node_{neighbor}$, $nodes_{visited}$)
  end for
end procedure

Since transitive closure adds connections to the feature graph, it becomes more dense. Due to this circumstance, we store a feature graph, restructured with transitive closure, as an adjacency matrix. Although the usage of an adjacency matrix leads to quadratic space consumption with respect to the number of variant features in the given feature model, contrary to an adjacency list, a matrix has a constant size regarding the number of connections within the feature graph.

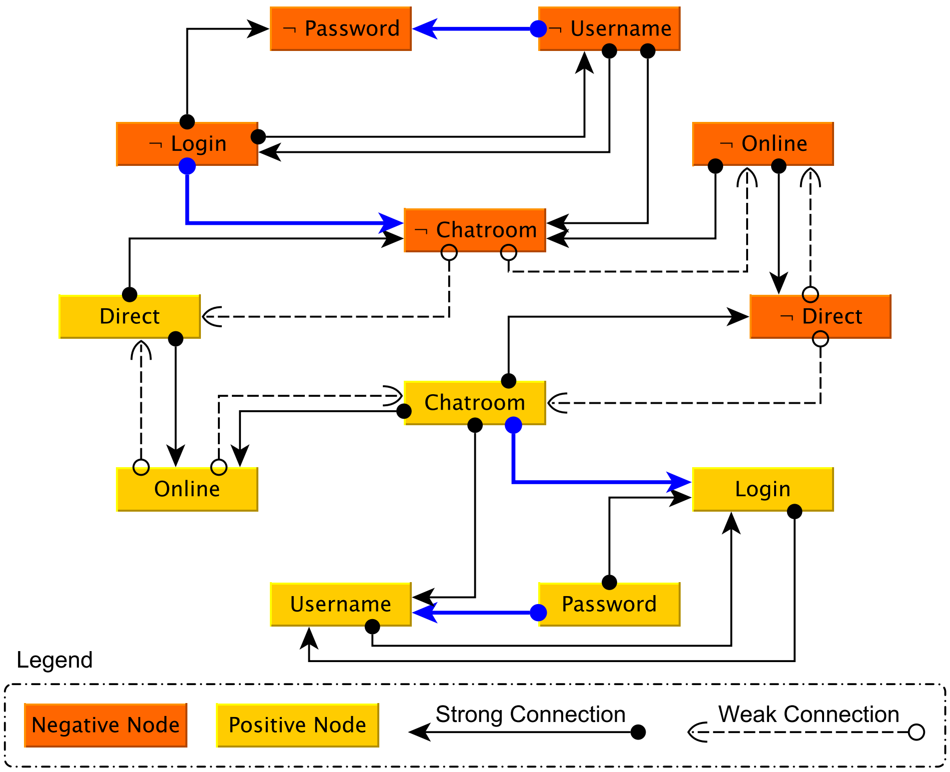

Figure 3.9: Feature graph for the Chat feature model (cf. Figure 3.1) during feature-graph restructuring (using transitive closure).

Transitive Reduction

With transitive reduction, we try to make the graph as minimal as possible by removing transitive connections within the graph. When we consider the complete feature graph of our Chat feature model (cf. Figure 3.2), we can see that it is already free of transitive connections. Due to the small amount of cross-tree constraints, no redundant connections were added during the feature-graph construction.

The strategy of transitive reduction can help to reduce the graph's space consumption and may lead to performance improvements during the configuration phase. A sparse graph can be saved efficiently, in terms of space consumption, by using an adjacency list. Regarding computation time during the configuration phase, there exists both, advantages and disadvantages. On the one hand, the selection algorithm needs to traverse through the whole feature graph (i.e., perform a full DFS) in order to find all potential selection states it needs to consider. This may lead to performance loss compared to the simplified selection algorithm, which we described above. On the other hand, it is possible to enhance the selection algorithm to exclude certain paths and, subsequently, reduce the amount of complex propagation tests. By this, we are able to compensate a weakness of the alternative strategy, transitive closure.

During the application of transitive closure, weak paths always lead to weak transitive connections, which can cause unnecessary computations in certain cases. We demonstrate this situation with the help of the transitive reduced feature graph (see Figure 3.2) and the transitive closed feature graph (see Figure 3.3) of our Chat feature model. Assume, we select feature `Online` and afterwards deselect feature `Direct`. By the application of transitive closure, there exists a weak connections from $\neg Direct$ to $Chatroom$, $Username$, and $Login$. Therefore, the selection algorithm would apply the complex propagation test to determine the selection states of the features `Chatroom`, `Username`, and `Login`. However, from the selection state of `Chatroom`, we can directly infer the selection states of `Username` and `Login`, due to the strong connection between these three features. A selection algorithm that is able to consider this circumstance and traverses carefully through the feature graph, could exclude unnecessary complex propagation tests and, thus, would have a faster performance. Of course, the success rate of this method highly depends on the traversing order. However, since we are using transitive closure, we do not further investigate feasible traversing orders for the selection algorithm.

Strategy Comparison

Both strategies have their individual advantages, however, in our actual implementation we use transitive closure. The main advantage of transitive closure is that the simplified selection algorithm does not need to consider a specific traversing order. Thus, we are able to use any arbitrary traversing order, which is the demand of our secondary conditions that we specified in Chapter 1 (cf. condition 2). Another reason, why we choose this strategy over transitive reduction, is the lower implementation effort. Both the adjacency matrix as well as the automated decision propagation algorithm can be implemented more easily, which on the one hand saves time and on the other hand reduces the number of potential bugs in the implementation. Therefore, in the remainder of our thesis, we focus on the strategy of transitive closure and the simplified selection algorithm. Nevertheless, the strategy of transitive reduction might be worth considering in the future to further improve our approach.

3.3.3 Feature-Graph Storage

In sum, the initialization phase of our approach consists of two time consuming tasks, the determination of variant features and the restructuring of the feature graph. For each new interactive configuration process, we have to re-execute the initialization phase. However, all information from the initialization phase are available in the feature graph. Hence, it is wise to save the computed graph after the first initialization phase to the hard drive and load it again, when needed. As long as the used feature model is not modified, we can load the already computed feature graph into the main memory and, thus, are able to skip the initialization phase of our approach.

Above, we already discussed the space consumption for the two different restructuring techniques. A possible way to further minimize the required memory space is the usage of certain compression techniques. However, the additional investigation of a suitable feature-graph compression technique is beyond the scope of our thesis.

4. Implementation

In this chapter, we explain the implementation details of our approach, the configuration assistant. In particular, we describe the internal structure of the feature graph and the propagation algorithm that is used for the interactive configuration process. Furthermore, we explain the implementation of the complex propagation test and propose two modifications that improve its performance.

We prototypically implement our configuration assistant in Java 1.7 and embed it into FeatureIDE to use the already existing tool support such as loading and analyzing feature models. For instance, we use the analysis for variant features and the dependency analysis implemented in FeatureIDE (cf. Section 2.4).

4.1 Feature-Graph Structure

In Chapter 3, we introduced two alternative concepts for restructuring the feature graph, transitive closure and transitive reduction, which both have their individual advantages. Our implementation, presented in this chapter and used for our evaluation, is based on transitive closure. Due to the inclusion of all transitive connections, the feature graph can become relatively dense, in theory. Since an adjacency list produces too much spatial overhead for dense graphs, compared to an adjacency matrix, we use a matrix to store the feature graph data structure.

4.1.1 Underlying Data Structure

The adjacency matrix is a 2D array that contains all connections of the graph. Since we use a directed graph, the matrix is not symmetrical. Thus, when we access a single value, the order of the specified indices matters. For instance, if we want to check whether there is a connection from node A with the index 1 to Node B with the index 2, we read the matrix cell at the position $(1, 2)$. To check the other direction, we have to invert both indices (i.e., $(2, 1)$).

Our concept uses two different connection types, strong and weak connections (cf. Chapter 3). In addition, there can also be no connection between two features. Hence, we need at least two bits to indicate the existence of a connection. Therefore, we decided to use one byte for each cell of the adjacency matrix and store it as one linear byte array. A linear array infers that we map each index tuple (i, j) to just one value k with the function $k = (i * n) + j$ where n is the number of nodes in the feature graph.

4.1.2 Connection Encoding

To further utilize the storage capacity of a one-byte cell, we combine the positive and negative nodes of each feature. If one feature is not connected to another one, we can express this with an empty cell (i.e., it contains the value 0). Naturally, the main diagonal of the matrix only contains empty cells, since no feature is connected to itself. Otherwise, if a feature has at least one connection to another feature, the corresponding cell has to specify three distinct properties, which we state in Table 4.1.

Property	**Possible values**	**Meaning**
From	positive, negative	whether from-node is negative or positive
To	positive, negative	whether to-node is negative or positive
Connection	weak, strong	whether the connection is weak or strong

Table 4.1: Information in a matrix cell for one connection.

Using these three independent properties, there are 8 possible combinations. Thus, we use the byte of each matrix cell as a bit field, where each single bit represent a certain connection. We list the encodings of all 8 bits in Table 4.2. The advantage of using single bits is that they can be handle by using bitwise operations, such as shifting and logical operations, which has a positive effect on the runtime performance.

Bit	**From**	**To**	**Connection**
`00000001` (`0x01`)	negative	negative	weak
`00000010` (`0x02`)	negative	negative	strong
`00000100` (`0x04`)	negative	positive	weak
`00001000` (`0x08`)	negative	positive	strong
`00010000` (`0x10`)	positive	negative	weak
`00100000` (`0x20`)	positive	negative	strong
`01000000` (`0x40`)	positive	positive	weak
`10000000` (`0x80`)	positive	positive	strong
`00000000` (`0x00`)	-	-	none

Table 4.2: Meaning of each bit in a single cell of the adjacency matrix.

Since one cell refers to more than one node in the feature graph, the single bits can be combined with each other to indicate multiple connections. For example, consider our

Chat feature model from Chapter 3 (see Figure 3.1). The feature model has six variant features. Hence, the resulting byte array for the adjacency matrix has 36 entries. Based on preorder indexing of the features, the feature `Online` has the index 0 and `Chatroom` has the index 2. Thus, the 12th cell (i.e., $(2 \cdot 6) + 0 = 12$) in the byte array represents all connections in the feature graph from `Chatroom` to `Online`. In our example, the cell has the value `10000101`. With respect to our encoding given in Table 4.2, we see that there are three connections, a strong connection from the node *Chatroom* to the node *Online*, a weak connection from $\neg$*Chatroom* to *Online*, and a weak connection from $\neg$*Chatroom* to $\neg$*Online*.

Note that the bits in Table 4.2 are ordered in a certain way. The four upper bits represent connections from positive nodes, whereas the four lower bits represent connections from negative nodes. Thus, both bit-groups are independent of each other and can be combined in any way. By contrast, there are invalid bit combinations within a single bit-group. For example, the byte `00001010` is invalid, because the four lower bits represent contradictory strong connections (i.e., $\neg A \rightarrow B$ and $\neg A \rightarrow \neg B$). In total, there are six valid and ten invalid combination for each bit-group. We listed all possible combinations in Table 4.3.

Combination	**Valid**	**Connection**	**To**
`0000` (`0x00`)	yes	none	-
`0001` (`0x01`)	yes	weak	negative node
`0010` (`0x02`)	yes	strong	negative node
`0011` (`0x03`)	no	-	-
`0100` (`0x04`)	yes	weak	positive node
`0101` (`0x05`)	yes	weak	positive *and* negative node
`0110` (`0x06`)	no	-	-
`0111` (`0x07`)	no	-	-
`1000` (`0x08`)	yes	strong	positive node
`1001` (`0x09`)	no	-	-
`1010` (`0x0A`)	no	-	-
`1011` (`0x0B`)	no	-	-
`1100` (`0x0C`)	no	-	-
`1101` (`0x0D`)	no	-	-
`1110` (`0x0E`)	no	-	-
`1111` (`0x0F`)	no	-	-

Table 4.3: Validity of all possible bit combinations for one bit-group (i.e, the four upper or lower bits).

4.1.3 Feature-Graph Storage

The usage of an adjacency matrix means that the byte array grows quadratically in size with an increasing number of features. Considering only the byte array, the feature graph uses a memory space of $n^2 + 12$ byte where n is the number of variant features and 12 the overhead for a primitive array in Java. However, the byte array is only one part of a single Java class that represents the entire feature graph. Besides the byte array, the class also contains three string arrays to store the core, dead, and variant features separately. Although, we do not include core and dead features in the feature graph, these features must be present for the configuration process to ensure a consistent feature model. The array containing all variant features is used to define a unique index for each feature in the feature graph. Generally, the exact size of these arrays cannot be specified in advance, since it is dependent on the length of the single feature names. Anyway, the total size of all three arrays only grows linearly with the number of features. Thus, their impact on the overall space consumption of the feature graph class is negligible for large feature models.

Since the initial computation of the feature graph takes up some time, we implemented a store and load mechanism to save the feature graph to the hard drive. For this, we use the native serialization stream of Java. Hence, we are not using any compression techniques to shrink the feature graph's size. However, it is most likely that even standard compression techniques can reduce the size of the saved array drastically, which has mainly two reasons. First, more than half of the possible bit combinations are invalid and, second, it is unlikely that the valid bit combinations are evenly distributed.

4.2 Selection Algorithm

The connections in our feature graph determine whether we have to use the complex propagation test or are able to directly deduce the implied selection state. In this section, we describe the implementation of the traversal through a feature graph during one configuration step. Furthermore, we describe the implementation of the complex propagation test that we use for our evaluation. In addition, we propose two modifications that we use to improve the performance of the complex propagation test.

4.2.1 Feature-Graph Traversal

Each configuration step consists of one configured feature and the subsequent decision propagation (see Section 2.3). For the propagation, we have to traverse through the feature graph to find all possibly affected features. In our implementation, we use transitive closure to compute all transitive connections before the actual configuration process. Hence, the traversal of the feature graph during one configuration step can be reduced to an iteration of all direct neighbors of the configured feature. In detail, we iterate through a complete row of the adjacency matrix. That means, if the configured feature has the index i, we check all matrix cells from $(i, 0)$ to $(i, n-1)$, where n is the number of features in the feature graph.

Depending on the defined selection state of the configured feature, we either consider the four upper (i.e., positive) or the four lower bits (i.e., negative). For every other feature of the feature graph, we find one of the six valid bit combination as shown in

Table 4.3. Each bit combination infers an appropriate action, which we list in Table 4.4. The combination `0000` indicates that there is no connection from the configured feature to the other one. Thus, in this case, the algorithm has to do nothing and proceeds. If we find a strong connection (i.e., for the combinations `0010` or `1000`), we accordingly change the selection state of the other feature, to positive or negative. Otherwise, if we find a weak connection (i.e., `0001`, `0100`, or `0101`), we add the other feature to a list of features that we have to test with the complex propagation test. Since a weak connection can connect to a positive and a negative node, we manage two separate lists for potential conditionally core and conditionally dead features.

Valid Combination	**Action**
`0000` (`0x00`)	do nothing
`0010` (`0x02`)	deselect the current feature
`1000` (`0x08`)	select the current feature
`0001` (`0x01`)	add current feature to dead list
`0100` (`0x04`)	add current feature to core list
`0101` (`0x05`)	add current feature to core list and dead list

Table 4.4: Performed action for each valid bit combination for one bit-group.

After we finished the traversal through the feature graph, we changed the selection states of all features that could be reached via a strong connection. In addition, we collected all features that are weakly connected to the configured feature. Afterwards, we perform the complex propagation test with each feature in the core and dead list independently. Thus, we present our implementation of the complex propagation test in the following section.

4.2.2 Complex Propagation Test

All weakly connected features that were collected by the selection algorithm during the feature-graph traversal have to be tested with the complex propagation test. As we stated in Chapter 3, the complex propagation test consists of an arbitrary implementation of the dependencies analysis (cf. Section 2.4.3). Generally, our approach can be used with every dependencies-analysis implementation, as long as it is conform to our secondary conditions specified in Chapter 1. For our implemented prototype, we use a slightly adapted dependencies-analysis implementation of FeatureIDE, which is based on satisfiability solvers. In turn, FeatureIDE relies on the Sat4j library, which is a popular Java library that provides multiple satisfiability-solver implementations [LBP10].

FeatureIDE uses the dependency-analysis implementation concept that we presented in Chapter 2 (cf. Section 2.4.3). At first, FeatureIDE's algorithm assigns the truth values to all variables in the propositional formula according to the selection states in the current partial configuration. The truth value of the variable for each selected feature is set to $true$ and for each deselected feature to $false$. Then, the algorithm iterates over all undefined features and performs a satisfiability test for each feature as follows. The truth value of the variable for the undefined feature is set to $false$ and subsequently a satisfiability solver determines the satisfiability of the formula regarding the current variable assignment. If the formula is not satisfiable, then the current feature is conditionally core and, thus, is selected in the partial configuration. Otherwise, if the formula is satisfiable, the algorithm tests whether the undefined feature is conditionally dead by applying the same test with the initial truth value *true* and, if necessary, deselects the feature in the partial configuration. Thus, for each undefined feature that is checked, the algorithm has to query the satisfiability solver. In the worst-case, this results in $2 \cdot n$ satisfiability solver calls, where n is the number of undefined features in the given partial configuration.

Since we are using FeatureIDE's dependency-analysis implementation in our configuration assistant, we made two modifications that significantly speed up the process. We are using multi-threading for parallel computation and exploit a property of satisfiability solvers to reduce the number of checks it has to execute. It is possible to combine both modifications and, thus, we implemented both and use them in our evaluation. In the following, we present both modifications in more detail.

Satisfiability Model

Since our complex-propagation-test implementation uses satisfiability solvers, we can exploit a certain property of these solvers to decrease the total number of complex propagation tests during one configuration step. Each time a satisfiability solver positively tests a propositional formula for satisfaction, it has identified a satisfying variable assignment, also known as *model*. This model can be used to exclude some possible selection states in advance, without testing them explicitly. For instance, if a model defines a variable as *true*, we know that there exists at least one valid configuration that includes the corresponding feature. Thus, it is not possible that this feature is (conditionally) dead. Hence, we do not need to execute the according complex propagation test. Analogous, a variable cannot represent a core feature, if a model defines the variable as $false$.

As example for this modification, we use our Chat feature model from Chapter 3 (see Figure 3.1) for an interactive configuration. We perform a first configuration step by selecting the feature `Login`. Next, we use a satisfiability solver for the decision propagation. At first, we assign the truth value for $Login = true$ and perform a satisfiability check. Since the formula is still satisfiable, the solver finds a suitable model. Here we assume that the solver computes the model ($Chat = true$, $Online = false$, $Direct = false$, $Chatroom = false$, $Login = true$, $Username = true$, $Password = false$). Thereby, we now know that the variables $Online$, $Direct$, $Chatroom$, and $Password$ can be $false$ in a satisfying variable assignment. Therefore, the corresponding features cannot be conditionally core. Similarly, it is possible that the variables $Chat$, $Login$, and $Username$ are $true$ and, thus, it is impossible that their corresponding features are conditionally dead.

By using the proposed modification, at least half of all complex propagation tests become unnecessary. That means that this modification approximately improves the overall runtime of the decision propagation by factor 2. Moreover, we also update the current model after each complex propagation test, which should result in additional performance improvements.

Multi-Threading

Due to our secondary conditions from Chapter 1, our approach is able to determine the selection states of the features independently of each other. This means, we are able to compute multiple complex propagation tests in parallel.

Our prototype uses the Sat4j library, which, unfortunately, does not support concurrent access to a satisfiability solver. Therefore, we have to use an extra satisfiability-solver instantiation for each thread, which results in some minor disadvantages. An extra instance for each thread produces more overhead for the initialization phase and requires a higher amount of memory space. Since the single instances are not intended for parallel work, they cannot share their internal states, which might lead to some duplicate computations. However, when we combine both modifications, we are able to mitigate this disadvantage by sharing the computed model and all excluded truth values. Still, we must be aware of concurrent write access to the shared model and, thus, we have to synchronize its update method.

4.2.3 Graphical Interaction

Since, FeatureIDE provides an interactive graphical user interface (GUI), we have to make visible updates for the developer. In FeatureIDE, the developer can edit a configuration via a configuration editor, which list all features in form of a tree-structured list. Every feature has an advanced check box that indicates whether the feature is selected, deselected, or still undefined. Via clicking this check box, the developer can change the selection state of the corresponding feature (i.e., perform a configuration step). Each change then triggers the decision propagation for the altered partial configuration.

Normally, the GUI waits for the decision propagation to finish, before updating the check boxes of all features. However, as our secondary conditions from Chapter 1 demand, our configuration assistant computes the selection states of each feature individually. Thus, we are enabled to update the check boxes for each feature on its own. In addition, we start the decision propagation with the set of features that are currently visible to the developer. In most cases, this is a very small percentage of the total number of features. This approach empowers the developer to change the selection state of another feature before the current decision propagation has finished. When our selection algorithm executes the complex propagation tests, it checks after each test if the current partial configuration was altered manually by the developer. If so, the currently running selection algorithm interrupts itself and afterwards restarts with the new partial configuration as input. When the selection algorithm is interrupted, it saves the lists containing the not yet computed selection states from the current decision propagation and considers these lists in the restarted process. Thus, the final result of the new decision propagation is still correct, such as if both decision propagation processes were executed consecutively.

5. Evaluation

In this chapter, we evaluate our approach, the configuration assistant, to find answers to our research questions from Chapter 1. At first, we describe our evaluation concept, which properties we want to evaluate, the concrete evaluation set up, and the used feature models. Then, we present and analyze our evaluation results and discuss possible threads to validity. We compare our evaluation results with other state-of-the-art configuration tools, such as S.P.L.O.T. (Software Product Line Online Tools) and FeatureIDE. In addition, we collect and examine various statistical information of feature graphs from multiple feature models, during the evaluation process.

5.1 Evaluation Concept

As reminder, we, once more, list all of our three research questions below.

RQ1: *Does the usage of a feature graph significantly reduce the required computational effort for decision propagation?*

RQ2: *Does the performance improvement dependent on the used feature model and if so, which kinds of feature models are most suited for our approach?*

RQ3: *How is the overall performance of the feature graph, regarding construction time and memory consumption?*

In order to answer our research questions properly, we firstly present an evaluation concept that enables us to measure all necessary values. Initially, we define our evaluation objectives (i.e., which values we want to measure). Then, we describe our evaluation set up and what tools and hardware we use for the evaluation process. Finally, we present the feature model collection that we use as input for the evaluated configuration tools. We use a variety of feature models, which originated from different feature model repositories, our industrial partners, and the S.P.L.O.T. feature-model generator.

5.1.1 Evaluation Objectives

During our evaluation, we perform multiple measurements. In particular, we want to evaluate the following four properties for every feature model.

1. The *time* required for the *initialization phase* of each configuration tool.
2. The *time* required for the *decision propagation* by each configuration tool.
3. The *memory-space consumption* of the feature graph.
4. The *amount* of the different *connection types* within the feature graph.

Initialization Time

In Chapter 3, we mentioned that our approach requires an initialization phase to build the feature graph of a feature model. Anyway, all other configuration tools require certain initial computations as well. Therefore, we measure the computation time of the initialization phase for all used configuration tools on each feature model. In particular, all SAT-based configuration tools, including FeatureIDE, S.P.L.O.T., and our configuration assistant, have to determine the set of variant features for the given feature model. As we decided to use transitive closure in our implementation, the initialization phase of the configuration assistant is extended by the determination of all transitive connections for the feature graph. Beside S.P.L.O.T.'s SAT-based approach for the interactive configuration process, it offers another method, which, in its initialization phase, has to construct a suitable binary decision diagram (BDD). By comparing the initial computation times of all configuration tools, we are able to partly answer our research question RQ3.

Decision-Propagation Time

The next measured value, the required time for decision propagation, is the basis for answering our research question RQ1. Again, we measure the times for all used configuration tools and afterwards compare the results. Due to the exponential number of different valid configurations, it is practically impossible to compare all configurations of a large feature model. Hence, we thought of three configuration plans, `False`, `True`, and `Random`, to efficiently compare the performance of different configuration tools. To simulate an interactive configuration process, we iterate over all features of a feature model in a certain order and if a feature's selection state is undefined, we set it, according to the used configuration plan, to either selected or deselected. Our first configuration plan, `False`, tries to deselect as much features as possible by deselecting each undefined feature. By contrast, our second configuration plan, `True`, selects each undefined feature and, thus, tries to select as much features as possible. Lastly, our third configuration plan, `Random`, decides randomly whether to select or deselect an undefined feature. The used feature order is equal for each configuration plan. In detail, we use the order given by a preorder traversal of the corresponding feature diagram.

Our first and second configuration plan are straight-forward approaches for selecting and deselecting as much features as possible. Together, they represent an appropriate indicator for the average computation time. Our third configuration plan is designed as an approximation of how a developer would configure a product line. Normally, a person traverses through a tree in preorder (i.e., manually performing a depth-first search) and decides for each feature whether they want to in- or exclude it from the current product (if the feature is still undefined). Of course, we use the same random selection of features for each configuration tool. This is realized by using the same seed for the Java pseudo-random generator for all configuration tools. As the performance of this configuration plan can vary depending on the randomly chosen selection states, we use more than one random sample to receive a more significant result. In our evaluation, we use 6 passes for each model and configuration tool and then compute the arithmetic mean to get an average result.

Feature-Graph Memory-Space Consumption

To answer the second part of our research question RQ3, we measure the memory-space consumption of all feature graphs. For this, we save the feature graphs to the hard drive by using Java's native serialization mechanism (cf. Chapter 4). Afterwards, we determine the size of the saved feature-graph files. In addition, we want to measure the potential of reducing a feature graph's memory-space consumption through compression. Thus, we compress the feature-graph files with a standard compression technique using the open-source tool 7zip[1] (Version 9.20). As compression technique, we use the well-known LZMA algorithm with 7zip's default settings.

Feature-Graph Connections

Our research question RQ2 addresses the suitability of different feature models for our approach. Thus, we are interested in structural information about the feature graphs for our used feature models. Since we assume that the distribution of a feature graph's connection types has the most influence on the performance during the configuration phase, we measure this value and relate it to the computational time for the decision propagation. For this, we count the connections within the graph by using a static and a dynamic approach. First, we ingestive the entire feature graph and count all existing connections within it (i.e., *static analysis*). Second, we count the actually visited connections during the decision propagation (i.e., *dynamic analysis*). Additionally, in the dynamic analysis, we measure the total number of executed complex propagation tests. From the static analysis, we can deduce information about the feature graph's structure. For instance, its denseness and the ratio between its strong and weak connections. The dynamic analysis gives us information about the traversal in the feature graph during decision propagation.

5.1.2 Evaluation Set Up

In the following, we describe our evaluation set up, which tools we used, and our hardware specifications. For the evaluation, we use the prototypical implementation

[1] http://www.7-zip.org

of our configuration assistant, which we described in Chapter 4. We evaluate this implementation against other approaches for decision propagation. Additionally, since our configuration assistant allows the usage of multi-threading (cf. Chapter 4), we use a varying number of threads for the evaluation of our approach. In total, we use the following six methods for the interactive configuration process:

- FeatureIDE (FIDE)
- S.P.L.O.T. using a satisfiability solver (SplotSAT)
- S.P.L.O.T. using BDDs (SplotBDD)
- Configuration Assistant using 1 thread (CA1)
- Configuration Assistant using 2 threads (CA2)
- Configuration Assistant using 4 threads (CA4)

FeatureIDE

We already introduced FeatureIDE in Chapter 2 as a framework for various SPLE tasks, including the interactive configuration process. For our evaluation, we use the Version 2.7.4, which was published in June 2015. In Chapter 4, we stated that our approach is based on FeatureIDE and uses its dependency-analysis implementation for the complex propagation test. Therefore, we expect our approach to be at least as fast as FeatureIDE for the decision propagation.

S.P.L.O.T.

S.P.L.O.T. is a framework for configuring and analyzing SPLs [MBC09]. We use its latest version, which was build in November 2010. For our evaluation, we locally execute S.P.L.O.T. on our machine, instead of using its official web interface[2]. Thus, we are able to properly compare the results with other configuration tools. S.P.L.O.T. has two "configuration engines", one using satisfiability solvers and the other one using BDDs. In our evaluation, we test both of them. However, BDDs are not suited for very large feature models and, thus, we could only apply the BDD configuration engine to feature models with less than 5,000 features.

Despite using the latest version of Sat4j (2.3.5) in the actual prototype of our approach, in our evaluation we use the Version 2.0.0, which is also used by S.P.L.O.T.. Since there are performance differences between both Sat4j versions, we decided to use the same version for all configuration tools to ensure an unbiased comparison. Because of the incompatibility of S.P.L.O.T. with the latest Sat4j version, we downgrade the Sat4j version of FeatureIDE and our prototype, which works without any difficulty.

[2] http://www.splot-research.org/

Evaluation Platform

We execute our evaluation on a single machine with the following specifications:

- Processor: Intel Core i5-4670 (4 Cores @ 3.40 GHz)
- Main-Memory Size: 16 GB
- Operating System: Windows 7 Professional (64 Bit)
- Java Version: 1.7.0_71 (64 Bit)

To simulate an interactive configuration process, we implemented an evaluation tool that uses our three configuration plans and the different configuration tools. Our evaluation tool is based on Java 1.7 and contains wrapper interfaces for each configuration tool in order to ensure an equal interaction with each of them. To take time measurements, our evaluation tool uses the native Java command `System.nanoTime()`. In order to avoid excessive garbage collection and memory swapping, we increased the maximum heap size of the executing Java Virtual Machine (JVM) to 10 GB with an initial size of 6 GB.

Since some configuration tools are not suitable for certain feature models (e.g., SplotBDD for feature models with 5,000 or more features) and take an immense amount of time for decision propagation, we implemented a timeout mechanism to avoid wasting evaluation time. For instance, if we want to completely configure our largest feature model (with over 17,000 features), using the configuration plan `True`, FeatureIDE would need at least one week to finish. The timeout applies for the accumulated time of all executed configuration steps in one single configuration process. Before executing the next configuration step, our evaluation tool checks whether it reached the specified timeout and if so, cancels the current configuration process. For our evaluation set up, we determined an appropriate timeout value of 7,200,000 milliseconds (i.e., 2 hours). An exception is the feature model Splot10001, for which we increased the timeout value to $18,000,000$ ms (i.e., 5 hours).

5.1.3 Evaluated Feature Models

In order to evaluate our approach, we need large-scale feature models with at least 50 features. The time required for the configuration process of smaller feature models (i.e., feature models with less than 50 features) is too small (e.g., less than 10 ms) for a reasonable comparison. However, large-scale feature models are rare among online feature-model repositories. From our industrial partners, we got two feature models with over 2,000 and 17,000 features. In addition, we searched for large feature models in the repositories of S.P.L.O.T. and FeatureIDE. We found feature models with around 100, up to 300 features. Finally, we used artificial feature models of different sizes, which were created with the S.P.L.O.T. feature-model generator. In the following, we describe all used feature models in more detail. Additionally, we discuss the handling of different feature-model-file formats of S.P.L.O.T. and FeatureIDE.

In Table 5.1, we list certain structural information for the evaluated feature models. In detail, the table includes the feature model name, the number of features and cross-tree

Model	#Features	#Groups		#Constraints	Constraint
		Alternative	OR		Coverage (%)
BerkeleyDB1	76	8	4	20	42.1
EShopFIDE	326	0	39	21	10.4
Automotive1	2,513	407	43	2,833	50.9
Automotive2	17,365	1,165	111	948	6.5
Splot1001	1,120	62	75	100	8.4
Splot1006	1,109	62	76	100	8.7
Splot2004	2,212	141	128	100	6.9
Splot2005	2,236	145	136	100	7.1
Splot5001	5,545	339	336	150	5.3
Splot5005	5,543	350	324	150	5.3
Splot10001	11,065	676	617	100	2.4

Table 5.1: Structural information about evaluated feature models.

constraints, and the number of alternative- and OR-groups in the feature diagram. In addition, we state the relative number of features that are contained in one or more cross-tree constraints (i.e., constraint coverage). The provided structural information can be used as an indicator for a feature model's complexity. Due to spatial limitations, in this chapter, we just provide a representative selection of all used feature models. A complete list of all feature models and their corresponding statistical values can be found in Table A.1.

Real-World Feature Models

Both feature models that we got from our industrial partners are from the automotive domain. However, they are obfuscated in a way that all feature names are replaced with unique identifiers. Hence, we call the feature models Automotive1 and Automotive2. Automotive1 has 2,513 and Automotive2 17,365 features. We list more details for both feature models in Table A.1.

Feature-Model Repositories

We selected several feature models from the S.P.L.O.T. online repository[3] and from the example feature models provided by FeatureIDE[4]. In detail, we selected the following six feature models:

- Dell (S.P.L.O.T.)
- EShopSplot (S.P.L.O.T.)
- BerkeleyDB1 (FeatureIDE)

[3] http://www.splot-research.org/
[4] https://github.com/tthuem/FeatureIDE/tree/master/plugins/de.ovgu.featureide.examples/

- BerkeleyDB2 (FeatureIDE)
- Violet (FeatureIDE)
- EShopFIDE (FeatureIDE)

In Table A.1 provide some more details about the feature models size and structure.

Feature-Model Generator

A part of S.P.L.O.T. is a feature-model generator with various parameters that can be used to create artificial feature models for evaluation purposes. S.P.L.O.T. already provides several generated feature models in its repository [MBC09]. Since these feature models can be easily accessed by others, we decided to use them for our evaluation instead of generating completely new ones. In sum, we selected 31 feature models with sizes from 1,000, up to 10,000 features.

- Splot1001 - Splot1010 ($\approx$ 1,000 features)
- Splot2001 - Splot2010 ($\approx$ 2,000 features)
- Splot5001 - Splot5010 ($\approx$ 5,000 features)
- Splot10010 ($\approx$ 10,000 features)

Again, we provide more details for each feature model in Table A.1.

Feature-Model-File Format

While S.P.L.O.T. stores feature models in the Simple XML Feature Model format (SXFM), FeatureIDE relies on its XML-based file structure. Thus, for each configuration tool, we have to convert the feature models in the corresponding format. FeatureIDE is capable of im- and exporting feature models from and to SXFM. However, due to its own feature-model format, when importing a feature model from SXFM, FeatureIDE needs to insert some connection features for alternative- and OR-groups, which slightly increases the total number of features. Therefore, after importing a feature model from SXFM, we exported it again to SXFM to ensure that every configuration tool works on the same model with the same number of features.

5.2 Evaluation Results

We now present the result of our measurements before and during the configuration process. At first, we present the results of the time measurement for the initialization phase of each configuration tool. Next, we show the time-measurement results for the actual interactive configuration process. Finally, we present the data that originated from the static and dynamic analysis of all feature graphs. As our measurements produced a high amount of values, we list most of our results in multiple tables in Chapter A. Nevertheless, to provide a proper overview of our results, we display a subset of all results based on the representative selection of feature models given in Table 5.1.

Model	Initialization Time (in ms)					
	CA1	CA2	CA4	FeatureIDE	SplotBDD	SplotSAT
BerkeleyDB1	10	9	9	19	24	21
EShopFIDE	22	17	16	42	111	20
Automotive1	1,430	1,143	1,056	1,396	218,663	1,410
Automotive2	98,039	81,966	69,784	53,003	-	598,512
Splot1001	316	262	245	312	176,912	135
Splot1006	314	261	241	300	627,256	141
Splot2004	1,267	1,036	957	1,023	597,840	726
Splot2005	1,326	1,086	1,006	1,124	330,369	693
Splot5001	4,490	3,593	3,401	4,769	-	3,782
Splot5005	6,934	5,790	5,285	5,929	-	7,508
Splot10001	43,958	39,267	34,781	26,291	-	50,659

Table 5.2: Time required by each configuration tool for its initalization phase (regarding the feature-model selection given in Table 5.1).

5.2.1 Initialization Time

In Table A.2 we compare the times that each configuration tool needed for their initialization phase, before starting the configuration process. For a more convenient comparison, we visualize the results for our representative feature-model selection (cf. Table 5.1) in Figure 5.1 and provide a shortened list of the results in Table 5.2. As it can be seen in the dataset, SplotBDD has very high values compared to other configuration tools. For all feature models with 5,000 or more features our evaluation tool was not able to build a BDD at all, due to the limited main memory capacities. Hence, we omitted the bar plot for SplotBDD for all feature models, except for BerkeleyDB1 and EShopFIDE.

In the dataset, we can see a wide range of measured values, reaching from 5 milliseconds (CA1) to over 2,000,000 milliseconds (SplotBDD). Remarkably, the initialization time for all feature models with less than 1,000 features is below 200 milliseconds for every configuration tool. Moreover, there is a clear correlation between the measured time and the feature model size, for each configuration tool, except SplotBDD. A higher number of features always leads to a higher computation time for the initialization phase.

In comparison, for most feature models, SplotSAT has the shortest initialization time, which is less than 1 second, even for feature models with 2,000 features. However, for the two largest feature models, Splot10001 and Automotive2, SplotSAT's initialization time is significantly higher than the times of FeatureIDE and our approach. When comparing the times of our approach and FeatureIDE, we discover that the results for CA4 and FeatureIDE are highly similar for most feature models. Furthermore, there is a visible correlation between the three variants of our approach CA1, CA2, and CA4. The measured times of CA4 are mostly between 20% and 30% smaller than those of CA1. Whereas the results of CA2 are somewhere between those of CA1 and CA4.

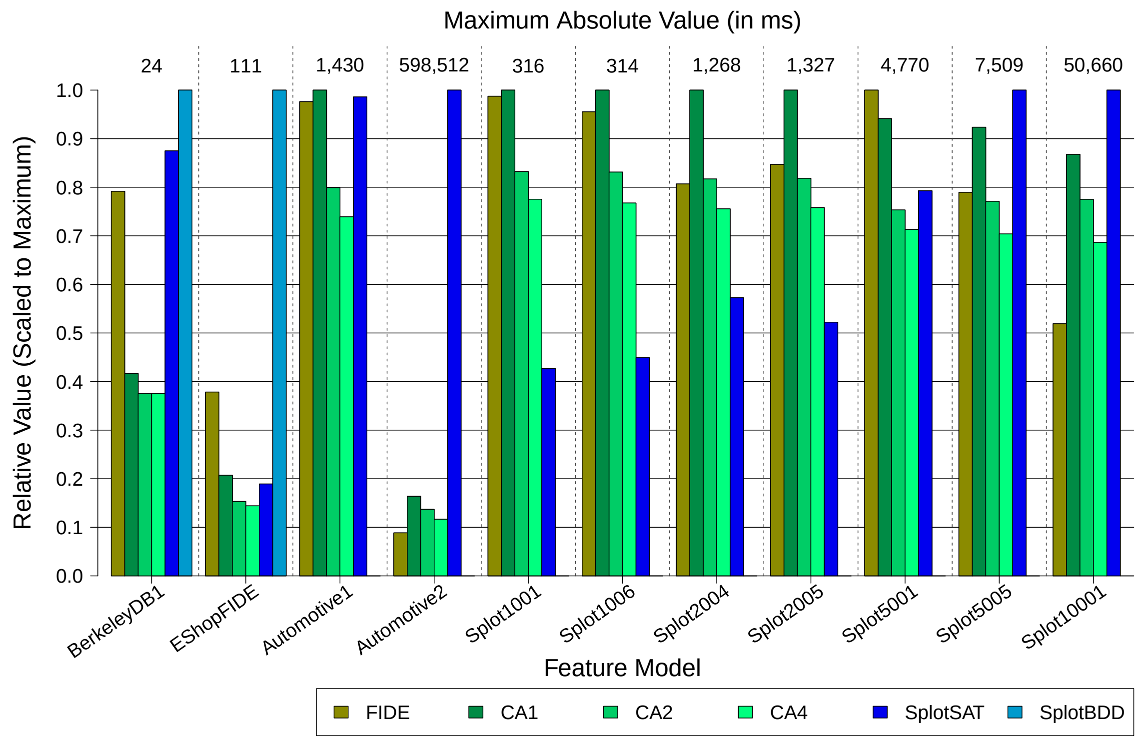

Figure 5.1: Comparison of initialization times for all configuration tools (regarding the feature-model selection given in Table 5.1).

5.2.2 Decision-Propagation Time

In the following, we show the decision propagation times for the different configuration tools. For each tool, we measured the maximum and average time needed for the decision propagation of one configuration step. We omitted the minimum time, as it was equal or close to 0 in almost all cases (except for the configuration processes with occurred timeouts). We also measured the accumulated computation time for each whole configuration process. We present the results of FeatureIDE, SplotSAT, SplotBDD, and CA1 in Table A.3, and for CA2 and CA4 in Table A.4. Additionally, we state the decision-propagation times for our feature-model selection (cf. Table 5.1) in Table 5.3 (for FeatureIDE and CA1) and in Table 5.4 (for SplotSAT and CA4). In all tables, each row contains the measured values for one feature model and a given configuration plan. For a proper overview, we group the results by the different feature models and, in addition, aggregate the results for all three configuration plans. Thus, the first row for each feature model contains the overall maximum time and the arithmetic mean of the average and the accumulated time (rounded down).

Model	FeatureIDE (in ms)			CA1 (in ms)		
	Max	∅	∑	Max	∅	∑
BerkeleyDB1	15	0	19	36	0	9
EShopFIDE	22	6	739	5	0	18
Automotive1	776	238	106,697	590	103	49,544
False	505	173	43,716	420	46	11,651
True	776	283	156,939	590	160	88,713
Random(∅)	687	257	119,437	426	102	48,268
Automotive2	* 66,762	62,239	7,248,604	2,762	105	921,950
Splot1001	175	49	15,142	152	41	13,325
Splot1006	182	59	16,562	144	43	12,554
Splot2004	692	245	135,552	510	153	88,433
Splot2005	931	218	118,029	599	209	119,133
Splot5001	2,644	1,073	985,222	1,952	493	463,505
Splot5005	3,818	1,343	1,709,376	4,061	1,102	1,475,103
Splot10001	* 13,992	6,152	9,429,971	* 17,880	6,194	8,985,031

Table 5.3: Decision-propagation times for FeatureIDE and CA1 (regarding the feature-model selection given in Table 5.1).

Note that there are missing values for SplotBDD, since it was not possible to construct a BDD for certain feature models. Moreover, timeouts occurred in our evaluation tool for the feature models Splot10001 and Automotive2 and the configuration tools FeatureIDE SplotSAT, CA1, and CA2. Therefore, all the measured values for those feature models and configuration tools are not accurate, but biased in certain ways. Naturally, the sum is capped to a value just over 7,200,000 milliseconds (18,000,000 for Splot10001), since this was the specified timeout value. By contrast, the average time is likely to be higher than for a complete configuration process, since later configuration steps are generally faster, due to less undefined selection states. Because of the same reasons, we can assume that the maximum value is close or equal to the real value. We annotated each data group (i.e., for one configuration tool) in a row that was affected by a timeout with an asterisk symbol (*).

We visualize the aggregated result, over all executed configuration plans for our feature-model selection (cf. Table 5.1) in the following three diagrams. In Figure 5.2 and Figure 5.3, we depict the *average* and the *maximum* computation time for one configuration step. In both diagrams, we omit the two smallest feature models, BerkeleyDB1 and EShopFIDE, as most values are close to 0 milliseconds. We show a comparison of the total computation times of all configuration processes in Figure 5.4. Since the measured values for SplotBDD are either missing or are disproportionately higher than the values of the other configuration tools, we only depict the values of SplotBDD for the first two feature models, BerkeleyDB1 and EShopFIDE.

Model	SplotSAT (in ms) Max	∅	∑	CA4 (in ms) Max	∅	∑
BerkeleyDB1	4	0	1	30	0	17
EShopFIDE	2	0	7	5	0	19
Automotive1	826	227	108,191	266	50	24,044
False	812	104	26,317	203	23	6,022
True	812	319	176,868	266	77	42,694
Random(∅)	826	259	121,390	217	49	23,416
Automotive2	* 508,729	499,464	7,491,963	1,266	71	634,596
Splot1001	73	12	4,232	93	19	6,295
Splot1006	91	17	5,052	91	18	5,395
Splot2004	541	133	73,735	252	63	36,697
Splot2005	548	97	56,101	305	88	50,053
Splot5001	2,293	457	436,579	1,103	262	246,664
Splot5005	6,647	1,232	1,595,739	2,230	578	774,688
Splot10001	* 48,342	14,374	10,071,199	9,546	2,711	5,555,055

Table 5.4: Decision-propagation times for SplotSAT and CA4 (regarding the feature-model selection given in Table 5.1).

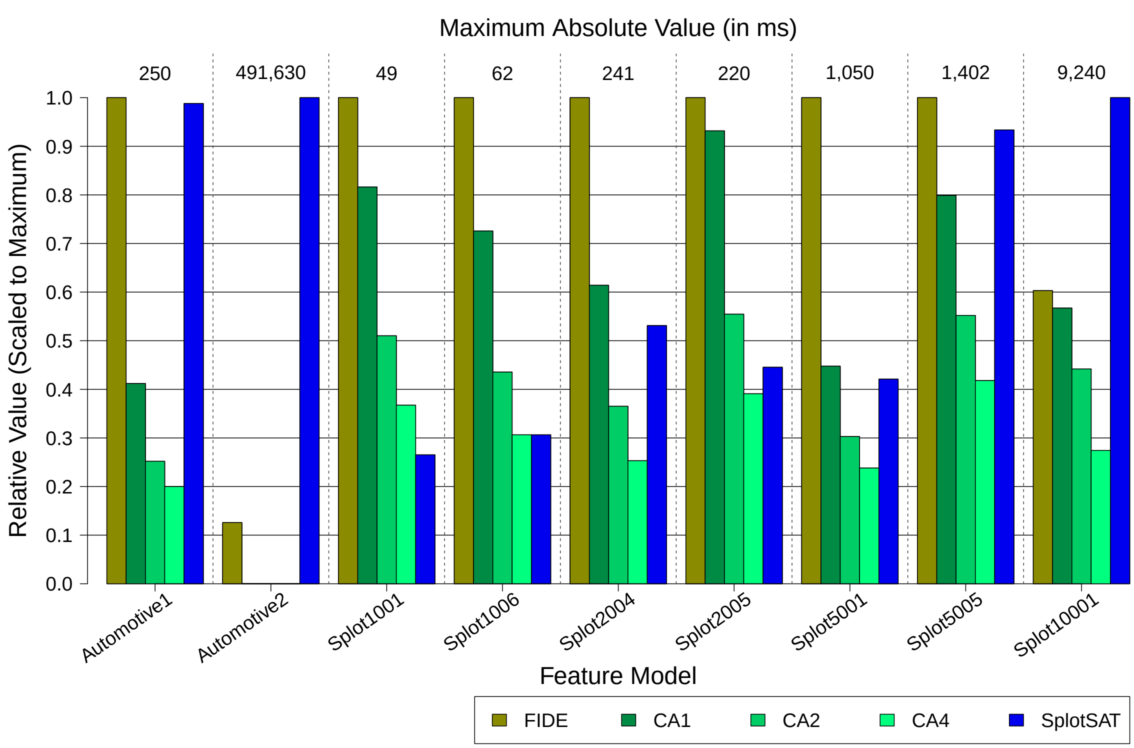

Figure 5.2: Comparison of the average decision-propagation times for each configuration tool (regarding the feature-model selection given in Table 5.1).

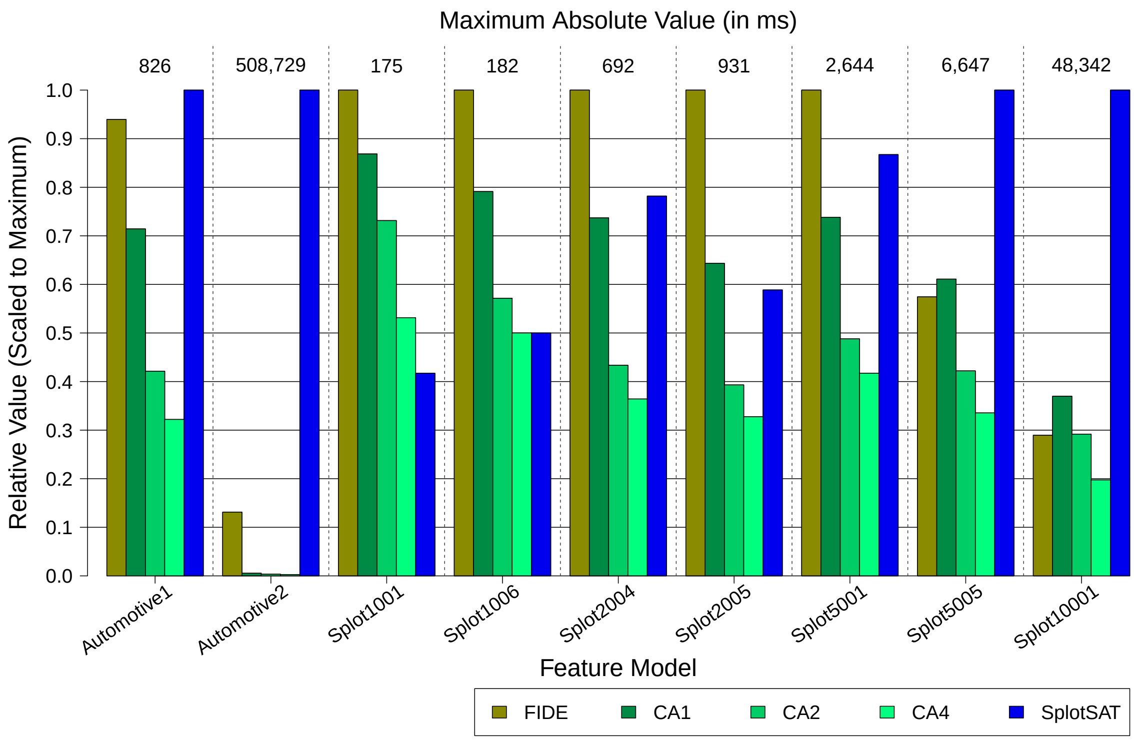

Figure 5.3: Comparison of the maximum decision-propagation times for each configuration tool and feature model (regarding the feature-model selection given in Table 5.1).

5.2.3 Feature-Graph Memory-Space Consumption

We now present the measured values for each feature graph's memory consumption in byte and its corresponding compression rate (i.e, $compression\ rate = \frac{size_{compressed}}{size_{uncompressed}}$), for a compression with LZMA (values rounded down). We list the values for all feature models in Table A.6 and for our feature-model selection (cf. Table 5.1) in Table 5.5.

The uncompressed feature-graph sizes are ranging from 5 kilobyte to 250 megabyte with compression rates from 33.6% to 0.1%. As we expected, we see a quadratic growth in size with an increasing number of features. Furthermore, with a higher number of features the compression rate decreases noticeably, which means that the compression is more effective for larger feature models. For instance, we can save 99.9% of the memory space for Automotive2.

5.2.4 Feature-Graph Connections

We state the results of our static analysis on each feature graph in Table A.6. The table contains the number of nodes and all weak and strong connections in the feature graph for each feature model. In addition, we calculate the number of non-existent potential connections between the nodes (i.e., $connections_{none} = nodes^2 - (connections_{weak} + connections_{strong})$). We list the result subset for our feature-model selection (cf. Table 5.1) in Table 5.5. In Figure 5.5, we visualize the number of connections in the

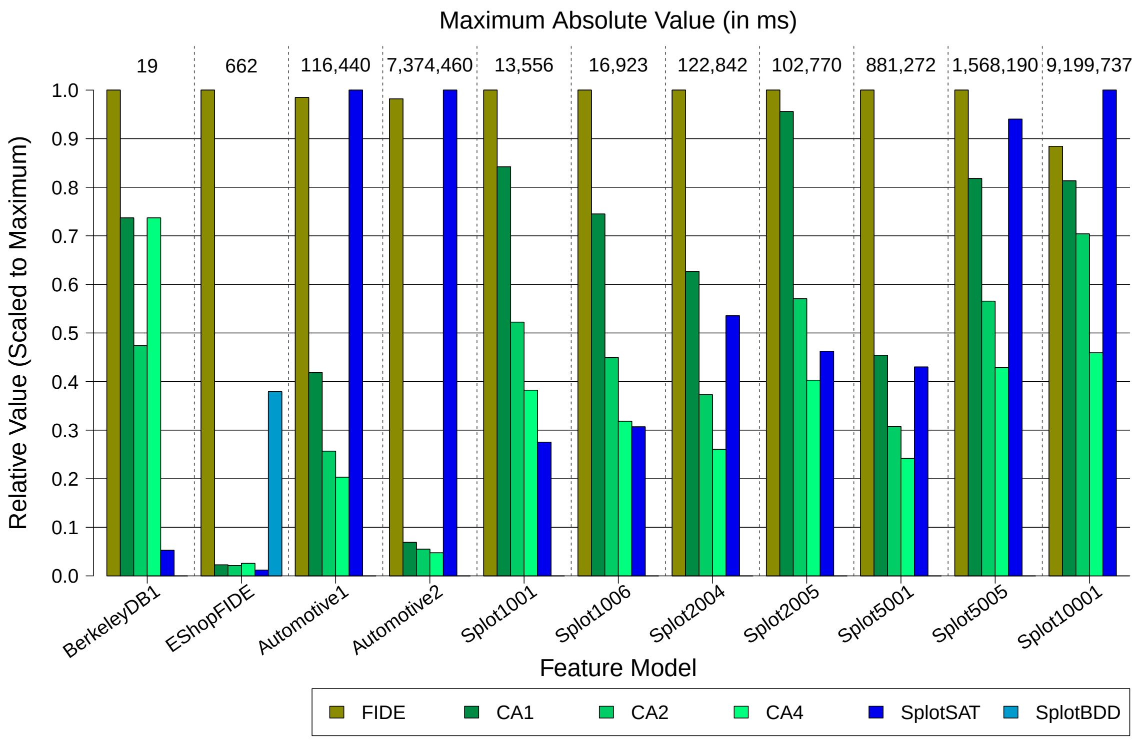

Figure 5.4: Comparison of the total computation times of the configuration process for each configuration tool (regarding the feature-model selection given in Table 5.1).

feature graph and relate them to the performance of CA4 compared to FeatureIDE and SplotSAT (i.e., the average accumulated decision-propagation times for each feature model).

The number of nodes in the feature graphs reaches from 76 to 31,614. Concerning the feature-graph connections, we can see that despite using transitive closure, all feature graphs are relatively sparse with a graph density below 50%. It is also visible that weak connections by far outnumber strong connections in every feature graph. While the number of strong connections range between 374 and 1,742,324, the number of weak connections reaches from 2,812 to 94,441,654.

Finally, we present the results of our dynamic analysis in Table A.5. For each configuration plan, we list the number of connections that the selection algorithm has visited in the feature graph. Additionally, we show the number of complex propagation tests (i.e., calls to the satisfiability solver) for CA1, CA2, and CA4. Again, we group the results by feature models and aggregate the values by calculating the arithmetic mean. Similar to the static analysis, we calculate the not-visited potential number of connections by using the following method. In a worst-case scenario the selection algorithm has to visit both nodes of each feature that is still undefined in the current partial configuration. Thus, the maximum number of connections that the selection algorithm is able to visit in one configuration step is two times the number of the currently undefined

Model	#Connections none	strong	weak	#Nodes	Size (in byte) (Compressed %)
BerkeleyDB1	13,325	1,500	3,671	136	9,409 (24.5)
EShopFIDE	254,162	1,958	12,204	518	105,324 (8.7)
Automotive1	14,375,894	195,698	5,106,504	4,436	5,086,553 (0.7)
Automotive2	996,174,355	695,346	2,575,295	31,614	250,875,368 (0.1)
Splot1001	2,687,005	55,596	1,983,675	2,174	1,248,816 (1.6)
Splot1006	2,757,169	66,760	1,755,671	2,140	1,210,071 (1.8)
Splot2004	12,286,639	73,296	6,216,165	4,310	4,771,038 (0.8)
Splot2005	10,888,199	116,370	8,904,875	4,462	5,111,251 (0.8)
Splot5001	39,969,030	539,128	20,020,242	7,780	15,396,671 (0.5)
Splot5005	74,445,701	348,354	44,758,301	10,934	30,206,394 (0.3)
Splot10001	389,559,638	967,192	94,441,654	22,022	121,877,300 (0.2)

Table 5.5: Results of the static analysis on certain feature graphs (regarding the feature-model selection given in Table 5.1).

features. Thereby, we can calculate the not-visited potential number of connections by subtracting the actual visited connections from the maximum value. We visualize the aggregated number of visited connections during the decision propagation in Figure 5.6 and again relate them to the performance of CA4 compared with FeatureIDE and Splot-SAT. In addition, we depict the number of complex propagation tests compared to the visited weak connections in Figure 5.7.

During the configuration phase, the ratio between weak and strong connections is even higher than for our static analysis, as the selection algorithm visits far more weak connections. The number of weak connections ranges from 50 to 20,115,423, whereas the number of strong connections just reaches from 1 to 12,248. However, the number of feature-graph nodes that the selection algorithm does not need to consider, due to absent connections ranges between 9 and 195,930,049. These numbers are comparatively high and indicate the high amount of avoided complex propagation tests during decision propagation. Moreover, the total number of executed complex propagation tests varies from 23 to 10,303,427 and is always at least 50% lower than the number of weak connections. When comparing CA1, CA2, and CA4, we notice that the number of complex propagation tests increases for a higher number of threads.

5.2.5 Result Discussion

In the following, we further assess our measured values and attempt to answer our research questions. We start by considering each of our three research questions individually, with regard to the evaluation results. Afterwards, we point out certain minor remarks and general conclusions that we can infer from our evaluation results.

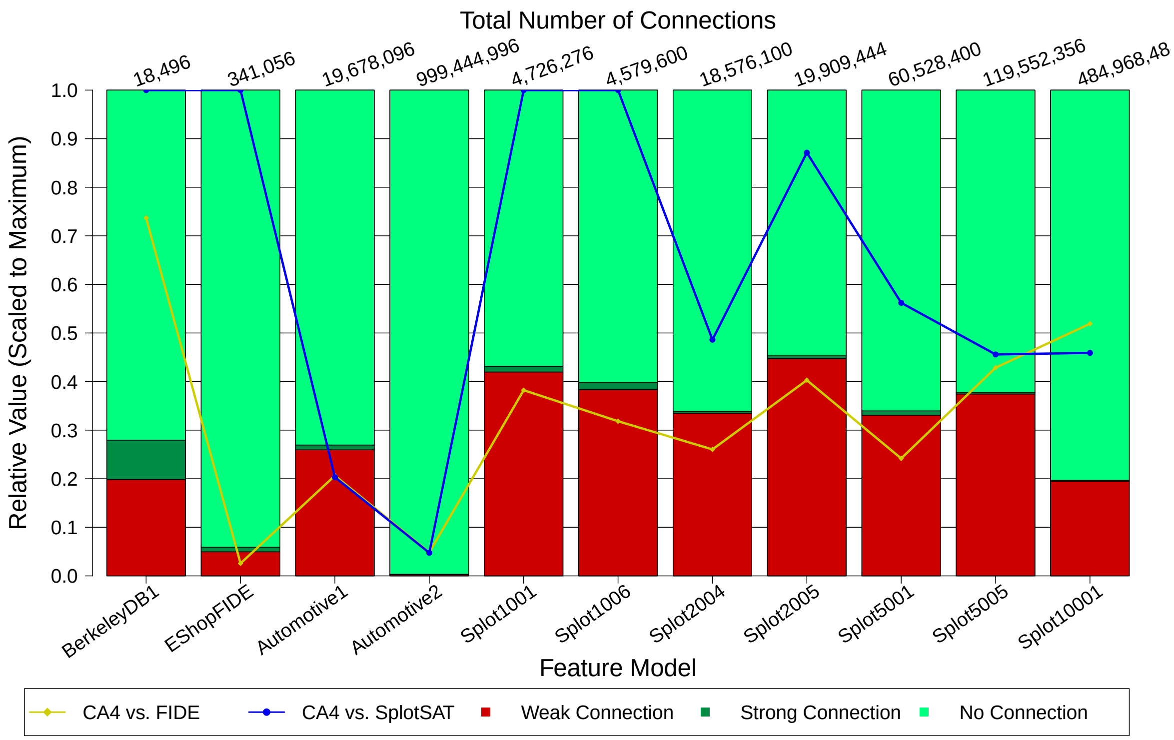

Figure 5.5: Number of connections between all nodes within a feature graph. Comparison of decision-propagation times for CA4 to FeatureIDE and SplotSAT. (Regarding the feature-model selection given in Table 5.1)

RQ1 - Faster Decision Propagation?

As we can see in Figure 5.2, the average computation time of our approach for the feature models Automotive1, Automotive2, Splot5005, and Splot10001 is significantly lower than the average computation time of the other evaluated configuration tools. Remarkably, in almost all cases our approach is faster than FeatureIDE. Though, we already expected this outcome, because our implementation is based on FeatureIDE and aims to reduce the number of complex propagation tests. By using more than one thread simultaneously, our approach is even able to outperform SplotSAT for feature models with 2,000 or more features and performs equally fast for feature models with only 1,000 features. In our evaluation, SplotBDD turned out to be unsuitable for larger feature models. Therefore, a serious comparison with our approach becomes obsolete. When we take a look at the absolute computation times of the configuration assistant, we can see that the maximum of all measured values is 18 seconds (rounded up), which comes from the Splot10001 feature model. For the real-world feature models Automotive1 and Automotive2, we got maximal values of 0.6 and 2.8 seconds, whereas the average time was at 0.1 seconds for both models. For the artificial models the average time was equal (Splot5001 - Splot5010) or lower (Splot1001 - Splot2010) than 1 second. Furthermore, when using 4 threads simultaneously, our approach performs approximately twice as fast.

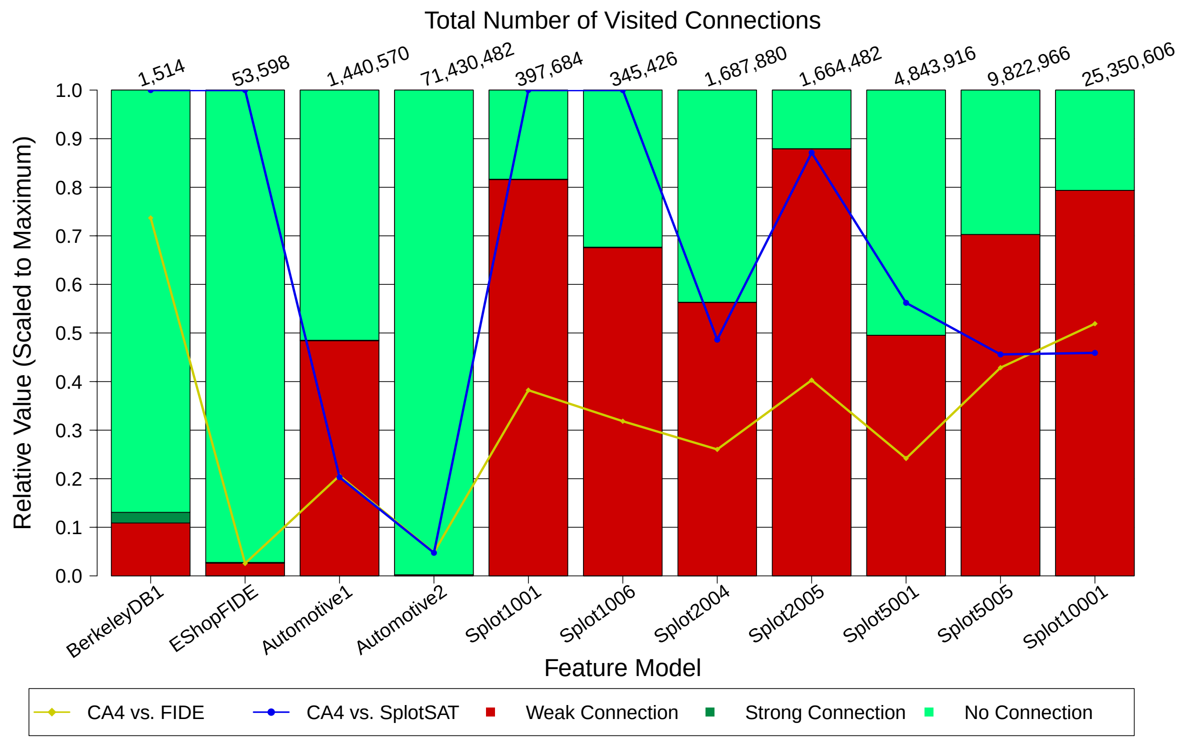

Figure 5.6: Number of visited connections during decision propagation. Comparison of decision-propagation times for CA4 to FeatureIDE and SplotSAT. (Regarding the feature-model selection given in Table 5.1)

In summary, we can conclude that our new approach is indeed capable of accelerating the decision propagation process, compared to other configuration tools. The absolute computation times also fortify the feasibility of our approach for an interactive configuration process. However, not all feature models are equally suited for our approach, which is the subject of our second research question.

RQ2 - Suitable Feature-Model Types?

From our evaluation results we can clearly see that our approach performs better, compared to the other configuration tools, when the total number of features increases. Especially the real-world feature models, Automotive1 and Automotive2, and the largest artificial feature model, Splot10001, benefit from our approach. However, when using only one thread for the configuration assistant, S.P.L.O.T. was usually faster for most of the artificial models. Since, our approach is based on FeatureIDE, in Figure 5.6, we can see a clear correlation between the number of visited connections in the feature graph and the performance compared to FeatureIDE. This correlation demonstrates the strong influence of weak connections in the feature graph to the performance of the configuration assistant. Thereby, it fortifies the importance of reducing the amount of weak connection within the feature graph.

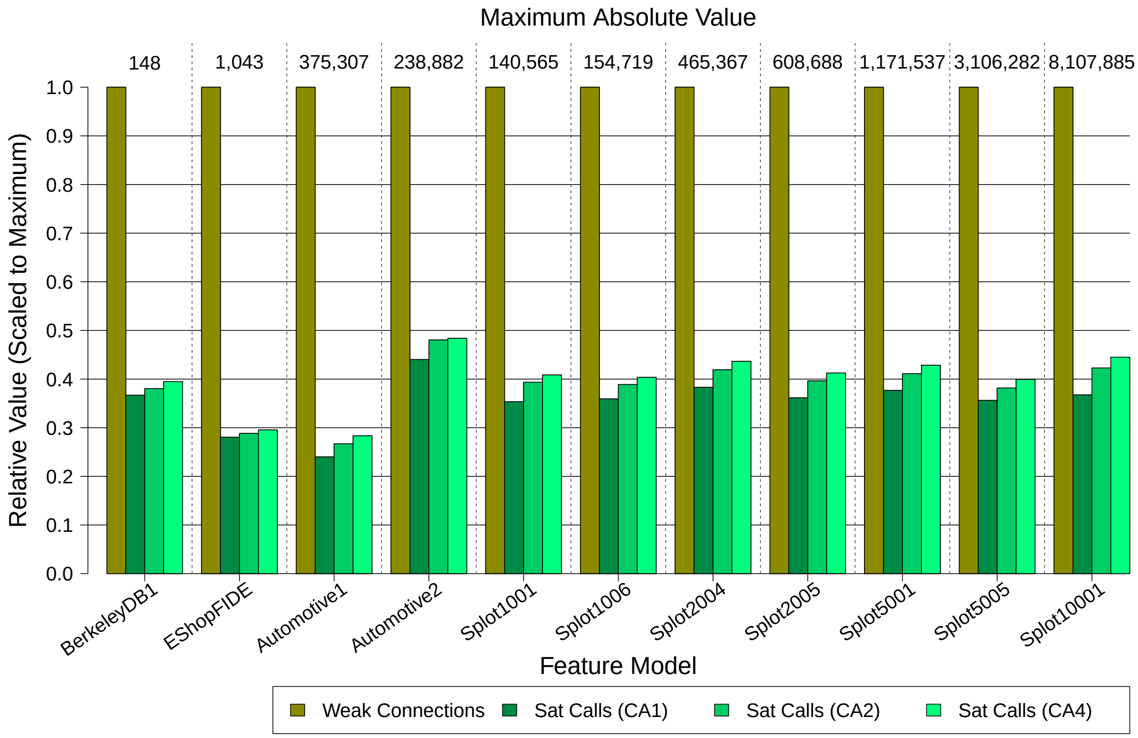

Figure 5.7: Number of weak connections and satisfiability tests during decision propagation (regarding the feature-model selection given in Table 5.1).

Independent from their number of features, both real-world feature models Automotive1 and Automotive2 seem to be well-suited for our approach, as the measured computation times are significantly lower compared to all other configuration tools. Although Automotive1 has a high number of cross-tree constraints (2,833) and also its constraint coverage is quite high (50.6%), it performs excellently when using our configuration assistant. A more detailed look at the feature model reveals that without exception all cross-tree constraints can be converted into 2-CNF. This circumstance reduces the amount of weak connections to about 25% of all possible connections. However, the statistical values of the second feature model are quite different. The constraint coverage of Automotive2 is similar to the coverage most of the artificial models (Splot1001 - Splot5010) and considerably lower than the coverage of Automotive1. In addition, there are fewer cross-tree constraints (948), which is still far more than for the Splot feature models. However, some of the constraints are more complex and involve up to 9 features. Thus, considering only the statistical values of the feature models, both look rather different. Their most obvious similarity is that both are designed by humans. Therefore, we assume that features that are contained in cross-tree constraints are not spread over the entire feature diagram, but are relatively close to each other, which has a positive influence on the complexity of the feature graph.

In conclusion, as far as we can infer from our evaluation, the configuration assistant is well-suited for highly large-scale feature models and feature models with simple feature dependencies. Unfortunately, we cannot make a definitive statement about the feasibility of different feature models from the measured statistical values alone. We presume that the most important influence is the overall design of the feature model and how separated single groups of features are. A structure that is designed by humans most likely leads to fewer arbitrary feature dependencies and consequently simplifies the corresponding feature graph.

RQ3 - Feature-Graph Passive Performance?

Comparing the initialization time of our configuration assistant with the time needed by the other configuration tools, we can see that CA1 is never the fastest tool for larger feature models. Either FeatureIDE or SplotSAT are faster than CA1 in most cases. However, the required time for the initialization phase is not disproportionately higher than those of the other configuration tools and has a maximal value of 98 seconds (for Automotive2) which is acceptable for an initial computation. In addition, the usage of multiple threads reduced the initialization time even further (70 seconds with 4 threads). Furthermore, since we implemented a load and store mechanism for the feature graph, we only have to compute the feature graph once and can use it henceforth, as long as the feature model is not modified.

Another concern of ours was the memory-space consumption of the feature graph. Indeed, the uncompressed memory space used by our feature-graph implementation grows quadratically in size and takes up several megabyte for large feature models (e.g., 250 megabyte, the maximum size in our evaluation). However, two things considerably mitigate the impact of these results. First, the constructed feature graphs are relatively sparse and, second, most of them have a very high compression potential. We assumed that a transitive closed graph would be more dense and, thus, stored the feature-graph data in an adjacency matrix. Considering our static analysis on the feature graphs, we probably would store the feature graph more efficiently, when using an adjacently list, which is more suited for these conditions. Furthermore, we can compress feature-graph data to save even more memory space. Unfortunately, we cannot use the LZMA compression for the main-memory storage, because we need a fast random access to the feature-graph data structure. Nevertheless, there exist other compression techniques that allow a transparent data access while still reducing the overall size. Moreover, we can use the shown compression technique when actually saving the feature graph to the hard drive. Therefore, the feature-graph file on the hard drive can make full-use of the shown compression rates.

Overall, we can conclude that the impact of the feature graph's passive performance is not as high as we suspected. The time required by our configuration assistant for the initialization phase is quite reasonable, especially when using more than one thread. In addition, although the memory consumption is relatively high with the current feature-graph implementation, it can potentially be reduced by using another underlying data structure or compression of the data.

General Conclusions

In Chapter 4, we proposed a modification to speed up the complex propagation test by using the model computed by the satisfiability solver to exclude certain queries. From Figure 5.7, we can infer that this modification saved over half the amount of calls to the satisfiability solver.

Another interesting detail from our evaluation regards the usage of multiple threads for the configuration assistant. Since a higher number of threads causes a larger overhead for initializing the satisfiability solvers, it is possible to lose performance for smaller feature models (e.g., BerkeleyDB and EShopFIDE). However, considering the low absolute values, measured for these feature models, the impact of the overhead becomes insignificant. In addition, when using feature models with 1,000 or more features, a higher number of threads always leads to a faster performance. Anyhow, due to the independent computation of the single selection states, we assumed an approximately 4 times faster performance, when using four threads simultaneously. In reality, however, we could only increased the overall performance by factor 2. Most likely this result follows from the lack of shared information between the satisfiability solver instantiations. Another modification that we described in Chapter 4 reduces the amount of calls to satisfiability solver by constantly updating the current solver model. In order to use the modification's full potential, the information about the current model must be shared among the single satisfiability solver instances. A too slow information spread is probably the reason for the unexpected performance impairment. This hypothesis is supported by the result in Table A.5, as we can see the increase in the number of complex propagation tests for a higher number of threads.

5.2.6 Threats to Validity

Like for all experimental evaluations, there exists certain threats to the validity of our results. Thus, we now address possible threats and explain how we tried to handle them in our evaluation. We differentiate between internal threats, which arise from our own evaluation set up and implementations and external threats, which are induced by other tools or implementations on which we rely.

Internal

In our evaluation, we evaluated just two large-scale, real-world feature model, since there are few large-scale real-world feature models freely available. Furthermore, we use artificial feature models. A randomly generated feature model might lack the structure from well designed feature models that are used in industry. Therefore, the composition of feature dependencies can differ from real-world model and, thus, bias our results. However, the artificial feature models were not chosen arbitrarily, but are present in the S.P.L.O.T. feature-model repository and are used in other evaluations as well. In addition, the feature models themselves are generated with different parameters and differ in size, number of cross-tree constraints, and constraint coverage.

To measure the computation time of the decision propagation, we used random configuration plans. This approach can induce a potential bias of the evaluation results. Unfortunately, it is practically impossible to test every potential configuration order. To mitigate the effect of a random bias, we used multiple samples and afterwards computed the arithmetic mean. However, the tested amount of configuration plans is just a small fraction of all possible plans and could still lead to biased values.

Another possible threat are bugs in our implemented prototype and evaluation tool. An unnoticed bug can produce invalid results and in addition falsifying the time measurements of our evaluation. Although we cannot guarantee the absence of bugs in our implementation, we successfully performed several unit tests that indicate a correct behavior of our implementation. Additionally, we compared the decision-propagation results from our configuration assistant with the results from other tools. In all cases, we received equal results. Thus, we can be relatively sure that our prototype works correctly.

External

We converted feature models from SXFM to the FeatureIDE XML format and vice-versa. Since, we rely on the implementation of FeatureIDE for importing and exporting feature models, we cannot guarantee an absolutely accurate conversion. However, after each configuration process we received an equal configuration from every configuration tool, which is a very good indicator that the corresponding input feature models represented the same feature dependencies.

In our evaluation tool, we used an older version of Sat4j. Hence, the real values for the execution times might be different. However, we used the same version for every configuration tool. Thus, a potential bias would apply to all configuration tools as well.

6. Related Work

Configuration of software product lines is a vital part of SPLE. Thus, there exist numerous works addressing the topic of the configuration process. In this chapter, we present certain publications that are related to our approach and point out major similarities and differences. In particular, we examine other approaches to perform the configuration process with and without using decision propagation.

6.1 Approaches for Decision Propagation

In the following, we show implementations of decision propagation in the interactive configuration process that are not based on satisfiability solvers, but use other reasoning techniques. For instance, in our evaluation, we used the configuration tool of S.P.L.O.T., which has two different configuration engines, based on satisfiability solvers and binary decision diagrams (BDDs) [MBC09].

Mendonça et al. showed how feature-model dependencies can be translated into BDDs to apply efficient reasoning [MWCC08]. The main problem of constructing a BDD is to find a suitable variable ordering to minimize its final size. However, once a BDD is created, subsequent queries to it can be answered relatively fast. The usage of BDDs for decision propagation was investigated by Hadzic et al. [HSJ+04]. Their general idea is to solve the difficult NP-complete problem before starting the actual configuration process (i.e., in the initialization phase). Thus, they construct a BDD and use it during the interactive configuration process. However, as we saw in our evaluation for the SplotBDD configuration tool, this approach does not scale for large feature models.

In our thesis, we implemented the complex propagation test by considering the dependencies of a feature model as a satisfiability problem (SAT). Another method is the translation of feature dependencies to a constraint satisfaction problem (CSP) as shown by Benavides et al. [BTRC05]. By contrast to a SAT-based method, a CSP allows the usage of finite variable domains rather than just boolean variables. To apply CSPs to an interactive configuration process, Amilhastre et al. propose assumption-based CSPs (A-CSP), an extension to classic CSPs that adds a set of assumptions, which originate from the users decisions during the configuration process [AFM02]. The A-CSP can be used to determine the remaining domains for each variable, which can still lead to a valid configuration.

Mendonça propose a method for decision propagation based only on the feature tree, which they call “Feature Tree Reasoning System” (FTRS) [Men09]. Using the graph-based algorithms of the FTRS, decision propagation can be executed with linear time complexity. However, in our thesis, we consider arbitrary cross-tree constraints, which are not applicable to this method. The authors are aware of this problem and propose a hybrid-system, the “Feature Tree Reasoning System” (FMRS), which extends their FTRS and is capable of handling additional cross-tree constraints. The general concept is to combine the FTRS with a more powerful solver engine and perform an interleaved reasoning process. If, for our approach, we would use transitive reduction and an intelligent selection algorithm that uses an efficient variable ordering, we would presumably achieve a similar behavior like the FMRS. Anyway, our current implementation separates the fast evaluation of strong connections within the feature graph and the slow, SAT-based evaluation of weak connections.

6.2 Approaches for Error Resolution

In Chapter 2, we talked about other methods to specify a valid configuration, besides the interactive configuration process. Nevertheless, in case the SPL developers do not use decision propagation in their configuration process, there exists the possibility of creating invalid configurations. In those cases, the developers have to resolve the configuration errors, which is difficult for large-scale feature models without tool support. An approach called “CURE” that resolves errors in an invalid configuration is introduced by White et al. [WSB+08]. CURE considers the configuration process as CSP and is capable of finding the minimal set of features that should be selected or deselected to make the current configuration valid. The authors especially focus on configurations that are created through staged configurations, since this process involves multiple developers and, thereby, increases the possibility of configuration errors. A configuration tool that support this kind of error detection in configurations is included in the FaMa framework [BSTRC07].

7. Conclusion

SPLE is used in software development to efficiently build new software products by reusing software artifacts (i.e., features). Valid combinations of different features that can be composed to a working software product are defined by a feature model. An important part of SPLE is the configuration process, in which the developer specifies a valid feature combination (i.e., a configuration). To support the developer in this process, certain configuration tools offer an interactive configuration process, which enforces a valid configuration state by updating the current configuration based on the decisions made by the developer (i.e., decision propagation). However, decision propagation for large-scale feature models is challenging, as it is an NP-complete problem. In our thesis, we addressed the problem of efficiently performing an interactive configuration process on large-scale feature models.

Contributions

We introduced a new concept for representing feature dependencies in a data structure based on implication graphs, the feature graph. We used the feature graph as basis for our new approach the configuration assistant. In addition, we proposed two alternative restructuring strategies for the feature graph. To evaluate our approach, we prototypically implemented the configuration assistant and embedded it in the SPLE framework FeatureIDE. Additionally, we raised three research questions to investigate the properties of our new approach and answered them with the help of our evaluation results. For this, we compared our implementation with two other configuration tools, FeatureIDE and S.P.L.O.T..

Research Results

In our evaluation, we discovered that our approach is well suited for large-scale feature models. The performance for the interactive configuration process is reasonable for all of our evaluated feature models. Thus, we are able to positively answer our first research question, whether we can profit from our new data structure. We can also draw a positive conclusion for our third research question concerning memory consumption and construction time of the feature graph. Our evaluation showed that the time required for constructing the feature graph was not disproportionately higher (and partly even lower) than the initialization time of other configuration tools. Moreover, although we saw a rather high memory consumption for our feature-graph data structure, we could also measure very high compression rates when applying a standard compression technique to saved feature-graph files. Unfortunately, due to insufficient data, we are not able to fully answer our second research question, for which we would require further case studies and experiments. However, as we mentioned above, the performance benefits of our approach increases with the size of the used feature model. Based on our evaluation results, we can further assume that a well-structured feature model increases the overall performance for decision propagation.

All in all, the evaluation results met our expectations. Yet, we were surprised by some of our results, both negatively and positively. A minor disappointment was the application of multi-threading, whose performance benefits stayed behind our expectations. We assumed a directly proportional performance benefit with an increasing number of threads. However, the real measured values only indicated a performance benefit by at most half the expected amount. By contrast, a positive surprise were the moderate initialization times of our approach and the extremely good compression rates of most feature graphs. Initially, we suspected a high computational effort for the feature-graph construction and restructuring and a large amount of required memory space. However, compared to other configuration tools, our configuration assistant performs quite well in its initialization phase and by using certain compression techniques it is possible to effectively reduce the feature-graph size.

In conclusion, we can say that our configuration assistant is a real benefit for the interactive configuration process of large feature models. Although we could only implement it prototypically, in the context of this thesis, our evaluation showed the potential of its core concept. Thus, we look forward improve both the configuration assistant and our feature-graph data structure in future work.

8. Future Work

In this chapter, we suggest several topics that can build upon our contributions in this thesis. On the one hand, we discuss multiple concepts that can be used to enhance our configuration assistant and the feature graph. On the other hand, we propose other possible applications for our feature graph, where we assume it can be useful. In addition, we point out related questions that we are interested in and that could be subjects of further research.

8.1 Feature-Graph Improvements

In Chapter 4, we described how we realized the configuration assistant and, within it, the feature-graph data structure. However, we can think of several improvements that might lead to an even faster performance.

Editing Feature Models

For a faster initialization phase of our configuration assistant, we save an already computed feature graph and use it consistently for each configuration process. However, when the corresponding feature model changes, the computed feature graph becomes obsolete. In our current implementation, we then have to recompute the entire feature graph. As our results in Chapter 5 show, the initial computation requires some time for larger feature models (i.e., in our evaluation up to 1 minute). In order to avoid a recomputation of the feature graph, we require a mechanism to adapt it, when there are changes in the feature model. Thus, we can raise the question, whether it is possible to efficiently adapt a feature graph for certain changes in the feature model. Furthermore, we can generalize the question to arbitrary feature-model changes.

Transitive Reduction

In Chapter 3 we introduced the restructuring strategy of transitive reduction for the feature graph. We assume that such an approach would be even faster and is capable of compensating a higher constraint coverage to a certain extent. Presumably, the selection algorithm would require more time for traversing the feature graph, than with precomputed transitive edges. However, due to the thesis' time constraints we were not able to evaluate this approach. Nevertheless, we are interested in the performance of our configuration assistant using transitive reduction and the corresponding selection algorithm. Furthermore, when we realize both strategies, there are two competing implementations, which raises the following question. Is one strategy outperforming the other in every case or does the performance dependent on the individual feature model?

Detect Strong Connections

In Chapter 3, we pointed out that the more weak connections are in the feature graph the more time our configuration assistant would need to finish the decisions propagation. With the result from our evaluation, we could confirm this assumption (cf. Chapter 5). We also pointed out that we use a rather simple approach of finding strong connections among cross-tree constraints. For each cross-tree constraint, we transform the corresponding propositional formula to CNF and add strong connections for all 2-CNF clauses. With this method, we might overlook strong connections that are not explicitly stated in the feature dependencies. Thus, we require more sophisticated ways of determining strong connections.

A possible method to find more strong connections is the application of the atomic-set analysis. However, the determination of atomic sets requires much computational effort and, thus, would drastically increase the time required for the initialization phase of the configuration assistant. Hence, we ask the following question. Is there an efficient way to find all or, at least, most of the possible strong connections in a feature model?

Alternative Complex Propagation Tests

For our evaluation, we realized our approach with a complex-propagation-test implementation based on FeatureIDE. However, there exist also other approaches to implement the complex propagation test, which may lead to a faster overall performance of the configuration assistant. Thus, it is reasonable to evaluate our approach with other implementations of the complex propagation test. As we can see from the results of our evaluation, S.P.L.O.T. mostly outperforms FeatureIDE in terms of required computation time. Hence, we would like to implement and evaluate a combination of S.P.L.O.T. and our approach.

Similar to the implementation of different restructuring strategies for the feature graph, we would like to know if there is a best implementation for the complex propagation test. If there is no implementation that provides an adequate performance for all feature models, it raises the following question. Is it possible to efficiently estimate the most suitable complex-propagation-test implementation for each feature model?

8.2 Feature-Graph Applications

We used the feature graph to improve the performance of the interactive configuration process. However, we can also imagine other application that can benefit from such a data structure. We assume that our feature graph can be used for visualization purposes and as basis for other feature-model analyses.

Feature-Model Visualization

Since, the feature graph is a directed graph, we assume that it is well suited to visualize feature models, such as feature diagrams do. Thus, we are interested, if there is a convenient way of representing a feature graph to a developer to illustrate the direct and indirect dependencies of all features. If so, it could be used for manual analyses and traversal in large feature models, which are both helpful for maintenance purposes. Thus, the resulting question is the following. How can a feature graph be used to visualize feature dependencies and thereby support an SPL developer?

Feature-Model Analyses

We already mentioned that the results of the atomic-set analysis can be used to improve the feature graph. However, it may also be possible to utilize the feature graph to improve the performance of the atomic set analysis. Since, we are pre-computing certain feature dependencies, these information can be used to reduce the number of satisfiability tests in a SAT-based implementation of the atomic-set analysis. It might even be possible to implement an iterative process for mutual computation of atomic sets and the feature graph in a way that both benefit from the other. Therefore, we raise the following, last question. In which way can a feature graph be used to support feature-model analyses?

A. Appendix

In the following, we list the complete datasets for each measurement in our evaluation (see Section 5.1).

Model	#Features	#Groups Alternative	#Groups OR	#Constraints	Constraint Coverage (%)
Dell	46	8	0	110	80.4
BerkeleyDB1	76	8	4	20	42.1
BerkeleyDB2	119	3	1	68	81.5
Violet	101	1	11	27	66.3
EShopSplot	287	0	39	21	11.8
EShopFIDE	326	0	39	21	10.4
Automotive1	2,513	407	43	2,833	50.9
Automotive2	17,365	1,165	111	948	6.5
Splot1001	1,120	62	75	100	8.4
Splot1002	1,096	64	72	100	8.7
Splot1003	1,104	61	67	100	8.6
Splot1004	1,090	56	67	100	8.8
Splot1005	1,103	78	56	100	8.9
Splot1006	1,109	62	76	100	8.7
Splot1007	1,107	74	67	100	8.3
Splot1008	1,106	75	60	100	8.4
Splot1009	1,106	61	65	100	8.4
Splot1010	1,106	62	75	100	8.6
Splot2001	2,223	140	121	100	6.7
Splot2002	2,230	132	139	100	7.1
Splot2003	2,223	129	134	100	6.6
Splot2004	2,212	141	128	100	6.9
Splot2005	2,236	145	136	100	7.1
Splot2006	2,219	131	140	100	6.7
Splot2007	2,204	131	111	100	6.7
Splot2008	2,242	132	155	100	7.0
Splot2009	2,206	133	114	100	7.2
Splot2010	2,229	139	127	100	7.0
Splot5001	5,545	339	336	150	5.3
Splot5002	5,523	291	352	150	5.2
Splot5003	5,519	322	322	150	5.4
Splot5004	5,556	343	340	150	5.3
Splot5005	5,543	350	324	150	5.3
Splot5006	5,514	349	317	150	5.4
Splot5007	5,503	316	327	150	5.4
Splot5008	5,524	327	328	150	5.3
Splot5009	5,529	316	334	150	5.1
Splot5010	5,518	326	317	150	5.4
Splot10001	11,065	676	617	100	2.4

Table A.1: Statistical values for used feature models.

Model	Initialization Time (in ms)					
	CA1	CA2	CA4	FeatureIDE	SplotBDD	SplotSAT
Dell	5	5	5	12	9	9
BerkeleyDB1	10	9	9	19	24	21
BerkeleyDB2	13	13	13	28	15	14
Violet	9	8	9	17	14	10
EShopSplot	21	17	18	39	86	18
EShopFIDE	22	17	16	42	111	20
Automotive1	1,430	1,143	1,056	1,396	218,663	1,410
Automotive2	98,039	81,966	69,784	53,003	-	598,512
Splot1001	316	262	245	312	176,912	135
Splot1002	296	247	221	288	110,841	145
Splot1003	313	261	238	299	530,893	145
Splot1004	323	269	244	296	388,923	138
Splot1005	331	278	255	301	173,477	151
Splot1006	314	261	241	300	627,256	141
Splot1007	327	274	250	313	113,109	139
Splot1008	301	244	219	296	1,901,977	157
Splot1009	315	260	237	303	363,395	143
Splot1010	335	284	259	298	133,441	144
Splot2001	1,389	1,162	1,084	1,141	297,261	598
Splot2002	1,538	1,299	1,230	1,077	163,243	645
Splot2003	1,210	978	904	1,022	339,935	581
Splot2004	1,267	1,036	957	1,023	597,840	726
Splot2005	1,326	1,086	1,006	1,124	330,369	693
Splot2006	1,532	1,293	1,218	1,054	160,823	613
Splot2007	1,157	928	864	1,035	822,147	666
Splot2008	1,351	1,112	1,028	1,067	2,392,786	600
Splot2009	1,059	840	769	946	296,490	583
Splot2010	1,315	1,087	1,015	1,068	329,165	694
Splot5001	4,490	3,593	3,401	4,769	-	3,782
Splot5002	8,217	6,769	6,338	5,955	-	6,370
Splot5003	9,023	7,831	7,386	6,007	-	6,562
Splot5004	8,542	7,388	6,880	5,969	-	6,658
Splot5005	6,934	5,790	5,285	5,929	-	7,508
Splot5006	9,514	8,265	7,748	6,106	-	7,063
Splot5007	7,747	6,458	5,934	6,141	-	6,185
Splot5008	7,737	6,576	5,997	6,126	-	6,597
Splot5009	7,936	6,664	6,134	6,258	-	6,039
Splot5010	9,607	8,381	7,861	6,071	-	6,669
Splot10001	43,958	39,267	34,781	26,291	-	50,659

Table A.2: Time required by each configuration tool for its initalization phase.

Model	FeatureIDE (in ms)			SplotSAT (in ms)			SplotBDD (in ms)			CA1 (in ms)		
	Max	∅	∑	Max	∅	∑	Max	∅	∑	Max	∅	∑
Dell	2	0	3	1	0	0	0	0	0	2	0	3
False	2	0	3	0	0	0	0	0	0	2	0	5
True	2	0	4	1	0	1	0	0	0	2	0	2
Random(∅)	2	0	3	1	0	0	0	0	0	2	0	4
BerkeleyDB1	15	0	19	4	0	1	1	0	0	36	0	9
False	3	1	10	1	0	2	1	0	1	2	0	8
True	2	0	29	1	0	1	0	0	0	2	0	4
Random(∅)	15	0	19	4	0	1	0	0	0	36	0	17
BerkeleyDB2	10	0	23	4	0	3	1	0	0	9	0	22
False	4	0	17	4	0	5	0	0	0	4	0	12
True	4	1	30	1	0	2	0	0	0	7	1	35
Random(∅)	10	0	24	4	0	2	1	0	0	9	0	21
Violet	13	1	65	3	0	0	0	0	0	3	0	7
False	13	1	42	0	0	0	0	0	0	2	0	11
True	3	1	96	0	0	0	0	0	0	1	0	3
Random(∅)	12	1	58	3	0	0	0	0	0	3	0	8
EShopSplot	26	5	683	3	0	5	60	1	285	6	0	26
False	21	6	320	1	0	5	25	0	37	4	0	21
True	26	5	1,165	3	0	6	51	3	614	3	0	19
Random(∅)	18	5	565	1	0	5	60	1	205	6	0	38
EShopFIDE	22	6	739	2	0	7	57	2	301	5	0	18
False	14	7	357	1	0	7	26	0	38	5	0	23
True	22	6	1,245	2	0	8	55	3	645	2	0	19
Random(∅)	18	6	616	2	0	8	57	2	220	4	0	13

Model	FeatureIDE (in ms)			SplotSAT (in ms)			SplotBDD (in ms)			CA1 (in ms)		
	Max	∅	∑	Max	∅	∑	Max	∅	∑	Max	∅	∑
Automotive1	776	238	106,697	826	227	108,191	162,375	2,491	1,699,066	590	103	49,544
False	505	173	43,716	812	104	26,317	94,051	558	170,966	420	46	11,651
True	776	283	156,939	812	319	176,868	151,080	2,233	2,026,185	590	160	88,713
Random(∅)	687	257	119,437	826	259	121,390	162,375	4,681	2,900,047	426	102	48,268
Automotive2	* 66,762	62,239	7,248,604	* 508,729	499,464	7,491,963	-	-	-	2,762	105	921,950
False	* 63,705	61,486	7,255,410	* 507,927	505,594	7,583,921	-	-	-	2,434	170	2,251,878
True	* 65,883	63,618	7,252,527	* 508,729	505,867	7,588,011	-	-	-	2,762	73	253,516
Random(∅)	* 66,762	61,614	7,237,876	* 508,076	486,930	7,303,958	-	-	-	2,606	74	260,457
Splot1001	175	49	15,142	73	12	4,232	102,639	2,754	1,004,708	152	41	13,325
False	130	37	5,654	55	6	983	99	4	210	98	23	3,514
True	175	61	27,171	71	18	8,286	102,304	5,513	2,425,874	152	59	26,191
Random(∅)	160	50	12,603	73	13	3,428	102,639	2,747	588,041	143	40	10,270
Splot1002	176	51	14,108	75	15	4,242	44,869	981	317,409	108	23	6,404
False	121	45	8,138	66	11	1,959	4,449	308	43,466	103	19	3,526
True	176	58	21,243	74	19	6,997	44,869	1,620	638,415	107	27	9,950
Random(∅)	147	50	12,943	75	14	3,771	44,585	1,016	270,346	108	22	5,737
Splot1003	154	61	17,059	80	18	5,209	402,356	6,776	1,918,316	131	38	11,148
False	137	61	7,772	78	18	2,296	195,801	2,896	234,616	118	33	4,251
True	154	61	27,890	79	19	8,639	400,505	6,898	3,456,157	131	43	19,629
Random(∅)	154	60	15,516	80	18	4,694	402,356	10,535	2,064,177	130	37	9,564
Splot1004	164	58	18,895	79	16	5,581	173,317	6,398	1,893,554	159	45	15,722
False	129	54	9,005	76	14	2,346	142,304	5,927	735,002	134	32	5,332
True	163	63	30,390	79	20	9,719	173,218	6,604	3,282,484	159	59	28,341
Random(∅)	164	56	17,291	79	15	4,679	173,317	6,664	1,663,176	147	44	13,495

Model	FeatureIDE (in ms)			SplotSAT (in ms)			SplotBDD (in ms)			CA1 (in ms)		
	Max	∅	∑	Max	∅	∑	Max	∅	∑	Max	∅	∑
Splot1005	149	45	10,312	88	13	3,166	101,070	1,179	303,295	135	34	8,182
False	116	41	5,786	63	9	1,345	247	6	692	112	24	3,339
True	149	44	13,448	87	14	4,413	99,796	1,364	488,406	134	40	12,212
Random(∅)	143	49	11,703	88	15	3,741	101,070	2,167	420,787	135	37	8,996
Splot1006	182	59	16,562	91	17	5,052	479,197	8,454	2,782,326	144	43	12,554
False	120	55	9,433	59	14	2,403	1,911	180	24,753	121	33	5,693
True	182	59	23,115	91	19	7,478	465,208	10,836	4,789,527	144	49	19,331
Random(∅)	159	64	17,139	77	19	5,276	479,197	14,346	3,532,700	143	47	12,638
Splot1007	153	57	13,694	78	19	4,625	45,867	548	108,388	138	45	10,852
False	151	52	8,689	76	17	2,873	44,588	863	134,742	138	39	6,477
True	137	61	17,930	77	20	6,107	5,226	150	59,880	131	49	14,354
Random(∅)	153	58	14,465	78	20	4,896	45,867	633	130,543	138	47	11,725
Splot1008	160	58	16,570	93	23	6,736	898,950	302,839	4,041,079	148	39	11,519
False	128	48	8,621	85	16	2,943	724,342	5,660	764,107	124	28	5,101
True	160	70	27,128	92	30	11,753	898,950	889,052	8,001,471	148	54	20,832
Random(∅)	147	57	13,962	93	22	5,512	891,581	13,806	3,357,659	130	34	8,626
Splot1009	193	53	15,857	109	17	5,285	292,851	5,105	1,780,808	190	42	13,121
False	121	46	7,564	66	11	1,871	344	7	615	118	30	4,996
True	193	60	24,774	109	22	9,086	292,065	7,559	3,658,590	190	54	22,268
Random(∅)	159	55	15,235	84	17	4,900	292,851	7,751	1,683,220	154	43	12,100
Splot1010	157	41	9,479	80	11	2,579	52,568	301	101,575	138	35	8,757
False	113	27	2,152	61	6	518	16	2	234	106	18	1,460
True	157	53	19,586	80	14	5,204	51,259	643	256,888	138	53	19,343
Random(∅)	131	42	6,700	76	12	2,016	52,568	257	47,604	138	34	5,468

Model	FeatureIDE (in ms)			SplotSAT (in ms)			SplotBDD (in ms)			CA1 (in ms)		
	Max	∅	∑	Max	∅	∑	Max	∅	∑	Max	∅	∑
Splot2001	650	202	106,677	451	78	44,787	246,302	2,613	1,717,759	556	170	96,987
False	514	156	41,042	362	42	11,076	6,759	101	21,989	498	92	24,281
True	578	248	189,040	451	118	90,418	233,104	4,828	3,906,315	544	255	194,129
Random(∅)	650	202	89,950	447	73	32,869	246,302	2,911	1,224,975	556	162	72,553
Splot2002	957	195	88,595	495	68	31,669	95,791	1,623	943,186	581	138	66,639
False	485	181	47,497	348	59	15,462	676	87	17,693	457	101	26,565
True	957	211	139,771	488	79	52,543	93,356	3,422	2,272,364	574	181	119,769
Random(∅)	656	193	78,517	495	65	27,003	95,791	1,360	539,501	581	131	53,583
Splot2003	699	242	151,631	753	102	66,590	277,916	3,049	2,319,419	458	121	78,214
False	539	202	63,557	386	73	23,136	537	31	7,231	436	93	29,501
True	681	275	258,029	394	126	118,815	277,916	4,922	4,780,227	455	150	141,059
Random(∅)	699	250	133,309	753	108	57,820	277,082	4,193	2,170,801	458	120	64,084
Splot2004	692	245	135,552	541	133	73,735	399,527	2,812	2,390,917	510	153	88,433
False	527	236	65,122	496	129	35,587	7,111	489	90,076	469	133	36,650
True	656	260	226,320	540	143	124,611	399,527	7,321	6,816,335	494	182	158,542
Random(∅)	692	238	115,216	541	126	61,009	392,398	627	266,342	510	145	70,109
Splot2005	931	218	118,029	548	97	56,101	234,657	2,665	1,717,720	599	209	119,133
False	512	167	39,636	492	56	13,383	3,221	137	21,809	547	138	32,669
True	702	264	220,836	546	134	112,544	228,160	4,050	3,637,719	598	286	239,036
Random(∅)	931	222	93,615	548	100	42,378	234,657	3,807	1,493,633	599	203	85,694
Splot2006	633	206	94,188	383	66	31,850	91,800	983	550,421	508	131	63,011
False	465	195	55,131	244	52	14,735	1,325	30	8,120	412	102	28,817
True	614	216	140,559	383	81	52,624	89,700	1,606	1,055,650	492	163	106,336
Random(∅)	633	206	86,874	383	66	28,191	91,800	1,313	587,495	508	127	53,880

Model	FeatureIDE (in ms)			SplotSAT (in ms)			SplotBDD (in ms)			CA1 (in ms)		
	Max	∅	∑	Max	∅	∑	Max	∅	∑	Max	∅	∑
Splot2007	988	207	133,378	468	82	57,193	582,804	173,330	3,625,967	504	131	92,983
False	479	170	47,331	360	50	14,016	842	58	12,183	432	77	21,385
True	988	247	247,583	463	115	115,415	* 567,862	425,285	7,229,859	504	194	194,036
Random(∅)	687	204	105,220	468	80	42,148	582,804	94,647	3,635,860	499	121	63,529
Splot2008	956	230	150,538	430	88	61,130	2,106,644	701,488	4,980,964	531	151	103,977
False	514	193	72,858	369	56	21,318	1,307,254	5,225	1,369,153	444	109	41,128
True	693	278	263,805	412	124	117,964	* 2,106,644	1,120,224	7,841,572	510	214	202,600
Random(∅)	956	220	114,953	430	84	44,109	2,085,719	979,014	5,732,168	531	129	68,203
Splot2009	620	166	76,181	339	51	24,886	240,857	77,877	3,353,362	471	112	55,083
False	419	126	28,602	228	26	6,045	799	73	12,781	343	64	14,536
True	620	190	128,617	308	65	44,341	* 230,649	226,239	7,239,654	471	154	104,446
Random(∅)	558	181	71,325	339	61	24,273	240,857	7,318	2,807,652	456	117	46,267
Splot2010	722	186	97,149	524	71	40,456	182,578	689	469,348	623	153	87,266
False	469	149	28,906	316	45	8,901	709	9	1,383	426	99	19,293
True	722	236	204,157	523	105	91,106	182,578	1,309	1,177,494	623	228	197,433
Random(∅)	608	174	58,384	524	63	21,361	179,022	748	229,168	604	133	45,072
Splot5001	2,644	1,073	985,222	2,293	457	436,579	-	-	-	1,952	493	463,505
False	2,371	1,016	502,946	2,249	383	190,047	-	-	-	1,952	443	219,428
True	2,567	1,168	1,633,818	2,208	554	775,285	-	-	-	1,781	578	808,767
Random(∅)	2,644	1,035	818,902	2,293	433	344,405	-	-	-	1,856	457	362,321
Splot5002	3,249	1,207	1,699,190	5,720	790	1,239,173	-	-	-	3,442	1,014	1,532,622
False	2,942	945	537,172	3,347	343	195,127	-	-	-	2,973	622	353,488
True	3,047	1,416	3,057,141	5,339	1,119	2,416,896	-	-	-	3,392	1,397	3,015,787
Random(∅)	3,249	1,259	1,503,258	5,720	907	1,105,496	-	-	-	3,442	1,025	1,228,593

Model	FeatureIDE (in ms)			SplotSAT (in ms)			SplotBDD (in ms)			CA1 (in ms)		
	Max	∅	∑	Max	∅	∑	Max	∅	∑	Max	∅	∑
Splot5003	3,026	1,133	1,127,008	5,345	671	667,388	-	-	-	3,278	939	969,619
False	2,939	1,045	524,722	5,345	624	313,350	-	-	-	3,255	799	401,355
True	2,696	1,186	1,786,257	3,969	703	1,059,519	-	-	-	2,748	1,117	1,682,889
Random(∅)	3,026	1,167	1,070,046	5,317	687	629,296	-	-	-	3,278	900	824,615
Splot5004	3,415	1,250	1,540,107	5,864	967	1,291,224	-	-	-	3,549	1,035	1,357,664
False	3,047	1,088	544,316	5,266	694	347,182	-	-	-	3,176	760	380,148
True	3,198	1,434	2,831,010	5,834	1,332	2,628,958	-	-	-	3,519	1,369	2,701,887
Random(∅)	3,415	1,227	1,244,996	5,864	875	897,534	-	-	-	3,549	976	990,958
Splot5005	3,818	1,343	1,709,376	6,647	1,232	1,595,739	-	-	-	4,061	1,102	1,475,103
False	3,599	1,089	542,519	6,526	921	459,028	-	-	-	3,965	785	391,104
True	3,701	1,504	3,102,131	6,554	1,419	2,926,606	-	-	-	4,026	1,390	2,866,836
Random(∅)	3,818	1,437	1,483,478	6,647	1,356	1,401,584	-	-	-	4,061	1,130	1,167,371
Splot5006	3,200	1,163	1,289,228	5,114	691	790,159	-	-	-	3,155	905	1,087,126
False	2,972	1,031	535,206	4,749	580	301,345	-	-	-	3,047	637	330,955
True	3,041	1,349	2,383,182	4,180	866	1,530,988	-	-	-	2,897	1,264	2,233,261
Random(∅)	3,200	1,109	949,297	5,114	625	538,144	-	-	-	3,155	813	697,163
Splot5007	3,408	1,407	1,830,278	5,522	1,082	1,514,747	-	-	-	3,701	1,258	1,716,218
False	3,166	1,277	739,755	5,071	823	476,547	-	-	-	3,388	1,019	590,543
True	3,375	1,653	3,400,197	5,509	1,518	3,121,566	-	-	-	3,595	1,657	3,407,865
Random(∅)	3,408	1,292	1,350,882	5,522	906	946,129	-	-	-	3,701	1,099	1,150,247
Splot5008	3,413	1,251	1,492,908	5,902	913	1,179,150	-	-	-	3,528	1,074	1,384,742
False	2,998	1,052	578,799	5,479	619	340,661	-	-	-	3,307	717	394,361
True	3,413	1,441	2,686,314	5,888	1,271	2,369,048	-	-	-	3,525	1,495	2,786,327
Random(∅)	3,304	1,259	1,213,612	5,902	850	827,742	-	-	-	3,528	1,010	973,538

Model	**FeatureIDE (in ms)**			**SplotSAT (in ms)**			**SplotBDD (in ms)**			**CA1 (in ms)**		
	Max	∅	∑	Max	∅	∑	Max	∅	∑	Max	∅	∑
Splot5009	3,466	1,358	1,596,608	5,458	940	1,187,515	-	-	-	3,807	1,357	1,736,957
False	2,982	1,293	505,872	3,396	767	300,057	-	-	-	3,029	1,077	421,172
True	3,282	1,476	3,073,879	5,168	1,171	2,439,119	-	-	-	3,807	1,741	3,626,307
Random(∅)	3,466	1,306	1,210,074	5,458	882	823,369	-	-	-	3,797	1,254	1,163,392
Splot5010	4,045	1,370	1,778,267	5,870	914	1,308,038	-	-	-	4,094	1,105	1,511,933
False	3,180	1,126	530,462	3,349	528	248,699	-	-	-	2,662	803	378,454
True	4,045	1,564	3,347,494	5,870	1,235	2,642,244	-	-	-	4,068	1,416	3,029,596
Random(∅)	3,897	1,420	1,456,847	5,865	980	1,033,173	-	-	-	4,094	1,095	1,127,749
Splot10001	* 13,992	6,152	9,429,971	* 48,342	14,374	10,071,199	-	-	-	* 17,880	6,194	8,985,031
False	13,181	4,864	2,928,597	44,547	5,851	3,522,395	-	-	-	17,880	3,942	2,373,088
True	* 13,810	8,369	18,002,079	* 47,655	31,112	18,014,344	-	-	-	* 16,959	9,972	18,000,842
Random(∅)	13,992	5,224	7,359,238	48,342	6,160	8,676,859	-	-	-	17,330	4,669	6,581,164

Table A.3: Decision-propagation times for evaluated configuration tools.

Model	CA2 (in ms) Max	CA2 ∅	CA2 ∑	CA4 (in ms) Max	CA4 ∅	CA4 ∑
Dell	3	0	3	21	0	5
False	2	0	3	2	0	4
True	3	0	3	3	0	3
Random(∅)	3	0	3	21	1	8
BerkeleyDB1	11	0	8	30	0	17
False	5	1	11	2	0	8
True	2	0	3	30	0	32
Random(∅)	11	0	10	6	0	12
BerkeleyDB2	7	0	18	6	1	30
False	3	0	10	5	1	19
True	4	0	25	6	1	46
Random(∅)	7	0	20	5	1	25
Violet	19	0	10	42	0	12
False	2	0	12	2	0	11
True	1	0	1	1	0	2
Random(∅)	19	0	18	42	0	23
EShopSplot	6	0	27	12	0	30
False	4	0	20	11	0	34
True	3	0	23	3	0	20
Random(∅)	6	0	40	12	0	37
EShopFIDE	26	0	23	5	0	19
False	26	0	40	4	0	19
True	2	0	21	2	0	25
Random(∅)	3	0	9	5	0	15
Automotive1	348	63	30,296	266	50	24,044
False	285	29	7,464	203	23	6,022
True	348	97	53,767	266	77	42,694
Random(∅)	291	63	29,659	217	49	23,416
Automotive2	1,757	83	731,869	1,266	71	634,596
False	1,690	135	1,787,553	1,166	118	1,572,406
True	1,739	57	198,422	1,121	43	152,352
Random(∅)	1,757	59	209,634	1,266	50	179,032
Splot1001	128	25	8,268	93	19	6,295
False	128	16	2,407	40	10	1,601
True	98	36	16,028	93	28	12,771
Random(∅)	99	25	6,369	81	17	4,514

Model	CA2 (in ms)			CA4 (in ms)		
	Max	∅	$\sum$	Max	∅	$\sum$
Splot1002	81	14	3,914	61	9	2,749
False	81	12	2,184	42	8	1,518
True	78	16	6,058	60	11	4,263
Random(∅)	74	13	3,502	61	9	2,466
Splot1003	89	23	6,781	67	16	4,854
False	73	20	2,586	60	14	1,867
True	85	26	11,976	67	18	8,545
Random(∅)	89	22	5,782	64	16	4,150
Splot1004	98	27	9,502	83	19	6,770
False	77	19	3,254	61	13	2,299
True	95	35	17,054	83	25	12,082
Random(∅)	98	26	8,200	83	19	5,931
Splot1005	99	20	5,011	81	14	3,563
False	68	14	2,049	47	10	1,454
True	95	25	7,530	73	17	5,377
Random(∅)	99	22	5,454	81	16	3,858
Splot1006	104	26	7,601	91	18	5,395
False	71	20	3,455	54	14	2,414
True	93	30	11,747	77	21	8,390
Random(∅)	104	28	7,602	91	20	5,382
Splot1007	110	27	6,657	82	20	4,788
False	110	24	4,011	68	17	2,844
True	90	30	8,776	66	21	6,263
Random(∅)	102	29	7,185	82	21	5,257
Splot1008	94	24	7,021	75	17	4,958
False	73	17	3,140	49	12	2,237
True	93	33	12,689	70	23	8,922
Random(∅)	94	21	5,235	75	15	3,716
Splot1009	104	25	7,928	105	18	5,645
False	91	18	3,064	55	13	2,165
True	104	32	13,454	83	23	9,585
Random(∅)	101	26	7,268	105	18	5,187
Splot1010	86	22	5,431	68	15	3,828
False	66	11	942	45	8	657
True	83	32	11,929	63	23	8,404
Random(∅)	86	21	3,422	68	15	2,424

Model	CA2 (in ms)			CA4 (in ms)		
	Max	∅	∑	Max	∅	∑
Splot2001	339	101	57,867	248	71	40,529
False	287	55	14,477	198	38	10,121
True	339	152	115,805	248	106	81,169
Random(∅)	329	97	43,319	240	67	30,297
Splot2002	353	83	40,198	251	58	28,178
False	275	61	16,165	208	43	11,433
True	343	109	72,176	251	76	50,514
Random(∅)	353	79	32,253	250	55	22,588
Splot2003	279	72	46,643	212	50	32,729
False	270	56	17,753	184	39	12,411
True	275	89	84,083	193	63	59,133
Random(∅)	279	71	38,095	212	50	26,645
Splot2004	300	91	52,670	252	63	36,697
False	277	78	21,723	208	55	15,187
True	299	109	94,621	225	75	65,753
Random(∅)	300	86	41,666	252	60	29,151
Splot2005	366	125	71,169	305	88	50,053
False	326	83	19,646	233	58	13,843
True	364	170	142,762	270	119	100,103
Random(∅)	366	121	51,101	305	85	36,214
Splot2006	302	78	37,623	237	55	26,855
False	242	60	17,138	163	43	12,127
True	302	98	63,608	237	70	45,662
Random(∅)	300	75	32,125	228	53	22,777
Splot2007	302	77	54,747	246	54	38,371
False	246	46	12,796	163	31	8,863
True	302	113	113,741	235	79	79,815
Random(∅)	299	72	37,704	246	50	26,436
Splot2008	317	90	61,766	240	63	43,655
False	270	66	24,900	187	45	17,287
True	316	126	119,435	234	89	85,066
Random(∅)	317	77	40,963	240	54	28,612
Splot2009	287	66	33,065	230	47	23,158
False	216	35	8,410	134	27	6,148
True	287	92	62,737	224	64	43,522
Random(∅)	279	71	28,048	230	50	19,804

Model	**CA2 (in ms)** Max	∅	∑	**CA4 (in ms)** Max	∅	∑
Splot2010	361	91	51,866	271	64	36,623
False	262	59	11,472	188	41	8,030
True	361	135	117,268	271	95	82,753
Random(∅)	360	79	26,858	255	56	19,087
Splot5001	1,290	332	313,330	1,103	262	246,664
False	1,180	297	147,177	988	235	116,550
True	1,233	391	547,627	997	308	430,625
Random(∅)	1,290	309	245,187	1,103	243	192,818
Splot5002	2,454	710	1,074,666	2,026	537	814,100
False	1,800	431	245,348	1,489	328	186,563
True	2,454	982	2,120,438	2,026	745	1,608,638
Random(∅)	2,446	716	858,214	1,991	539	647,101
Splot5003	2,321	669	691,215	1,908	501	517,897
False	2,289	568	285,628	1,892	424	213,340
True	1,945	798	1,201,833	1,497	597	900,116
Random(∅)	2,321	640	586,185	1,908	480	440,235
Splot5004	2,517	725	950,723	2,094	545	714,918
False	2,187	534	267,390	1,649	402	201,277
True	2,517	959	1,893,996	2,068	722	1,424,967
Random(∅)	2,501	681	690,785	2,094	511	518,512
Splot5005	2,805	762	1,020,249	2,230	578	774,688
False	2,710	545	271,851	2,074	411	204,694
True	2,791	961	1,982,078	2,171	731	1,508,840
Random(∅)	2,805	781	806,818	2,230	591	610,531
Splot5006	2,297	634	760,926	1,840	469	562,297
False	2,028	450	233,735	1,593	338	175,520
True	2,049	884	1,561,524	1,544	653	1,153,310
Random(∅)	2,297	569	487,521	1,840	417	358,061
Splot5007	2,571	875	1,192,851	2,050	649	885,225
False	2,445	710	411,133	1,858	525	304,305
True	2,554	1,151	2,366,466	2,037	855	1,759,193
Random(∅)	2,571	765	800,954	2,050	565	592,179
Splot5008	2,576	754	971,007	2,087	559	721,063
False	2,346	505	277,992	1,692	374	205,716
True	2,571	1,047	1,950,775	2,034	778	1,449,528
Random(∅)	2,576	710	684,254	2,087	527	507,947

Model	**CA2 (in ms)**			**CA4 (in ms)**		
	Max	∅	∑	Max	∅	∑
Splot5009	2,757	944	1,209,948	2,095	698	895,938
False	2,160	746	292,064	1,659	549	214,799
True	2,718	1,214	2,529,590	2,095	900	1,874,737
Random(∅)	2,757	871	808,192	2,051	645	598,279
Splot5010	2,816	760	1,039,521	2,169	563	772,361
False	1,843	556	261,960	1,346	410	193,437
True	2,816	974	2,084,668	2,169	726	1,553,118
Random(∅)	2,811	750	771,935	2,140	554	570,529
Splot10001	* 14,103	4,603	8,404,833	9,546	2,711	5,555,055
False	12,814	3,149	1,895,748	8,274	2,042	1,229,628
True	* 14,103	6,888	18,000,143	9,389	3,663	12,011,160
Random(∅)	14,010	3,773	5,318,608	9,546	2,429	3,424,379

Table A.4: Decision-propagation times for CA2 and CA4.

Model	#Visited Connections			#SAT Calls		
	none	strong	weak	CA1	CA2	CA4
Dell(∅)	22	19	152	69	74	75
False	9	1	252	114	116	119
True	30	30	50	25	28	29
Random(∅)	24	20	153	69	75	76
BerkeleyDB1(∅)	693	47	148	54	56	58
False	292	58	54	23	24	26
True	1,316	33	165	34	34	34
Random(∅)	656	47	161	63	65	68
BerkeleyDB2(∅)	239	50	1,053	330	362	375
False	236	67	281	145	145	148
True	244	42	1,602	451	514	501
Random(∅)	239	48	1,090	341	374	392
Violet(∅)	3,810	53	201	64	64	65
False	2,083	69	186	25	25	25
True	6,300	22	300	76	76	76
Random(∅)	3,683	55	187	68	69	71
EShopSplot(∅)	25,409	141	983	315	324	337
False	13,112	200	350	111	114	115
True	47,554	57	1,257	268	272	284
Random(∅)	23,767	146	1,043	356	368	383
EShopFIDE(∅)	28,612	176	1,043	292	300	308
False	14,873	232	327	111	111	111
True	52,112	90	1,396	256	270	280
Random(∅)	26,985	181	1,104	329	337	345
Automotive1(∅)	624,771	1,639	375,307	90,071	100,233	106,326
False	239,397	1,830	82,879	20,181	22,971	24,590
True	741,831	1,530	697,209	165,777	183,254	194,151
Random(∅)	669,490	1,625	370,395	89,102	99,273	105,312
Automotive2(∅)	84,869,670	10,887	238,882	105,157	114,770	115,557
False	195,930,049	1,908	952,273	434,987	460,217	464,113
True	71,265,392	12,248	152,842	66,061	76,260	77,689
Random(∅)	68,626,986	12,157	134,323	56,701	63,613	63,776
Splot1001(∅)	54,701	519	140,565	49,710	55,336	57,411
False	31,893	591	40,860	15,176	17,421	18,123
True	72,932	345	324,407	114,694	126,188	130,362
Random(∅)	55,465	537	126,543	44,635	49,847	51,800

Model	#Visited Connections			#SAT Calls		
	none	strong	weak	CA1	CA2	CA4
Splot1002(∅)	124,422	414	73,348	27,169	29,586	30,195
False	74,992	446	42,310	16,021	17,650	18,113
True	195,587	337	125,664	45,351	49,132	49,992
Random(∅)	120,799	422	69,802	25,997	28,318	28,910
Splot1003(∅)	123,081	619	128,976	44,967	49,018	51,411
False	66,900	716	54,898	19,015	20,656	21,708
True	187,261	415	247,454	87,053	95,406	99,907
Random(∅)	121,749	637	121,577	42,278	46,013	48,279
Splot1004(∅)	89,858	583	167,683	64,729	70,780	72,704
False	65,077	693	61,054	24,054	26,532	27,274
True	113,421	439	349,164	129,086	139,846	144,013
Random(∅)	90,061	589	155,208	60,783	66,644	68,391
Splot1005(∅)	64,466	522	105,346	37,998	41,429	42,871
False	44,643	592	38,083	14,547	15,850	16,380
True	63,115	425	146,998	53,349	58,687	61,219
Random(∅)	67,995	527	109,615	39,348	42,817	44,228
Splot1006(∅)	100,882	490	154,719	55,575	60,152	62,430
False	72,528	572	68,356	25,043	27,067	27,955
True	111,658	362	233,406	85,177	92,781	96,782
Random(∅)	103,812	498	155,999	55,730	60,228	62,450
Splot1007(∅)	78,038	587	138,165	49,531	54,564	57,200
False	51,925	672	81,149	28,414	31,142	32,716
True	101,131	393	180,764	62,639	68,728	71,530
Random(∅)	78,541	606	140,568	50,866	56,107	58,893
Splot1008(∅)	99,877	573	117,380	43,167	47,126	48,768
False	62,789	645	58,532	22,460	24,817	25,673
True	139,833	442	259,917	93,220	101,969	105,475
Random(∅)	99,399	582	103,432	38,277	41,704	43,166
Splot1009(∅)	74,808	562	151,253	55,264	60,159	62,849
False	45,618	654	57,404	21,959	23,921	24,828
True	94,948	387	272,553	98,748	108,137	113,101
Random(∅)	76,317	577	146,678	53,568	58,203	60,811
Splot1010(∅)	31,887	408	80,009	29,555	32,997	34,055
False	13,163	484	16,963	6,545	7,401	7,614
True	52,660	233	224,321	85,232	94,638	97,119
Random(∅)	31,545	425	66,465	24,111	26,989	27,952

Model	#Visited Connections none	strong	weak	#SAT Calls CA1	CA2	CA4
Splot2001(∅)	232,996	1,096	499,937	182,624	200,411	208,269
False	158,929	1,231	137,702	53,337	58,741	61,040
True	270,218	1,080	1,238,006	437,285	479,327	498,859
Random(∅)	239,137	1,077	437,299	161,729	177,537	184,376
Splot2002(∅)	250,829	1,031	340,179	131,175	145,212	150,895
False	180,287	1,133	150,002	59,739	66,190	68,692
True	308,395	639	715,336	270,316	299,196	310,608
Random(∅)	252,992	1,080	309,349	119,891	132,719	137,977
Splot2003(∅)	642,991	1,164	447,312	161,382	176,236	184,610
False	295,705	1,264	187,859	68,359	75,216	78,927
True	1,122,032	984	913,780	330,680	361,384	378,638
Random(∅)	621,032	1,178	412,810	148,670	162,214	169,886
Splot2004(∅)	456,972	1,313	465,367	178,250	195,031	203,078
False	265,365	1,521	223,150	84,612	92,518	96,633
True	737,088	934	949,858	370,306	406,376	421,769
Random(∅)	442,220	1,341	424,988	161,848	176,893	184,370
Splot2005(∅)	171,883	1,355	608,688	220,036	241,191	251,051
False	92,403	1,508	197,859	73,053	79,886	83,215
True	200,622	979	1,462,881	538,483	593,310	616,625
Random(∅)	180,340	1,392	534,794	191,459	209,388	218,094
Splot2006(∅)	294,217	1,205	349,616	129,403	141,432	149,169
False	221,230	1,264	173,004	64,563	70,600	74,507
True	354,859	1,258	651,561	241,635	264,175	278,594
Random(∅)	296,275	1,186	328,728	121,504	132,780	140,042
Splot2007(∅)	393,331	1,333	491,943	173,400	187,200	194,994
False	211,267	1,582	134,449	48,812	53,423	55,414
True	653,114	957	1,307,437	454,669	487,317	508,591
Random(∅)	380,378	1,354	415,610	147,287	159,477	165,991
Splot2008(∅)	445,663	1,170	514,602	184,742	203,150	211,426
False	284,494	1,221	246,119	92,811	103,437	107,538
True	695,718	1,009	1,314,711	458,395	498,007	518,829
Random(∅)	430,849	1,188	425,998	154,455	170,627	177,508
Splot2009(∅)	242,264	910	305,267	113,751	125,273	129,154
False	115,868	1,036	82,142	32,712	34,533	37,388
True	318,934	641	651,835	239,579	262,814	270,923
Random(∅)	250,552	934	284,693	106,287	117,473	120,820

Model	#Visited Connections			#SAT Calls		
	none	strong	weak	CA1	CA2	CA4
Splot2010(∅)	160,040	1,519	377,835	136,171	148,093	153,516
False	91,119	1,706	115,757	42,871	46,641	48,321
True	303,320	1,004	1,248,284	443,908	482,373	500,109
Random(∅)	147,647	1,574	276,440	100,432	109,289	113,283
Splot5001(∅)	1,431,471	2,034	1,171,537	441,230	481,584	502,027
False	819,696	2,424	650,038	245,254	267,448	279,188
True	2,444,359	1,597	2,397,960	900,714	984,937	1,025,611
Random(∅)	1,364,619	2,042	1,054,050	397,312	433,381	451,903
Splot5002(∅)	1,790,574	3,154	3,343,119	1,247,737	1,358,871	1,417,132
False	834,311	3,623	813,414	317,580	349,427	363,462
True	2,470,373	2,478	7,644,137	2,829,502	3,077,077	3,216,515
Random(∅)	1,836,651	3,189	3,047,900	1,139,136	1,240,744	1,292,847
Splot5003(∅)	1,277,021	3,344	2,209,961	797,120	874,675	909,141
False	694,026	3,669	1,001,705	364,896	400,702	415,684
True	1,626,052	2,712	4,274,598	1,528,712	1,675,153	1,745,120
Random(∅)	1,316,015	3,395	2,067,231	747,225	820,258	852,054
Splot5004(∅)	1,570,051	3,087	2,847,898	1,035,187	1,119,253	1,162,645
False	806,951	3,526	935,283	346,633	377,101	391,133
True	2,538,671	2,195	6,943,366	2,507,897	2,711,972	2,818,225
Random(∅)	1,535,798	3,163	2,484,090	904,494	977,492	1,015,300
Splot5005(∅)	1,857,081	3,284	3,106,282	1,106,153	1,185,298	1,240,155
False	788,519	3,651	939,674	338,571	363,938	380,475
True	2,918,327	2,463	6,902,176	2,458,997	2,634,566	2,760,024
Random(∅)	1,858,300	3,360	2,834,734	1,008,609	1,080,647	1,130,123
Splot5006(∅)	1,240,798	3,781	2,012,713	739,896	806,277	838,421
False	787,276	4,084	810,990	300,613	328,532	342,722
True	1,960,021	3,087	5,369,906	1,967,214	2,141,134	2,238,065
Random(∅)	1,196,514	3,847	1,653,468	608,557	663,425	687,763
Splot5007(∅)	1,549,766	3,001	3,246,208	1,199,714	1,305,034	1,364,473
False	902,797	3,281	1,387,154	520,975	569,105	594,273
True	2,684,666	2,706	8,371,342	3,031,598	3,281,935	3,432,498
Random(∅)	1,468,445	3,004	2,701,861	1,007,523	1,098,206	1,148,169
Splot5008(∅)	1,408,162	3,391	2,777,770	985,691	1,076,817	1,126,533
False	899,219	3,777	939,584	341,056	374,943	392,002
True	1,914,869	2,320	7,047,231	2,458,296	2,671,188	2,801,443
Random(∅)	1,408,535	3,506	2,372,558	847,697	928,067	969,803

Model	#Visited Connections			#SAT Calls		
	none	strong	weak	CA1	CA2	CA4
Splot5009(∅)	1,001,662	3,365	3,336,595	1,203,464	1,304,168	1,358,240
False	575,149	4,164	1,014,785	365,704	395,580	411,417
True	1,149,891	2,265	8,878,764	3,190,852	3,456,729	3,604,234
Random(∅)	1,048,043	3,415	2,799,868	1,011,860	1,096,840	1,141,711
Splot5010(∅)	1,803,787	3,655	3,059,741	1,103,057	1,184,043	1,232,366
False	692,081	4,187	896,116	329,246	355,006	370,321
True	3,127,394	2,674	7,368,566	2,633,023	2,825,036	2,940,415
Random(∅)	1,768,470	3,730	2,702,208	977,031	1,048,717	1,091,366
Splot10001(∅)	3,866,194	6,109	8,107,885	2,979,358	3,427,822	3,608,834
False	1,949,125	7,265	2,541,950	932,487	1,000,499	1,041,130
True	5,231,623	3,560	20,115,423	7,272,285	9,578,770	10,303,427
Random(∅)	3,958,134	6,342	7,034,285	2,605,016	2,807,218	2,921,020

Table A.5: Results of the dynamic analysis on all feature graphs.

Model	#Connections none	strong	weak	#Nodes	Size (in byte) (Compressed %)
Dell	2,590	374	2,812	76	5,042 (33.6)
BerkeleyDB1	13,325	1,500	3,671	136	9,409 (24.5)
BerkeleyDB2	24,028	858	20,910	214	23,215 (15.3)
Violet	33,332	1,566	3,518	196	16,939 (17.2)
EShopSplot	320,934	3,216	16,906	584	84,572 (9.0)
EShopFIDE	254,162	1,958	12,204	518	105,324 (8.7)
Automotive1	14,375,894	195,698	5,106,504	4,436	5,086,553 (0.7)
Automotive2	996,174,355	695,346	2,575,295	31,614	250,875,368 (0.1)
Splot1001	2,687,005	55,596	1,983,675	2,174	1,248,816 (1.6)
Splot1002	2,973,541	63,790	1,322,413	2,088	1,153,524 (1.7)
Splot1003	3,034,024	26,488	1,587,824	2,156	1,225,928 (1.7)
Splot1004	2,738,043	48,032	1,810,661	2,144	1,212,723 (1.6)
Splot1005	2,754,860	71,978	1,787,066	2,148	1,217,676 (1.7)
Splot1006	2,757,169	66,760	1,755,671	2,140	1,210,071 (1.8)
Splot1007	2,879,573	35,350	1,907,493	2,196	1,271,436 (1.7)
Splot1008	2,958,948	29,038	1,583,058	2,138	1,207,465 (1.8)
Splot1009	2,875,274	28,688	1,839,722	2,178	1,250,618 (1.6)
Splot1010	2,489,295	80,968	1,907,193	2,116	1,184,885 (1.7)
Splot2001	11,211,199	115,620	8,156,577	4,414	4,996,237 (0.9)
Splot2002	10,888,970	398,746	8,001,948	4,392	4,950,473 (0.8)
Splot2003	13,128,464	60,946	5,077,666	4,274	4,689,681 (0.9)
Splot2004	12,286,639	73,296	6,216,165	4,310	4,771,038 (0.8)
Splot2005	10,888,199	116,370	8,904,875	4,462	5,111,251 (0.8)
Splot2006	11,285,974	397,172	6,841,270	4,304	4,757,997 (0.8)
Splot2007	11,769,781	66,444	6,073,599	4,232	4,597,382 (0.8)
Splot2008	12,059,915	77,808	6,749,993	4,346	4,851,654 (0.8)
Splot2009	9,049,995	112,772	5,338,097	3,808	3,738,642 (1.0)
Splot2010	11,347,708	87,604	7,995,152	4,408	4,988,491 (0.8)
Splot5001	39,969,030	539,128	20,020,242	7,780	15,396,671 (0.5)
Splot5002	72,310,755	602,238	46,901,923	10,946	30,270,738 (0.3)
Splot5003	70,613,896	1,742,324	48,292,036	10,984	30,479,499 (0.3)
Splot5004	72,820,347	720,062	47,283,655	10,992	30,532,648 (0.3)
Splot5005	74,445,701	348,354	44,758,301	10,934	30,206,394 (0.3)
Splot5006	72,137,339	1,407,498	45,614,219	10,916	30,111,231 (0.3)
Splot5007	71,045,123	522,698	48,729,203	10,968	30,389,606 (0.3)
Splot5008	71,438,280	362,856	48,495,888	10,968	30,390,393 (0.3)
Splot5009	68,102,749	485,608	53,337,407	11,042	30,802,384 (0.3)
Splot5010	74,528,787	373,846	44,824,731	10,942	30,246,096 (0.3)
Splot10001	389,559,638	967,192	94,441,654	22,022	121,877,300 (0.2)

Table A.6: Results of the static analysis on all feature graphs.

Bibliography

[ABKS13] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer, Berlin, Heidelberg, 2013. (cited on Page 1 and 7)

[AFM02] Jérôme Amilhastre, Hélene Fargier, and Pierre Marquis. Consistency Restoration and Explanations in Dynamic CSPs - Application to Configuration. *Artificial Intelligence*, 135(1):199–234, 2002. (cited on Page 64)

[AKL13] 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, 2013. (cited on Page 7)

[APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas. *Information Processing Letters*, 8(3):121–123, 1979. (cited on Page 2 and 11)

[Bat05] Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20, Berlin, Heidelberg, 2005. Springer. (cited on Page 9 and 14)

[Bat06] Don Batory. A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite. In *Proceedings of the Summer School on Generative and Transformational Techniques in Software Engineering (GTTSE)*, pages 3–35, Berlin, Heidelberg, 2006. Springer. (cited on Page 7)

[BSRC10] David Benavides, Sergio Segura, and Antonio Ruiz-Cortés. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems*, 35(6):615–708, 2010. (cited on Page 13, 14, and 15)

[BSTRC07] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. FAMA: Tooling a Framework for the Automated Analysis of Feature Models. In *Proceedings of the Workshop on Variability Modelling of Software-*

intensive Systems (VaMoS), pages 129–134, Limerick, Ireland, 2007. Technical Report 2007-01, Lero. (cited on Page 64)

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *Proceedings of the International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 677–682, 2005. (cited on Page 63)

[CE00] Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM/Addison-Wesley, New York, NY, USA, 2000. (cited on Page 1, 5, 6, and 8)

[CHE05] Krzysztof Czarnecki, Simon Helsen, and Ulrich Eisenecker. Staged Configuration through Specialization and Multi-Level Configuration of Feature Models. *Software Process: Improvement and Practice*, 10(2):143–169, 2005. (cited on Page 1 and 12)

[CN01] Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, Boston, MA, USA, 2001. (cited on Page 1 and 5)

[Coo71] Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of The Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York, NY, USA, 1971. ACM. (cited on Page 1 and 13)

[CW07] Krzysztof Czarnecki and Andrzej Wąsowski. Feature Diagrams and Logics: There and Back Again. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 23–34, Washington, DC, USA, 2007. IEEE Computer Science. (cited on Page 11)

[Das05] Jürgen Dassow. *Logik für Informatiker*. Vieweg+Teubner Verlag, 2005. In German. (cited on Page 10)

[HJ90] Pierre Hansen and Brigitte Jaumard. Algorithms for the Maximum Satisfiability Problem. *Computing*, 44(4):279–303, 1990. (cited on Page 2)

[HSJ+04] Tarik Hadzic, Sathiamoorthy Subbarayan, Rune M. Jensen, Henrik R. Andersen, Jesper Møller, and Henrik Hulgaard. Fast Backtrack-Free Product Configuration Using a Precompiled Solution Space Representation. *Proceedings of the International Conference on Economic*, 10(1):131–138, 2004. (cited on Page 2 and 63)

[Käs10] Christian Kästner. *Virtual Separation of Concerns: Toward Preprocessors 2.0*. PhD thesis, University of Magdeburg, 2010. (cited on Page 7)

[KCH+90] 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 7 and 8)

[KLM+97] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 220–242, Berlin, Heidelberg, 1997. Springer. (cited on Page 7)

[LBP10] Daniel Le Berre and Anne Parrain. The Sat4j Library, Release 2.2. *Journal on Satisfiability, Boolean Modeling and Computation*, 7:59–64, 2010. (cited on Page 39)

[MBC09] Marcílio Mendonça, Moises Branco, and Donald Cowan. S.P.L.O.T.: Software Product Lines Online Tools. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 761–762, New York, NY, USA, 2009. ACM. (cited on Page 12, 46, 49, and 63)

[Men09] Marcílio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, Canada, 2009. (cited on Page 3, 12, and 64)

[MWC09] Marcílio Mendonça, 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, Pittsburgh, PA, USA, 2009. Software Engineering Institute. (cited on Page 2 and 13)

[MWCC08] Marcílio Mendonça, Andrzej Wąsowski, Krzysztof Czarnecki, and Donald 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, New York, NY, USA, 2008. ACM. (cited on Page 63)

[PBvdL05] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, Berlin, Heidelberg, 2005. (cited on Page 1, 5, 6, 7, and 11)

[Pre97] Christian Prehofer. Feature-Oriented Programming: A Fresh Look at Objects. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 419–443, Berlin, Heidelberg, 1997. Springer. (cited on Page 7)

[Seg08] Sergio Segura. Automated Analysis of Feature Models Using Atomic Sets. In *Workshop on Analyses of Software Product Lines (ASPL)*, pages 201–207. Lero Int. Science Centre, University of Limerick, Ireland, 2008. (cited on Page 15)

[STSS13] Reimar Schröter, Thomas Thüm, Norbert Siegmund, and Gunter Saake. Automated Analysis of Dependent Feature Models. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 9:1–9:5, New York, NY, USA, 2013. ACM. (cited on Page 14)

[TBC06] Pablo Trinidad, David Benavides, and Antonio Ruiz Cortés. Isolated Features Detection in Feature Models. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, Aachen, Germany, 2006. CEUR-WS.org. (cited on Page 14)

[TGH97] Paul Tafertshofer, Andreas Ganz, and Manfred Henftling. A SAT-Based Implication Engine for Efficient ATPG, Equivalence Checking, and Optimization of Netlists. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD)*, pages 648–655, Washington, DC, USA, 1997. IEEE Computer Science. (cited on Page 11)

[TKB+14] 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 (SCP)*, 79(0):70–85, 2014. (cited on Page 6 and 12)

[TLD+11] Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *Operating Systems Review*, 45(3):10–14, 2011. (cited on Page 2)

[TRC09] Pablo Trinidad and Antonio Ruiz-Cortés. Abductive Reasoning and Automated Analysis of Feature Models: How are They Connected? In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 145–153, Essen, Germany, 2009. Universität Duisburg-Essen. (cited on Page 14)

[WSB+08] Jules White, Douglas C. Schmidt, David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Automated Diagnosis of Product-Line Configuration Errors in Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 225–234. IEEE Computer Science, 2008. (cited on Page 2 and 64)

[ZZM04] Wei Zhang, Haiyan Zhao, and Hong Mei. A Propositional Logic-Based Method for Verification of Feature Models. *Formal Methods and Software Engineering*, pages 115–130, 2004. (cited on Page 15)

I hereby declare that I have written this thesis without any help from others and without the use of documents and aids other than those stated above. I have mentioned all used sources and cited them correctly according to established academic citation rules.

Magdeburg, 19.10.2015

Sebastian Krieter