OTTO VON GUERICKE UNIVERSITÄT MAGDEBURG

Efficient Interactive and Automated Product-Line Configuration

DISSERTATION

zur Erlangung des akademischen Grades

Doktoringenieur (Dr.-Ing.)

angenommen durch die Fakultät für Informatik
der Otto-von-Guericke-Universität Magdeburg

von M.Sc. Sebastian Krieter

geb. am 18.10.1990 in Magdeburg

Gutachterinnen/Gutachter

Prof. Dr. rer. nat. habil. Gunter Saake
Prof. Dr.-Ing. Thomas Leich
Univ.-Prof. Mag. Dr. Rick Rabiser

Magdeburg, den 30.06.2022

Otto-von-Guericke University Magdeburg

Faculty of Computer Science

Dissertation

Efficient Interactive and Automated Product-Line Configuration

Author:

Sebastian Krieter

22.03.2022

Reviewers:

Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke University, Magdeburg, Germany

Prof. Dr.-Ing. Thomas Leich
Harz University of Applied Sciences, Wernigerode, Germany

Univ.-Prof. Mag. Dr. Rick Rabiser
Johannes Kepler University, Linz, Austria

Krieter, Sebastian:
Efficient Interactive and Automated Product-Line Configuration
Dissertation, Otto-von-Guericke University Magdeburg, 2022.

Contents

Abstract

Modern, highly-configurable systems comprise an enormous number of variants that are created from a common code base and tailored to specific user requirements. Managing this amount of variability poses challenges for users and developers of these systems alike. Users must correctly configure a system, keeping in mind all of its dependencies, in order to get their desired variant. Developers must ensure that every correctly configured variant does not contain any faults and behaves according to its specifications. Both of these activities are challenging, because testing or even considering all possible combinations of configuration options is unfeasible for most systems. In this thesis, we address these issues by investigating tool assistance for fully-automated and semi-automated configuration processes of configurable systems. In particular, we introduce *modal implication graphs (MIGs)*, a type of a knowledge compilation technique to speed up the configuration process by making information about dependencies between configuration options easily accessible. We find that MIGs can help to speed up the configuration process for users and automated techniques and also increases performance of other common variability analyses. Further, we introduce a new *sampling algorithm* for automatically generating variants *for t-wise interaction coverage*, which can be adapted to specific properties of a system. We extend our sampling algorithm to enable our new concept of *t-wise presence condition coverage*, which aims to facilitate product-based testing by taking presence conditions of implementation artifacts into account. With t-wise presence condition coverage and our corresponding sampling algorithm, we are able to compute sets of variants (i.e., samples) that are more effective for testing than samples from state-of-the-art algorithms. Furthermore, we are able to produce samples faster and with fewer variants, which decreases the overall testing effort for a system. In summary, we facilitate the fully-automated and semi-automated configuration process with a focus on improving the effectiveness and efficiency of testing and analyzing configurable systems.

Zusammenfassung

Moderne, hochgradig konfigurierbare Systeme umfassen eine enorme Anzahl von Varianten, die aus einer gemeinsamen Codebasis erstellt und auf spezifische Benutzeranforderungen zugeschnitten werden. Die Verwaltung und Benutzung dieser hohen Variabilität stellt sowohl die Benutzer als auch die Entwickler dieser Systeme vor Herausforderungen. Die Benutzer dieser Systeme müssen in der Lage sein unter Berücksichtigung aller Abhängigkeiten eine korrekte Konfiguration zu erstellen, um ihre gewünschte Variante zu erhalten. Die Entwickler müssen hingegen sicherstellen, dass jede korrekt konfigurierte Variante keine Fehler enthält und sich entsprechend ihrer Spezifikationen verhält. Beides ist eine Herausforderung, da es für die meisten Systeme nicht möglich ist, alle möglichen Kombinationen von Konfigurationsoptionen zu testen oder auch nur zu berücksichtigen. In dieser Arbeit betrachten wir diese Probleme, indem wir Werkzeugunterstützung für automatische und halbautomatische Konfigurationsprozesse von konfigurierbaren Systemen untersuchen. Konkret führen wir *modale Implikationsgraphen (MIGs)* ein, eine Art von Wissensaufbereitung, die den Konfigurationsprozess beschleunigen, indem sie Informationen über Abhängigkeiten zwischen Konfigurationsoptionen leicht zugänglich machen. Wir können beobachten, dass MIGs dazu beitragen, den Konfigurationsprozess für Benutzer und automatisierte Techniken zu beschleunigen und auch die Leistung anderer gängiger Variabilitätsanalysen zu erhöhen. Darüber hinaus stellen wir einen neuen *Sampling-Algorithmus* zur automatischen Generierung von Varianten für *t-wise interaction coverage* vor, der an die spezifischen Eigenschaften eines Systems angepasst werden kann. Wir erweitern unseren Sampling-Algorithmus, um unser neues Konzept der *t-wise presence condition coverage* umsetzen zu können, das darauf abzielt, produktbasiertes Testen durch die Berücksichtigung von *Presence Conditions* für Implementierungsartefakte zu verbessern. Mit *t-wise presence condition coverage* und unserem entsprechenden Sampling-Algorithmus können wir Mengen von Varianten (d.h. Stichproben) berechnen, die ein effektiveres Testen ermöglichen als Stichproben von anderen State-of-the-Art-Algorithmen. Darüber hinaus sind wir in der Lage, Stichproben schneller und mit weniger Varianten zu erstellen, was den Gesamttestaufwand für ein System verringert. Zusammenfassend lässt sich sagen, dass wir den automatischen und halbautomatischen Konfigurationsprozess erleichtern und dabei

den Schwerpunkt auf die Verbesserung der Effektivität und Effizienz des Testens und der Analyse konfigurierbarer Systeme legen.

List of Publications

In the following, we list the publications contributing to this thesis. We differentiate between *main publications* on which the main contributions of this thesis are based and *other publications* that only partly contribute to this thesis. The thesis share material with the main publication, which we also state this at the beginning of the corresponding chapters. Furthermore, we list *additional publications* that developed in parallel to our core research.

Main Publications Contributing to the Thesis

1. **Sebastian Krieter**, Thomas Thüm, Sandro Schulze, Sebastian Ruland, Malte Lochau, Gunter Saake, Thomas Leich: *T-Wise Presence Condition Coverage and Sampling for Configurable Systems.* arXiv:2205.15180 [cs.SE] https://arxiv.org/abs/2205.15180, 2022

2. **Sebastian Krieter**, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, Christoph Seidl: *Incremental construction of modal implication graphs for evolving feature models.* SPLC (A) 2021: 64-74

3. **Sebastian Krieter**: *Large-scale T-wise interaction sampling using YASA.* SPLC (A) 2020: 29:1-29:4

4. **Sebastian Krieter**, Thomas Thüm, Sandro Schulze, Gunter Saake, Thomas Leich: *YASA: yet another sampling algorithm.* VaMoS 2020: 4:1-4:10

5. **Sebastian Krieter**: *Enabling efficient automated configuration generation and management.* SPLC (B) 2019: 93:1-93:7

6. **Sebastian Krieter**, Thomas Thüm, Sandro Schulze, Reimar Schröter, Gunter Saake: *Propagating configuration decisions with modal implication graphs.* ICSE 2018: 898-909

Other Publications Partly Contributing to the Thesis

1. Tobias Pett, Thomas Thüm, Tobias Runge, **Sebastian Krieter**, Malte Lochau, Ina Schaefer: *Product sampling for product lines: the scalability challenge.* SPLC (A) 2019: 14:1-14:6

2. Thomas Thüm, **Sebastian Krieter**, Ina Schaefer: *Product Configuration in the Wild: Strategies for Conflicting Decisions in Web Configurators.* ConfWS 2018: 1-8

3. Thomas Thüm, Thomas Leich, **Sebastian Krieter**: *Feature Modeling and Development with FeatureIDE.* Modellierung 2018: 297-298

4. Elias Kuiter, **Sebastian Krieter**, Jacob Krüger, Kai Ludwig, Thomas Leich, Gunter Saake: *PClocator: a tool suite to automatically identify configurations for code locations.* SPLC 2018: 284-288

5. Mustafa Al-Hajjaji, **Sebastian Krieter**, Thomas Thüm, Malte Lochau, Gunter Saake: *IncLing: efficient product-line testing using incremental pairwise sampling.* GPCE 2016: 144-155

6. Mustafa Al-Hajjaji, Jens Meinicke, **Sebastian Krieter**, Reimar Schröter, Thomas Thüm, Thomas Leich, Gunter Saake: *Tool demo: testing configurable systems with FeatureIDE.* GPCE 2016: 173-177

7. Juliana Alves Pereira, **Sebastian Krieter**, Jens Meinicke, Reimar Schröter, Gunter Saake, Thomas Leich: *FeatureIDE: Scalable Product Configuration of Variable Systems.* ICSR 2016: 397-401

8. **Sebastian Krieter**, Reimar Schröter, Thomas Thüm, Wolfram Fenske, Gunter Saake: *Comparing algorithms for efficient feature-model slicing.* SPLC 2016: 60-64

Additional Publications

1. Elias Kuiter, **Sebastian Krieter**, Jacob Krüger, Gunter Saake, Thomas Leich: *variED: an editor for collaborative, real-time feature modeling.* Empir. Softw. Eng. 26(2): 24 (2021)

2. Tobias Pett, **Sebastian Krieter**, Thomas Thüm, Malte Lochau, Ina Schaefer: *AutoSMP: an evaluation platform for sampling algorithms.* SPLC (B) 2021: 41-44

3. Chico Sundermann, Tobias Heß, Dominik Engelhardt, Rahel Arens, Johannes Herschel, Kevin Jedelhauser, Benedikt Jutz, **Sebastian Krieter**, Ina Schaefer: *Integration of UVL in FeatureIDE.* SPLC (B) 2021: 73-79

4. Tobias Pett, **Sebastian Krieter**, Tobias Runge, Thomas Thüm, Malte Lochau, Ina Schaefer: *Stability of Product-Line Sampling in Continuous Integration.* VaMoS 2021: 18:1-18:9

5. Sofia Ananieva, Sandra Greiner, Thomas Kühn, Jacob Krüger, Lukas Linsbauer, Sten Grüner, Timo Kehrer, Heiko Klare, Anne Koziolek, Henrik Lönn, **Sebastian Krieter**, Christoph Seidl, S. Ramesh, Ralf H. Reussner, Bernhard Westfechtel: *A conceptual model for unifying variability in space and time.* SPLC (A) 2020: 15:1-15:12

6. Jacob Krüger, **Sebastian Krieter**, Gunter Saake, Thomas Leich: *EXtracting product lines from vAriaNTs (EXPLANT).* VaMoS 2020: 13:1-13:2

7. Joshua Sprey, Chico Sundermann, **Sebastian Krieter**, Michael Nieke, Jacopo Mauro, Thomas Thüm, Ina Schaefer: *SMT-based variability analyses in FeatureIDE.* VaMoS 2020: 6:1-6:9

8. Alexander Knüppel, Stefan Krüger, Thomas Thüm, Richard Bubel, **Sebastian Krieter**, Eric Bodden, Ina Schaefer: *Using Abstract Contracts for Verifying Evolving Features and Their Interactions.* 20 Years of KeY 2020: 122-148

9. Elias Kuiter, **Sebastian Krieter**, Jacob Krüger, Thomas Leich, Gunter Saake: *Foundations of collaborative, real-time feature modeling.* SPLC (A) 2019: 36:1-36:8

10. **Sebastian Krieter**, Tobias Thiem, Thomas Leich: *Using Dynamic Software Product Lines to Implement Adaptive SGX-enabled Systems.* VaMoS 2019: 9:1-9:9

11. Juliana Alves Pereira, Pawel Matuszyk, **Sebastian Krieter**, Myra Spiliopoulou, Gunter Saake: *Personalized recommender systems for product-line configuration processes.* Comput. Lang. Syst. Struct. 54: 451-471 (2018)

12. Vasily A. Sartakov, Nico Weichbrodt, **Sebastian Krieter**, Thomas Leich, Rüdiger Kapitza: *STANlite - A Database Engine for Secure Data Processing at Rack-Scale Level.* IC2E 2018: 23-33

13. **Sebastian Krieter**, Jacob Krüger, Nico Weichbrodt, Vasily A. Sartakov, Rüdiger Kapitza, Thomas Leich: *Towards secure dynamic product lines in the cloud.* ICSE (NIER) 2018: 5-8

14. Juliana Alves Pereira, Jabier Martinez, Hari Kumar Gurudu, **Sebastian Krieter**, Gunter Saake: *Visual guidance for product line configuration using recommendations and non-functional properties.* SAC 2018: 2058-2065

15. Elias Kuiter, Jacob Krüger, **Sebastian Krieter**, Thomas Leich, Gunter Saake: *Getting rid of clone-and-own: moving to a software product line for temperature monitoring.* SPLC 2018: 179-189

16. Thomas Thüm, **Sebastian Krieter**, Thomas Leich: *Clean your variable code with FeatureIDE.* SPLC 2018: 299

17. **Sebastian Krieter**, Jacob Krüger, Thomas Leich: *Don't Worry About it: Managing Variability On-The-Fly.* VaMoS 2018: 19-26

18. Juliana Alves Pereira, Sandro Schulze, **Sebastian Krieter**, Márcio Ribeiro, Gunter Saake: *A Context-Aware Recommender System for Extended Software Product Line Configurations.* VaMoS 2018: 97-104

19. Reimar Schröter, **Sebastian Krieter**, Thomas Thüm, Fabian Benduhn, Gunter Saake: *Compositional Analyses of Highly-Configurable Systems with Feature-Model Interfaces.* Software Engineering 2017: 129-130

20. **Sebastian Krieter**, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, Gunter Saake: *FeatureIDE: Empowering Third-Party Developers.* SPLC (B) 2017: 42-45

21. Jacob Krüger, Sebastian Nielebock, **Sebastian Krieter**, Christian Diedrich, Thomas Leich, Gunter Saake, Sebastian Zug, Frank Ortmeier: *Beyond Software Product Lines: Variability Modeling in Cyber-Physical Systems.* SPLC (A) 2017: 237-241

22. Juliana Alves Pereira, Pawel Matuszyk, **Sebastian Krieter**, Myra Spiliopoulou, Gunter Saake: *A feature-based personalized recommender system for product-line configuration.* GPCE 2016: 120-131

23. Jens Meinicke, Thomas Thüm, Reimar Schröter, **Sebastian Krieter**, Fabian Benduhn, Gunter Saake, Thomas Leich: *FeatureIDE: taming the preprocessor wilderness.* ICSE (Companion Volume) 2016: 629-632

24. Reimar Schröter, **Sebastian Krieter**, Thomas Thüm, Fabian Benduhn, Gunter Saake: *Feature-model interfaces: the highway to compositional analyses of highly-configurable systems.* ICSE 2016: 667-678

25. Thomas Thüm, Thomas Leich, **Sebastian Krieter**: *Clean your variable code with FeatureIDE.* SPLC 2016: 308

1. Introduction

Configurable systems, such as software product lines, allow users to adapt the behavior of a software system to their particular requirements. To this end, a configurable system comprises a common code base with multiple configuration options that are mapped to corresponding implementation artifacts [Apel et al., 2013; Czarnecki and Eisenecker, 2000]. A user can provide a configuration that defines the state of each configuration option in order to create their desired variant of the system [Apel et al., 2013; Czarnecki and Eisenecker, 2000]. Modern configurable systems can offer thousands of configuration options, which results in an enormous number of different possible variants [Berger et al., 2010, 2013a; Hubaux et al., 2012; Sundermann et al., 2020]. Typically, configuration options have complex interdependencies stemming from their relationships defined in a variability model and interactions of their respective implementation artifacts [Knüppel et al., 2017; Nadi et al., 2015].

The high amount of variability and complex interdependencies pose challenges for the developers of modern configurable systems, as well as for their users [Batory et al., 2006; Ye and Liu, 2005]. Users of a configurable system have to perform a configuration process to derive a variant suitable fo their needs. During this process they must keep in mind all relevant dependencies between their chosen configuration options in order to derive a working system variant. For instance, some options may be mutually exclusive or require a specific setting of another option. This can make the configuration process very tedious and error-prone, as some combinations of configuration options can be impossible or introduce undesired effects [Batory et al., 2006]. For developers this issue is even more complex, because they have to make sure that every configurable variant behaves according to its specifications and does not contain any unforeseen interactions or faults. Simply inspecting every possible variant is unfeasible due to the enormous number of configurations of a system. Thus, developers must reason about all possible interactions by other means. Solving these issues for modern configurable systems requires automated analyses and tool support during the development and configuration process. In this thesis, we take a look at tool support specifically for the configuration process and try to improve its efficiency and effectiveness in creating configurations for users and developers.

The configuration process is one of the fundamental tasks for developers and users of configurable systems. In order to generate a working variant a configuration needs to be valid (i.e., define a non-conflicting combination of configuration options) [Apel et al., 2013; Czarnecki and Eisenecker, 2000]. There exist many different possibilities to facilitate this process by means of automation. Tool support can be used to enable a semi-automated configuration process to assist users during their decision making. For instance, an algorithm can recommend to set certain configuration options [Pereira et al., 2018], set configuration options that are implied by previous decisions (i.e., decision propagation) [Batory, 2005], auto-complete a partial configuration, or repair errors that cause a configuration to be invalid [Hubaux et al., 2012]. These techniques mitigate the risk of creating invalid configurations and decrease the manual effort for users, while still providing them sufficient control over their created configurations. A semi-automated configuration process also supports developers in analyzing the variability of a system, testing variants, and optimizing parameters.

Tool support can also enable a fully-automated configuration process, in which, configurations are generated entirely by an algorithm based on certain input specifications. For instance, an algorithm can generate random configurations (e.g., for testing purposes), configurations that satisfy a certain coverage criterion (e.g., t-wise interaction coverage), or configurations that are similar to previously created configurations (e.g., for parameter optimization) [Oh et al., 2017]. While a fully-automated configuration process requires no manual effort for creating a configuration, users have only limited control over the created configurations, as these entirely depend on the chosen algorithm and input data. Thus, a fully-automatic process is often more useful for testing, parameter optimization, and evaluation of a system, rather than creating a single user configuration. In our thesis, we identify and address current short comings within the tool support for both, the semi- and fully-automated configuration process, such as increasing the scalability and effectiveness of certain techniques. In particular, we consider *facilitating decision propagation*, as it is one of the most helpful techniques for the semi-automatic configuration process and can also be used in automated analyses and algorithms for a fully-automatic configuration process. Regarding the fully-automated configuration process, we consider *improving sampling algorithms* with a focus on creating configurations for testing purposes.

In an interactive configuration process it is often critical for users to know the consequences of their decisions immediately, in order to avoid unwanted effects later on. For example, a configurable system may offer a configuration option for compatibility with an operating system and a configuration option for compatibility with a file system. These options can have interdependencies, for instance the file system NTFS only works for Windows and the file system EXT only works for Linux. If a user is unaware of these dependencies and decides to use EXT as file system and Windows as operating system, the resulting variant will be faulty. Considering that real-world systems can contain thousands of interdependent configuration options, manually finding such contradictions within a configuration is infeasible. Decision propagation determines all configuration options that must be set, because they are implied by previous user decisions, and thus informs users about all consequences of their decisions at any point during the configuration process [Batory, 2005; Hubaux et al., 2012; Janota, 2008; Mendonça, 2009]. This effectively prevents users from

making contradictory decisions and reduces the amount of decisions a user has to make. When employing decision propagation in our example, the user would be unable to configure a incompatible file and operating systems, because as soon as they select EXT, Linux would be selected as well.

While decision propagation provides helpful assistance in an interactive configuration process, it is also a computationally expensive algorithm, because the underlying problem of finding valid assignments for interdependent Boolean variables (i.e., the Boolean satisfiability problem) is NP-complete [Cook, 1971]. The open-source tool FeatureIDE uses decision propagation in its configuration tool [Meinicke et al., 2017]. In practice, we experienced no performance issues when configuring small and medium-sized variability models with FeatureIDE. However, when applied to industrial systems with more than 10,000 configuration options, propagation of a single decision took us over 20 seconds on average on modern consumer hardware. The total time spend on decision propagation for creating a single configuration for this system summed up to 13 hours, which does not include the time required for a user to reason about decisions and to interact with the configuration tool. Thus, decision propagation can be a bottleneck within an automated configuration process, such as sampling [Al-Hajjaji et al., 2016a; Johansen et al., 2012a]. For this reason, we aim to improve the performance of decision propagation using novel data structures.

A fully-automated configuration process by means of sampling is a valuable technique for testing configurable systems. Testing itself is an important task within the development process in order to detect faults and ensure correct behavior [Ammann and Offutt, 2016; McGregor, 2010]. However, exhaustive testing of configurable systems is often impossible, because of their high number of possible variants [Engström and Runeson, 2011; Lee et al., 2012; Sundermann et al., 2020]. One testing method that addresses this issues is to generate a small but representative list of configurations (i.e., sampling) and execute the test cases of a system for each variant from the sample [Thüm et al., 2014a; Varshosaz et al., 2018]. There exist a number of different sampling strategies for generating representative samples [Varshosaz et al., 2018]. One promising sampling strategy is t-wise interaction sampling, which aims to generate a minimal sample that covers all possible interactions of t configuration options (e.g., all selected, none selected, only one selected, etc.) [Cohen et al., 2008; Marijan et al., 2013; Oster et al., 2010; Perrouin et al., 2010]. Even for small values of t (i.e., $t \in \{2, 3\}$) t-wise interaction sampling produces effective samples for testing [Abal et al., 2018; Kästner et al., 2009; Marijan et al., 2013]. However, regarding highly-configurable systems with thousands of features, even when using small values of t and a state-of-the-art sampling algorithm, sampling can take an infeasible amount of time (i.e., sampling time) and produce samples containing a high amount of configurations (i.e., sample size) [Pett et al., 2019]. Therefore, we look into increasing the efficiency of t-wise interaction sampling algorithms by improving its general performance and introducing additional parameters to fine-tune a sampling process for a given configurable system.

Another issue when using t-wise interaction sampling is that it works purely on the dependencies defined by the variability model of a configurable system. This means, that t-wise interaction sampling does not take into account the implementation artifacts of a system, such as source code, models, and test cases. This can lead to

unnecessary large samples and high sampling time. Furthermore, we suspect that the resulting samples may not accurately represent actual interactions of implementation artifacts, and thus are less effective, when used for testing. Within our thesis, we aim to address this issues by introducing a new coverage criterion for a t-wise coverage algorithm to create a more efficient and effective sample.

1.1 Goals and Contributions

Our main goal of the thesis is to improve the tool support for the configuration process of configurable systems. To this end, we introduce three new techniques to facilitate the semi-automated and fully-automated configuration process. First, we introduce *modal implication graphs (MIGs)*, a new data structure based on implication graphs. With MIGs, we make information about dependencies between configuration options more easily accessible, which we utilize to speed up the decision propagation, sampling techniques, and other variability analyses. Second, we introduce a new *sampling algorithm* for efficiently generating samples for *for t-wise interaction coverage*. Our new sampling algorithm *YASA* aims to reduce sampling time and sample size compared to current state-of-the-art algorithms by using MIGs and other optimizations. Further, we allow the adaption of YASA by means of runtime parameters and a modular architecture. This results in users having more control over sampling time and sample size, as well as over other properties of a sample, such as similarities between configurations within a sample, and thus are able to fine-tune samples for a given system. Third, we present the coverage criterion *t-wise presence condition coverage*, which we use to create better samples for testing configurable system. Rather than counting only interactions between configuration options, our new criterion considers interactions of actual implementation artifacts. We extend our sampling algorithm YASA for t-wise presence condition coverage, such that we can generate samples that are more effective for testing than samples from state-of-the-art algorithms that use t-wise interaction coverage. Furthermore, we aim to produce samples faster and with fewer variants, which decreases the overall testing effort for a system.

With our contributions and their evaluation in this thesis, we aim to answer multiple research questions. First, we want to know, can we increase the efficiency of configuration generation in an interactive and automated configuration process using MIGs? Second, can we control the initial build costs and effectiveness of a MIG by modifying its build process? Third, can we increase the efficiency of configuration generation in an automated configuration process using YASA? Fourth, can we control the sample size and sampling time of YASA with its parameter settings? Fifth, can we increase the testing efficiency or testing effectiveness by employing t-wise presence condition coverage?

In summary, we contribute the following:

- We introduce modal implication graphs (MIGs) to represent and efficiently access dependencies between configuration options.
  - We propose an algorithm to propagate decisions utilizing MIGs.

  - We present a build process for creating MIGs and an incremental build process for updating MIGs after evolution of a configurable system.
- We propose YASA, a configurable sampling algorithm for t-wise interaction sampling.
- We introduce the coverage criterion t-wise presence condition coverage for creating more effective samples for testing.
  - We extend YASA to support t-wise presence condition sampling.
- We evaluate all our contributions using multiple real-word configurable system in different versions.
  - We provide our developed algorithms as part of the open-source framework FeatureIDE[1].
  - We publish the data from our experiments[2].

1.2 Structure

We structure the remainder of our thesis as follows. In Chapter 2, we explain the foundations necessary to comprehend our following concepts and contributions. In Chapter 3, we introduce modal implication graphs (MIGs), including a regular and incremental build process to create MIGs and the application of MIGs in decision propagation in a semi-automated configuration process. In Chapter 4, we present our flexible and efficient sampling algorithm YASA for t-wise interaction sampling. In Chapter 5, we propose our new coverage criterion t-wise presence condition coverage and a corresponding algorithm for t-wise presence condition sampling, which is an extension of YASA. In Chapter 6, we present work related to our concepts. In Chapter 7, we conclude our findings and provide an outlook on potential future research.

[1] https://featureide.github.io/
[2] https://github.com/skrieter/evaluation-phd

2. Background

In this chapter, we present all foundations on the *problem* and *solution space* of configurable systems that are necessary to understand the concepts and technical details in the remainder of our thesis. The problem space describes all valid configurations of a configurable system, while the solution space describes all actual variants that can be build from that system [Czarnecki and Eisenecker, 2000]. In our thesis, we are concerned with improving variability analyses and the configuration process, which are concerned with the configuration space, and with improving product testing, which is mainly focused on the solution space.

In the following, we define the fundamentals and our terminology of problem and solutions space and give more details on relevant concepts and techniques that we use in the remainder of our thesis. First, we present fundamentals of the problem space with a special focus on variability modeling, configurations, and feature model analyses (see Section 2.1). Second, we present fundamentals of the solution space and describe our notion of presence conditions for implementation artifacts (see Section 2.2). Third, we give more details on the configuration process (see Section 2.3), which lays the foundation for our contributions in the following Chapters 3–5.

2.1 Problem Space

A configurable system uses configuration options to enable the creation of different variants of itself. All possible settings of these configuration options make up the *problem space* of a configurable system [Czarnecki and Eisenecker, 2000]. The process of determining the configuration options and all valid combinations of their settings (i.e., configurations) is called *variability modeling* [Clements and Northrop, 2001]. The result of this process is a *variability model*, which then defines the problem space of a configurable system [Apel et al., 2013; Clements and Northrop, 2001; Pohl et al., 2005].

Variability modeling can be done in a number of forms either implicitly through the internal architecture of a system or explicitly with a concrete variability modeling technique. For explicit variability modeling, there are multiple different technique, such

as feature models [Apel et al., 2013; Kang et al., 1990], decision models [Dhungana et al., 2007], orthogonal variability models [Pohl et al., 2005], textual- [Sundermann et al., 2021] and UML-based variability models [Gomaa, 2004], and other open-source or proprietary approaches. In this thesis, we focus on variability modeling with *feature models*, because they are a popular and commonly used technique in academia and industry [Apel et al., 2013; Kang et al., 1990]. Note that, for many of the above mentioned techniques there exists methods for transforming any of them into any other [Feichtinger et al., 2021]. Thus, if there is an operation or analysis based on one particular variability modeling technique it is likely that the same or a similar analysis or operation can be used on a different technique as well. In the following, we first, go into detail on feature models and their representations, as well as the concrete formalism we use throughout our thesis. Second, we present details of configurations for feature models and how we formalize them in our thesis. Finally, we describe existing analysis for feature modeling that are used to detect anomalies within a given feature model.

2.1.1 Feature Model

A *feature model* is a variability modeling approach that defines a list of *features* and a list of *constraints* to describe a problem space [Apel et al., 2013; Kang et al., 1990]. Features correspond to configuration options and constraints describe dependencies between features, which impose restrictions on the problem space.

Features

The term *feature* has many different, but similar definitions throughout the literature [Apel and Kästner, 2009]. We use the definition of Kang et al. and define a feature as a "prominent or distinctive user-visible aspect, quality, or characteristic of a system" [Kang et al., 1990]. We have to differentiate the meaning of a feature within the problem space and the solution space. In the problem space, a feature can be seen as a variable that can be assigned a value (i.e., configured) by a user. In the solution space, a feature can be mapped to one or more concrete implementation artifacts, which, when properly combined, can form a variant. Most commonly, features are of a Boolean nature, meaning that they can either be assigned the value *true* (i.e., selected) or *false* (i.e., deselected) in a configuration [Czarnecki and Eisenecker, 2000]. However, in general, features may also allow for ternary (e.g., kconfig [Community, 2018]) or even n-ary values [She et al., 2010]. It is also possible to have an infinite range of values for a feature, such as a real number [She et al., 2010]. Nevertheless, in this thesis, we focus on Boolean features as these are by far the most commonly used type of features. Note that, for features with a finite range of values it is always possible to construct a feature model containing only Boolean features that resembles the same problem space. Thus, almost all techniques we present in this thesis may be applied to feature models with non-Boolean features as well.

In Figure 2.1, we show the feature model of our running example *Server* system. The feature model is represented as a *feature diagram* and defines the ten features *Server*, *File System (FS)*, *Operating System (OS)*, *Logging (Log)*, *NTFS*, *HFS*, *EXT*, *Windows (Win)*, *MacOS (Mac)*, and *Debian (Deb)*. All feature in this feature model

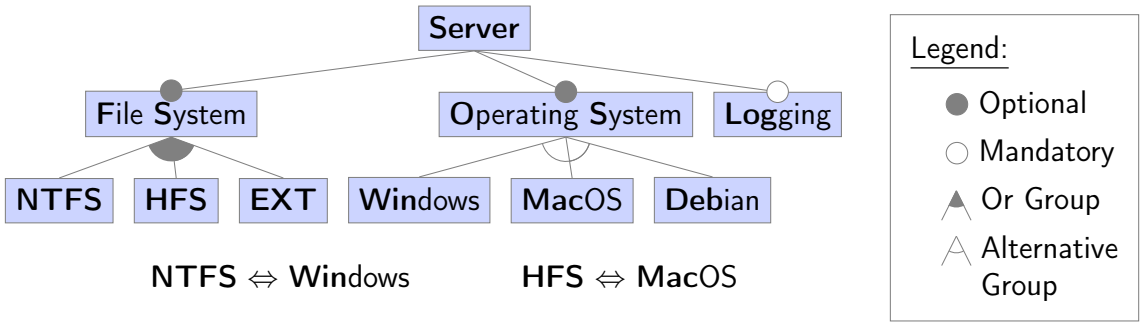

Figure 2.1: Feature diagram of the example system *Server*

are Boolean, and thus can be set to either *true* or *false*. In the remainder of the thesis, we use the provided abbreviations of the feature names to ease the readability of all propositional formulas using these features.

Constraint

A constraint describes the relationship between features and thereby effectively limits the valid problem space (i.e., excludes particular configurations of features) [Czarnecki and Eisenecker, 2000]. The nature of a constraint is dependent on the type of feature used within the feature model. For Boolean features, a constraint can be expressed as a propositional formula [Batory, 2005]. This form is also sufficient when using features with a finite range of values. For non-Boolean features, constraints may also be expressed in first-order logic [Mannion, 2002]. As stated above, in this thesis, we focus on Boolean features, and thus use propositional formulas as constraints. A feature model can contain multiple constraints that are in conjunction. That means all constraints must be fulfilled by a configuration for that feature model in order to describe a valid variant of a configurable system.

The feature model in Figure 2.1 defines multiple constraints. Most constraints of the feature model are implicitly defined via the tree structure of the feature diagram. For example, each feature in the tree implies its parent feature. As the feature FS is the parent of the feature $NTFS$ the feature diagram defines the constraint $NTFS \Rightarrow FS$. Other constraints arise from the usage of different types of edges and groups in the diagram. Additionally, the feature diagram defines the two cross-tree constraints $NTFS \Leftrightarrow Win$ and $HFS \Leftrightarrow Mac$, which are displayed beneath the feature tree and are explicitly stated as propositional formulas.

Representation

There are many different representations for a feature model. Most notably, *feature diagrams* [Czarnecki and Wąsowski, 2007; Schobbens et al., 2006], *propositional formulas* [Batory, 2005], and textual representations like the *universal variability language (UVL)* [Sundermann et al., 2021]. With feature diagrams, a feature model can be presented in a human-readable way. For storing a feature model within persistent memory, often a textual representation is used. In contrast, for most automated analysis techniques a propositional formula is needed.

All of these representations represent a feature model and by this describe its corresponding problem space. However, some representations may also provide additional information, such a feature hierarchy, feature properties, descriptions, and so on. For describing the problem space, each representation is sufficient, and thus, each representation can be transformed into any other without losing information on the problem space [Feichtinger et al., 2021]. Nevertheless, some of the additional information that can be expressed in one representation are difficult or even impossible to represent in another one, and thus might be lost during a transformation. A common example is the feature hierarchy in a feature diagram, which is lost, when it is transformed into a propositional formula. Still, all dependencies of the features remain unchanged. Within our thesis, we do not focus on any additional information, and are therefore using some representations interchangeably depending on the situation. We visualize feature models with the help of feature diagrams, as this type of representation is more intuitive for most people. For all other purposes and especially for our formalization, we rely on propositional formulas and set notation.

The feature diagram in Figure 2.1 consists of a feature tree and additional cross-tree constraints at the bottom. The feature tree contains all features of the feature model and partly describes their relationships. The feature *Server* is the root of the feature tree and represents the common part of all variants of the configurable system. The features *File System (FS)* and *Operating System (OS)* are both mandatory children of *Server*, which means each variant that contains *Server* must also contain *FS* and *OS*. By contrast, the feature *Logging (Log)* is an optional child of *Server* and is not required if *Server* is part of a variant. The children of *FS* (i.e., *NTFS*, *HFS*, and *EXT*) are part of an or-group, which means that at least one of them must be part of a variant that contains *FS*. The children of *OS* (i.e., *Windows (Win)*, *macOS (Mac)*, and *Debian (Deb)*) are part of an alternative, which means that if *OS* is part of a variant, exactly one of them must be present too. Additionally, the feature model contains two cross-tree constraints, a bi-implication between *NTFS* and *Win* and a bi-implication between *HFS* and *Mac*.

Formalism

Our formal definition of feature model, which we use in the remainder of this thesis, is based on its representation as a propositional formula. Moreover, we use the *conjunctive normal form (CNF)* of propositional formulas. Any propositional formula can be transformed into CNF using standard Boolean algebra.Thus, for any feature model formula, there is at least one equivalent formula in CNF. Practically, this means that every constraint consists of one or multiple clauses in the CNF, which each is a disjunction of literals. Finally, the formula represented by the feature model is a conjunction of all clauses.

Definition 2.1 Feature Model**.** *We define a feature model $\mathcal{M} = (\mathcal{F}, \mathcal{D})$ as a tuple, consisting of a set of features $\mathcal{F}$ and a set of dependencies (also referred to as constraints) $\mathcal{D}$ over $\mathcal{F}$.*

- *Let $\mathbb{F}$ be the universe of all possible feature variables.*
- *The feature set $\mathcal{F} = \{f_1, \ldots, f_n\} \subset \mathbb{F}$ is a finite set, containing all features of the feature model with n being the total number of features.*

- *Based on $\mathbb{F}$, we define the universe of literals $\mathbb{L}$, which represents all possible assignments of all feature variables in $\mathbb{F}$.*
  - *We define the functions $\phi^+ : \mathbb{F} \to \mathbb{L}$ with $\phi^+(f) = f$ and $\phi^- : \mathbb{F} \to \mathbb{L}$ with $\phi^-(f) = \neg f$ that respectively return the positive and negative literal for a feature (i.e., both possible assignments for a Boolean feature).*
  - *Note that the following expressions are equivalent:*
    - $l \equiv \neg\neg l$, $(l \in \mathbb{L})$
    - $\phi^+(f) \equiv \neg\phi^-(f)$, $(f \in \mathbb{F})$
    - $\phi^-(f) \equiv \neg\phi^+(f)$, $(f \in \mathbb{F})$
  - *Further, we define the function $\mathcal{L} : \mathbb{M} \to 2^{\mathbb{L}}$ with $\mathcal{L}(\mathcal{M}) = \{\phi^+(f) \mid f \in \mathcal{F}(\mathcal{M})\} \cup \{\phi^-(f) \mid f \in \mathcal{F}(\mathcal{M})\}$ that maps a feature model to a set of literals such that for each feature $f \in \mathcal{F}$, it contains a corresponding positive and negative literal.*
- *The set of dependencies $\mathcal{D} = \{d_1, \ldots, d_m\} \subseteq 2^{\mathbb{L}}$ is a finite set, containing all dependencies (i.e., constraints) of the feature model with m being the total number of constraints.*
  - *Every dependency d is a subset of the set of literals $\mathcal{L}(\mathcal{M})$ and represents a disjunction of the contained literals.*
  - *To convert a clause d to its propositional formula we define the function $\bigvee : 2^{\mathbb{L}} \to \mathbb{P}$ with $\bigvee(d) = \bigvee_{l \in d} l$.*
  - *Further, we define the function $\phi : \mathbb{M} \to \mathbb{P}$ with $\phi(\mathcal{M}) = \bigwedge_{d \in \mathcal{D}(\mathcal{M})}(\bigvee(d))$ that maps a feature model to a propositional formula in CNF such that the formula is a conjunction of all dependencies in $\mathcal{D}$ and describes the valid problem space.*
- *We denote the set of features for a given feature model $\mathcal{M}$ with $\mathcal{F}(\mathcal{M})$ and the set of dependencies for the model with $\mathcal{D}(\mathcal{M})$.*

2.1.2 Configuration

A configuration describes a product (i.e., variant) of a configurable system. It stores all values assigned to the feature variables of a feature model. For Boolean features this mean that they can either be assigned the value *true* (i.e., selecting the feature) or *false* (i.e., deselecting the feature). From a configuration, a corresponding variant can be derived using the variability mechanism of the system [Apel et al., 2013; Clements and Northrop, 2001; Pohl et al., 2005]. In order to derive a valid variant, all assigned values must be in accordance with the dependencies described in the feature model.

Formalism Our formal notation of configurations is based on the set of literals $\mathcal{L}(\mathcal{M})$ that can be derived from a feature model.

Definition 2.2 Configuration**.** *We define a configuration as set of literals* $c = \{l_1, \ldots, l_k\} \subset \mathbb{L}$*, where* k *is the number of literals contained in the configuration.*

- *Iff a configuration* c *assigns the value true to a feature variable* f*, it contains the positive literal of* f *(i.e.,* $\phi^+(f) \in c$*).*
- *Analogous, iff a configuration* c *assigns the value false to a feature variable* f*, it contains the negative literal of* f *(i.e.,* $\phi^-(f) \in c$*).*
- *A configuration* c *may never contain more than one literal of any feature variable (i.e.,* $\forall\ f \in \mathbb{F} : \{\phi^+(f), \phi^-(f)\} \not\subseteq c$*).*
- *A configuration is compatible with a given feature model* $\mathcal{M}$ *iff it only contains literals from that feature model (*$c \subset \mathcal{L}(\mathcal{M})$*).*
- *We define the function* $\bigwedge : 2^{\mathbb{L}} \to \mathbb{P}$ *with* $\bigwedge(c) = \bigwedge_{l \in c} l$ *to map a configuration to a propositional formula that is a conjunction of all its contained literals.*

Completeness Property In order to derive a variant from a configuration, the configuration must assign a value to every feature variable of the feature model. However, during the configuration process, a configuration can be in a state, where it does not assign a value to every feature variable, yet. We call such a configuration *partial*. Otherwise, if all features are defined by a configuration, we call it *complete*. Formally, we define this property with the function:

$$complete(c, \mathcal{M}) = \begin{cases} true & |c| = |\mathcal{F}(\mathcal{M})| \\ false & \text{otherwise} \end{cases}$$

Validity Property If a configuration is complete, every feature variable has an assigned value, and thus we can check, whether this complete assignment satisfies the feature model formula. Given a feature model formula $\mathcal{M}$, if a configuration contains *at least one* literal from a clause in $\mathcal{D}(\mathcal{M})$, it satisfies this clause. Contrary, if a configuration contains *all* complementary literals of a clause, it contradicts this clause, and hence the entire feature model formula. Thus, we call a complete configuration *valid*, if it satisfies *all* clauses of a feature model If a configuration is incomplete or contradicts *at least one* clause, we call it *invalid*. We express this property with the following function:

$$valid(c, \mathcal{M}) = \begin{cases} true & complete(c, \mathcal{M}) \wedge \forall d \in \mathcal{D}(\mathcal{M}) : c \cap d \neq \emptyset \\ false & \text{otherwise} \end{cases}$$

We denote the set of all valid configurations of a feature model $\mathcal{M}$ with $\mathcal{C}(\mathcal{M})$.

Satisfiability Property According to our definition above, only a complete configuration can be valid, a partial configuration is always invalid. However, there is a difference between a partial configuration that can lead to a valid configuration by including more literals until it is complete and a partial configuration that already

contradicts a feature model formula. In order to differentiate these cases, we rely on the *satisfiability* property. We call a partial configuration *satisfiable*, if it is a subset of a valid configuration. In contrast, if every complete configuration that is a super set of a partial configuration is invalid, we call the partial configuration *unsatisfiable*. We express this property with the following function:

$$satisfiable(c, \mathcal{M}) = \begin{cases} true & \exists c' \supseteq c : valid(c', \mathcal{M}) \\ false & \text{otherwise} \end{cases}$$

2.1.3 Feature Model Analysis

A feature model may contain a number of anomalies, such as dead features or redundant constraints [Benavides et al., 2010]. An anomaly within a feature model may indicate that the problem space described by the model is not in accordance with the intentions of the modeler. For instance, the model could contain a contradiction, resulting in an empty problem space.

In order to find anomalies within a feature model, there are several automated reasoning techniques (i.e., feature model analyses) [Benavides et al., 2010]. Throughout this thesis we use and improve several of these analyses. Thus, in the following, we provide a brief overview of the most important feature model anomalies and how an analyses can be implemented based on our presented formalism. To this end, we show how each analysis can be reduced to one or more instances of the Boolean satisfiability problem (SAT), which determines whether a propositional formula has at least one solution. This reduction makes it possible to shift the complex part of any analysis to the same standard problem, for which there are efficient algorithms for solving it (i.e., SAT solvers) [Benavides, 2007; Eén and Sörensson, 2004; Janota, 2008; Le Berre and Parrain, 2010; Mendonça, 2009].

Detecting Void Feature Models We call a feature model *void*, if it has no valid configuration. This can be the case if one or more of the constraints or a combination of them are a contradiction. In this case the feature model describes an empty problem space, which makes it useless for most applications. We formalize this anomaly with the following expression:

$$void(\mathcal{M}) = \begin{cases} true & \nexists c : valid(c, \mathcal{M}) \\ false & \text{otherwise} \end{cases}$$

Determining whether a feature model is void using the satisfiability problem is quite straight-forward. A feature model $\mathcal{M}$ is void if and only if the representing formula is unsatisfiable (i.e., $SAT(\phi(\mathcal{M})) = false$).

Detecting Core and Dead Features A feature variable of a feature model is called *core* if there is no valid configuration for the model in which the feature variable is assigned the value *false*. Analogous, a feature variable is called *dead* if there is no valid configuration for the model in which the feature variable is assigned the value *true*. As an example, consider the feature model from Figure 2.1. The

three feature variables *Server*, *OS*, and *FS* must be *true* in every valid configuration. Thus, these three feature variables are core.

We use the following two expressions to formalize both anomalies:

$$core(f, \mathcal{M}) = \begin{cases} true & \forall c \in \{c \mid valid(c, \mathcal{M})\} : \phi^+(f) \in c \\ false & \text{otherwise} \end{cases}$$

$$dead(f, \mathcal{M}) = \begin{cases} true & \forall c \in \{c \mid valid(c, \mathcal{M})\} : \phi^+(f) \notin c \\ false & \text{otherwise} \end{cases}$$

Whether a single feature variable f is core can be reduced to an instance of SAT. For this, we start by constructing a new formula $\phi(\mathcal{M}) \Rightarrow \phi^+(f)$. Assuming that the feature model is not void (i.e., $SAT(\phi(\mathcal{M})) = true$), iff this new formula is a tautology, then every valid configuration of the feature model formula implies that f is *true*, which means that there is no valid configuration where f is *false*, and therefore f must be core. Unfortunately, we cannot directly show that a formula is a tautology using SAT. However, we can show that the complement of a formula is a contradiction, which is equivalent. Therefore, we apply SAT to the complement of our constructed formula:

$$\begin{aligned} & \neg(\phi(\mathcal{M}) \Rightarrow \phi^+(f)) \\ \equiv\ & \neg(\neg\phi(\mathcal{M}) \vee \phi^+(f)) \\ \equiv\ & \phi(\mathcal{M}) \wedge \neg\phi^+(f) \\ \equiv\ & \phi(\mathcal{M}) \wedge \phi^-(f) \end{aligned}$$

Consequently, iff $SAT(\phi(\mathcal{M}) \wedge \phi^-(f)) = false$ then f is core. For determining whether a feature variable is dead, we can apply a similar reasoning. A feature variable f is dead iff $SAT(\phi(\mathcal{M}) \wedge \phi^+(f)) = false$.

Finding Atomic Sets An atomic set is a set of feature variables that cannot be assigned a value independently from each other in any valid configuration. For instance, the two feature variables $NTFS$ and Win in our example in Figure 2.1 have the dependency $NTFS \Leftrightarrow Win$. Thus, the value assigned to $NTFS$ is always dependent on the value assigned to Win and vice-versa. If we assign the value *true* to $NTFS$ then we must also assign *true* to Win. In this case $NTFS$ and Win are in the same atomic set. We express whether two feature variables f and f' are in the same atomic set with the following formula:

$$atomic(f, f', \mathcal{M}) = \begin{cases} true & \forall c \in \{c \mid valid(c, \mathcal{M})\} : \phi^+(f) \in c \Leftrightarrow \phi^+(f') \in c \\ false & \text{otherwise} \end{cases}$$

The set of all atomic sets of a feature model is a partition of the set of all literals $\mathcal{L}(\mathcal{M})$. As a consequence, we could replace all occurrences of a feature variable in a feature model formula with any other variable in the same atomic set.

Analogous to the determination of core and dead features, we can reduce the question of whether two feature variables are in the same atomic set to the solving of a SAT instance for a newly constructed formula. In this case, two features f and f' are in the same atomic set iff the following formula is a contradiction $\phi(\mathcal{M}) \wedge \neg(\phi^+(f) \Leftrightarrow \phi^+(f'))$.

Finding Redundant Constraints A redundant constraint is a constraint that can be removed from the feature model without changing its valid problem space.

$$redundant(d, \mathcal{M}) = \begin{cases} true & \phi(\mathcal{M}) \equiv \bigwedge_{d' \in \mathcal{D}(\mathcal{M}) \setminus \{d\}} (\bigvee(d')) \\ false & \text{otherwise} \end{cases}$$

We can imply a similar reasoning as for the other analyses to determine whether a constraint d is redundant using SAT. If we construct a new formula consisting of a conjunction of all constraints except d and this formula still implies d, then d is redundant. More formally, a constraint d is redundant iff $SAT(\bigwedge_{d' \in \mathcal{D}(\mathcal{M}) \setminus \{d\}} (\bigvee(d')) \wedge \neg \bigwedge(d)) = false$.

We further differentiate between *internally* and *externally redundant* clauses. A clause is internally redundant, if it is itself a tautology. Thus, the redundancy of the clause is not dependent on other clauses. For instance, the clause $EXT \vee Log \vee \neg Log$ is a tautology, because it contains the literal Log and its complement $\neg Log$. Externally redundant clauses are no tautologies, but are only redundant in context with other clauses. For example, the clause $\neg FS \vee EXT \vee HFS \vee NTFS$ is externally redundant with regards to the clause $EXT \vee HFS \vee NTFS$.

2.2 Solution Space

The problem space is mapped to the solution by a variability mechanism. This mechanism creates a variant by combining a list of implementation artifacts that is derived from a particular configuration [Apel et al., 2013]. The solution space of a configurable system comprises all variants that can be build from combining its implementation artifacts for any valid configuration. Within our thesis we are interested in the configuration process that leads to a valid configuration. For this we also consider the variability of implementation artifacts, which can be used in the configuration process. Specifically, we consider presence conditions that determine when a particular implementation artifact is included in a variant.

```
1  // #if Log
2  public class Logger {
3    public void log(String message) {
4      System.out.println(message);
5    }
6
7    public void logOS() {
8      StringBuilder sb = new StringBuilder();
9      sb.append("OS = ");
10     // #if Deb
11     sb.append("linux");
12     // #endif
13     // #if Win
14     sb.append("windows");
15     // #endif
16     log(sb.toString());
17   }
18 }
19 // #endif
```

Listing 2.1 Class `Logger` of the example system *Server*

2.2.1 Implementation Artifacts

An implementation artifact is any component of a configurable system that is used as a part of a variant. These can be, for instance, source code, textual data, model files, and audio and video data. Similar to features, there are mandatory implementation artifacts that form the common code base of a system and optional implementation artifacts that are conditionally dependent on a specific configuration. For example, regarding our feature model in Figure 2.1, an optional implementation artifact may be the class `Logger` in Listing 2.1, which implements some logging functionality for the server system. The inclusions of the class in a variant should be dependent on the selection of the feature *Logging*.

Each configurable system employs at least one variability mechanism which is responsible for combing implementation artifacts into a final variant. Popular variability mechanism are, for instance, *preprocessors* [Liebig et al., 2010; Medeiros et al., 2013; Munge Development Team, 2011; Stallman and Weinberg, 1987], plug-in frameworks [Acher et al., 2014], and runtime-variability [Hallsteinsen et al., 2008]. For instance, when employing preprocessors, such as the C preprocessor [Stallman and Weinberg, 1987], implementation artifacts are annotated with expressions that specify whether to include an artifact under a given configuration or not. If a given configuration fulfills the annotated expression of an implementation artifact the artifacts is included and otherwise it is ignored.

2.2.2 Presence Conditions

Whether and how an implementation artifact is used in a variant is dependent on the employed variability mechanism. However, it is possible to abstract the description of

the variability of a particular implementation artifact from the concrete mechanism by using the notion of *presence conditions*. A presence condition is a logical formula that describes whether a particular implementation artifact is considered in a variant given a configuration. We call a presence condition *active* under a given configuration, iff the configuration satisfies the formula of the presence condition and *inactive* otherwise. For instance, in Listing 2.1, the class `Logger` contains preprocessor annotations that enable its variability. The annotations `#if Log` in Line 1 and `#endif Log` in Line 19 specify that everything between these two annotations (i.e., the entire class) is only included in variants in which the feature Logging is selected. Thus, the formula for the presence condition of the class Logger is *Log*. The class will be present in all variants that are derived from a configuration that defines *Log* as *true* and the class will not be present for any other configuration.

Presence conditions are independent from the concrete variability mechanism(s) employed by the configurable system. They can be determined regardless of how artifacts are combined, simply by comparing the configurations that contain an artifact with the configuration that do not. This means that presence conditions are a representation of the variability of an implementation artifact and do not have to be stated explicitly in the code (unless this is required by the variability mechanism). Thus, in theory, presence conditions can be determined for any deterministic variability mechanism.

Within our thesis, we consider presence conditions to comprehend the variability of single statements within the source code. For each statement in the source code, we can identify a presence condition. If the formula of a presence condition is satisfied by a given configuration, this statement will be part of the variant resulting from that configuration. In contrast, if the presence condition is not satisfied the statement will not be part of the variant. For instance, the presence condition of Line 11 of Listing 2.1 consists of the formula $Log \wedge Deb$ due to the nested annotations within the source code. In contrast, the presence condition of Line 14 consists of the formula $Log \wedge Win$ and formula of Line 9 is simply Log. The configuration $c = \{Server, FS, OS, Log, NTFS, Win, \neg HFS, \neg EXT, \neg Mac, \neg Deb\}$ will lead to a variant that contains the Lines 9 and 14, but not the Line 11.

Formalism We base the formal notation of presence conditions on their propositional formulas in disjunctive normal form (DNF). In order to include an artifact in a variant, at least one clause of a DNF must be satisfied, which means that all of its literals must be selected within a configuration. The DNF representation for presence conditions has the advantage that all combinations of literals that satisfy it can be immediately identified.

Definition 2.3 Presence Condition**.** *We define a presence condition as set of clauses* $\mathcal{P} = \{d_1, ..., d_n\}$*, where* n *is the number of clauses.*

- *A clause* $d \subseteq \mathcal{L}(\mathcal{M})$ *represents a conjunction of literals that would satisfy the presence condition.*
- *The disjunction of all clauses in* $\mathcal{P}$ *forms the propositional formula of the presence condition in DNF.*

- *We define the function* $\bigwedge : 2^{\mathbb{L}} \to \mathbb{P}$ *with* $\bigwedge(d) = \bigwedge_{l \in d} l$ *that maps a clause of a presence condition to a conjunction of its containing literals.*
- *Further, we define the function* $\phi : 2^{2^{\mathbb{L}}} \to \mathbb{P}$ *with* $\phi(\mathcal{P}) = \bigvee_{d \in \mathcal{P}}(\bigwedge(d))$ *that maps a presence condition to a propositional formula in DNF.*

For example, the presence conditions for the Lines 9, 11, and 14 from Listing 2.1 are written as $\mathcal{P}_9 = \{\{Log\}\}$, $\mathcal{P}_{11} = \{\{Log\}, \{Deb\}\}$, and $\mathcal{P}_{14} = \{\{Log\}, \{Win\}\}$.

Activity Given a valid configuration the propositional formula of a presence conditions can be either satisfied, in which case we call the presence condition *active* or unsatisfied in which case we call it *inactive*. We describe this property with the following expression:

$$active(\mathcal{P}, c, \mathcal{M}) = \begin{cases} true & valid(c, \mathcal{M}) \wedge \exists d \in \mathcal{P} : d \subseteq c \\ false & \text{otherwise} \end{cases}$$

2.3 Configuration Process

The *configuration process* is a process that creates one or more valid configurations for a configurable system. For the purpose of our thesis, we differentiate between three different types of configuration processes, the *manual*, *semi-automated*, and *fully-automated* process. In the manual configuration process the configuration is done by hand by one or more users. On the other end of the spectrum lies the fully-automated configuration process. Here a configuration is automatically generated by an algorithm. The user can choose the algorithm and provide suitable parameters, but the actual configuration is done by the algorithm without any intervention from the user. Examples include the random generation of a configuration, the search for an optimized solution, and general sampling [Al-Hajjaji et al., 2016a; Carmo Machado et al., 2014; Lopez-Herrejon et al., 2014; Perrouin et al., 2010; Tartler et al., 2014]. Between these two extremes for creating a configuration is the semi-automated configuration process, which is a combination of automatically deriving parts of a configuration and using interact input from a user. Depending on the degree of the automation, this process can be seen rather as a manual configuration process facilitated by tool support or as an automated process that interacts with the user. The actual interaction and tool-support of this process can take many different forms. However, it always aims to guide a user towards a valid configuration by easing their mental effort. In the following, we provide more details on all three types of configuration processes. Nevertheless, we especially focus on the semi- and fully-automated configuration process, as these two play a major roll in the remainder of our thesis.

2.3.1 Manual Configuration Process

In a manual configuration process a user sets a value for each feature by hand. This can be done one feature at a time or in a batch containing multiple features. During this process, there is no automation, tool-support, or analysis. In the end, the result of the process is a complete configuration. As there is no mechanism involved that checks or even enforces the validity of the configuration, the final configuration may be invalid. This means that the user must take care of verifying the validity of the configuration themselves at the end of the process either by hand or with a downstream validity analysis. Consequently, an invalid configuration must also be fixed by the user manually. The fact that this process is error-prone and involves much manual effort makes this type of configuration process only feasible for small projects or systems that involve no complex constraints. Since we are interested in facilitating the configuration management of large-scale configurable systems, the manual process is therefore of limited interest within our thesis.

2.3.2 Semi-Automated Configuration Process

The semi-automated configuration process involves some degree of automation in the form of automated analyses or automated (de)selection of features. We look a the semi-automated configuration process from the perspective of an enhancement over the manual configuration process. It adds some form of tool-support, which is supposed to guide the user to a desired and valid configuration and overall reduce the manual effort required to arrive at a complete configuration.

There are several techniques that may facilitate the configuration process. On a high level, we can differentiate between techniques for information management, that show additional information to the user or hide irrelevant information from the user and configuration manipulation, which automatically assign values to features under specific circumstances. Information management can for example include visual guidance [Martinez et al., 2014; Nestor et al., 2008], hiding already defined features, providing explanations of conflicts, showing whether a partial configuration is still satisfiable, and presenting recommendations [Pereira et al., 2016b]. These presented information can enable a user to make discussions more efficiently and less error-prone. On the other hand, configuration manipulation may alter a partial configuration by, for example, resolving conflicts [Benavides et al., 2007; Ochoa et al., 2015; White et al., 2008], assigning values implied by the current state of the configuration (i.e., decision propagation), assigning default values, random values, or values based on a user's preference (i.e., auto-completion). The motivation behind configuration manipulation is to avoid conflicts and by this invalid configurations and to speed up the configuration process by reducing the number of manual decisions a user has to make.

In our thesis, we mainly focus on some of these techniques from configuration manipulation in order to improve their efficiency or effectiveness. Especially, we are interested in *decision propagation* and *auto-completion*. Thus, in the following we provide fundamental details on these two techniques.

Decision Propagation

Decision propagation is a technique to infer implied values for currently undefined features from a satisfiable partial configuration [Batory, 2005]. A decision can, for example, consist of assigning a value to one or multiple feature variables, adding a dynamic constraint, or any kind of action that leads to a partial configuration or a restriction of the current configuration space. In this work, we focus on decisions that consist of assigning a value to a single feature variable, as this is the most basic and direct way to manipulate a partial configuration. Every time a user makes a decision, decision propagation determines all features that are implicitly defined (i.e., conditionally core and dead features) and assigns corresponding values for these features in the current configuration. For instance, when creating a configuration for the feature model in Figure 2.1, the decision to assign the value *true* to the feature variable *NTFS* would also imply that the feature variable *Windows* must be assigned the value *true*. In addition, the feature variables *Mac*, *HFS*, and *Deb* must be set to *false*. As there is no other possibility for setting these feature variables, if *NTFS* is *true*, it is reasonable to automatically assign the corresponding values in the configuration.

Besides speeding-up the configuration process by reducing the number of necessary manual decisions, decision propagation solves one major problem from the manual configuration process. When making decision, it is possible that users unknowingly assign contradictory values to the feature variables and than later have to backtrack their steps to undo one or more of their decisions. This can be confusing and cumbersome to the user, especially if this occurs multiple times during the configuration process [Thüm et al., 2018]. Assuming that a user makes their decision one-at-a-time, employing decision propagation completely solves this problem. As every implied value is automatically assigned, it is not possible to assign a contradictory value. Thus, a partial configuration will always be satisfiable and a complete configuration will always be valid. Effectively, decision propagation avoids backtracking, meaning that users never have to revoke their decisions in order to obtain a valid configuration.

Auto-Completion

Given a partial configuration auto-completion assigns values to currently undefined features, such that afterwards the configuration is complete [Mendonça et al., 2009a; Pereira et al., 2016a]. Similarly to decision propagation, this technique tries to reduce the amount of manual decision a user has to make, and by this speed-up the configuration process. There exists many different strategies to automatically assigning values. Two of the most basic strategies are to assign a default value (e.g., *false*) or a random value to all undefined features. For both strategies the resulting configuration can be invalid, which is unsuitable for most applications. Thus, a more advanced strategy is to interlace automatically assigning values with decision propagation. After each automatic assignment decision propagation is applied. Analogous to regular decision propagation, the complete configuration will be valid, assuming that the given partial configuration is satisfiable. For this strategy, it does not matter whether we assign random or default values. Another strategy is to use a logic solver to compute a valid solution (i.e., configuration) [Janota, 2008; Mendonça, 2009]. The resulting configuration will be valid, but is dependent on the internal behavior of the used solver.

2.3.3 Fully-Automated Configuration Process

A fully-automated configuration process does not involve any user interaction aside from providing initial parameters for the process. An automated configuration process can produce one or more configurations. Although there exists many different kinds of automatically creating a configuration, in this thesis we are mainly concerned with different kinds of configuration sampling. Configuration sampling is a process that produces a set of complete configurations (i.e., a sample), that is supposed to resemble the variability of the feature model [Medeiros et al., 2016]. A sample can then be used to run analyses against each contained configuration or test the corresponding variants using single-system testing frameworks. It also allows to measure non-functional properties, such as memory footprint, energy consumption, execution time, etc.

A straight-forward testing strategy for configurable systems is product-based testing. For this purpose, a set of products is derived from a set of different configurations and then test cases are run on each selected product respectively [Thüm et al., 2014a]. As testing every possible configuration in a product-based manner is usually not feasible due to an enormous configuration space, sampling strategies have been defined to generate a small yet representative set of configurations to test (i.e., a configuration sample). One such sampling strategy is *t-wise interaction sampling*, which aims to generate a preferably small sample that covers all possible interactions of features of degree t [Cohen et al., 2008; Marijan et al., 2013]. Using t-wise interaction sampling, developers can ensure that all interactions of at most t features (e.g., all selected, none selected, only one selected, etc.) are indeed contained in at least one configuration in the generated sample. T-wise interaction sampling has shown to be a feasible trade-off between testing effectiveness and testing efficiency, as for small values of t (i.e., $t \in \{2, 3\}$) it usually returns a relatively small sample, while also achieving reliable test results [Abal et al., 2018; Kästner et al., 2009; Marijan et al., 2013].

There are many different types of configuration sampling [Medeiros et al., 2016; Varshosaz et al., 2018]. In the following, we focus on random sampling and optimization sampling.

Random Sampling

Random sampling creates a sample consisting of randomly chosen or generated configurations [Varshosaz et al., 2018]. In most cases, the user defines the size of the random sample beforehand and also whether the sample is allowed to include duplicates. There are many different algorithms are able to create a random configurations [Varshosaz et al., 2018]. We briefly introduce the three most popular algorithms. One method of creating a random configuration is to randomly assign a value to every feature. This process can be iterated as many times as necessary to generate new configurations. The major drawback of this method is that due to the constraints of the feature model, many configurations generated in this way can be invalid, and thus must be removed from the sample. Another method is to employ a logic solver (e.g., a SAT solver) that is able to create a configuration from a feature model [Munoz et al., 2019; Oh et al., 2017; Pereira et al., 2016a]. By manipulate the internal state of the solver (e.g., randomizing the order of features

or the strategy of assigning values) is is possible to produce random configurations. This method has the disadvantage that the generated configuration are dependent on the internal behavior of the employed solver. In most cases, the resulting sample will not be uniformly distributed over the problem space, which can be a problem for some applications. Finally, another method is to enumerate all valid configurations and then choosing a random subset of the enumeration with the desired size. This method has the advantage that it is guaranteed to be uniformly distributed over the problem space. However, unfortunately, it is unfeasible to enumerate all configuration for almost any larger system as the configuration space tends to grow exponentially with the number of features [Engström and Runeson, 2011; Halin et al., 2019; Lee et al., 2012].

Optimization Sampling

Optimization sampling tries to create a sample that optimizes one or more properties [Meinicke et al., 2014; Varshosaz et al., 2018]. What properties are considered heavily depends on the desired purpose of the sample. For instance, if a sample is used for product-based testing it is useful to maximize the fault-exposing potential of the variants that can be generated from the configurations in the sample. For measuring non-functional properties, the diversity of the variants of a sample should be maximized. However, some of these properties are hard to measure or hard to predict, especially, when generating multiple configurations. Thus, often other, more easily measurable metrics are used that act as an indicator for the actual property the should be optimized. Popular metrics are, for example, the similarity between configurations in a sample, used feature combinations, code coverage of variants, and testing coverage of variants. In our thesis, we are mainly focused on testing, and thus, we go into detail about some relevant metrics for sampling techniques with this purpose in mind. This includes, coverage of feature combinations (i.e., t-wise interaction coverage), code coverage, and testing coverage.

General Coverage Metrics All following metrics, are metrics hat are based on *coverage criteria*. This allows us to describe the common aspects of these coverage metrics together. In general, coverage metrics measure the degree of coverage of a certain basic set of some elements. In context of configurable system, the basic set contains elements that can be derived from the set of all valid configurations of a feature model, such as lines of code included in a variant and selection of features. For each configuration a coverage set can be determined that is a subset of the basic set:

$$cov(\mathcal{S}) = \bigcup_{c \in \mathcal{S}} cov(c)$$

$$cov(\mathcal{M}) = \bigcup_{c \in \mathcal{C}(\mathcal{M})} cov(c)$$

The union of all coverage set for all configuration within a sample is the coverage set of the sample. The degree of coverage is the ratio between the coverage set of a sample and the basic set:

$$cov^{\circ}(\mathcal{S}, \mathcal{M}) = \frac{|cov(\mathcal{S})|}{|cov(\mathcal{M})|}$$

The goal for most coverage-based sampling techniques is to maximize the degree of coverage for a given coverage criterion (i.e., achieve 100% coverage), while simultaneously minimizing the number of configurations within the generated sample.

T-Wise Interaction Coverage T-Wise interaction coverage is a coverage criterion based on the problem space [Lopez-Herrejon et al., 2015; Varshosaz et al., 2018]. It considered all valid combinations of all subsets of features with the size t. For example, *one-wise* interaction coverage considers all features individually (i.e., subsets with a size of one). In order to achieve 100% coverage, every feature variable that is neither core nor dead must be assigned the value *true* in at least one configuration and also must be assigned the value *false* in at least one configuration. For instance, the three configuration C_1, C_2, and C_3 in the following table achieve a 100% one-wise interaction coverage for the Server system from Figure 2.1:

Feature	**Configurations**		
	C_1	C_2	C_3
NTFS	✓	×	×
HFS	×	✓	×
EXT	×	×	✓
Win	✓	×	×
Mac	×	✓	×
Deb	×	×	✓
Log	✓	×	×

In contrast, *two-wise* interaction coverage considers the combination of every pair of features. For each combination, there are four ways to select two feature variables a and b (i.e., $a \wedge b$, $\neg a \wedge b$, $a \wedge \neg b$, $\neg a \wedge \neg b$). Thus, in total, there are $4 \cdot \frac{n \cdot (n-1)}{2}$ possible combinations, where n is the number of features. However, only valid combinations (i.e., combinations that appear in at least one valid configuration) are of interest.

This can be generalized to *t-wise* interaction coverage where t represent an arbitrary natural number. The general formula, for calculating the total number of possible combinations is $2^t \cdot \binom{n}{t}$. This number scales polynomial with the number of features n and t as exponent and exponentially with the parameter t. Thus, selecting a high t results in high number of combinations to cover. There is no known efficient algorithm for finding the minimal set of configurations even for a feature model that solely contains optional features [Hartman, 2005].

2.4 Summary

In this chapter, we introduced the fundamentals for problem and solution space, including feature models, configurations, implementation artifacts, and presence conditions. We presented the formalism that we use throughout our thesis. Further, we presented the different types of configuration processes, with a special focus on decision propagation for the semi-automated configuration process, which we build upon in Chapter 3, and sampling as a form of the fully-automated configuration process, which we extend in Chapter 4 and Chapter 5.

3. Modal Implication Graphs

This chapter introduces the concept of modal implication graphs. It shares material with the following two publications: Propagating Configuration Decisions with Modal Implication Graphs (ICSE'18) *[Krieter et al., 2018] and* Incremental Construction of Modal Implication Graphs for Evolving Feature Models (SPLC'21) *[Krieter et al., 2021]. First ideas and preliminary evaluations on modal implication graphs were also presented in the master's thesis* Efficient Configuration of Large-Scale Feature Models Using Extended Implication Graphs *[Krieter, 2015].*

In this chapter, we introduce a novel data structure called *modal implication graph* (*MIG*) that aims to reduce the computational effort of certain SAT-based analyses. A MIG stores useful information about feature dependencies in an easily accessible way, which facilitates any analyses that rely on these information. As it requires some computational effort to build a MIG, effectively they can be considered as trade of some initial computational effort for increased performance of all following analyses that employ a MIG. We find that a MIG is a helpful data structure for a semi-automated configuration process, such as an interactive configuration process that uses decision propagation, which we present in this chapter. In addition, a MIG can facilitate a fully-automated configuration approach, such as our basic and advanced sampling concept, which we describe in Chapter 4 and in Chapter 5. Furthermore, a MIG is beneficial in other feature model analyses as well, which discuss to some extend in Chapter 7.

In the following sections, we first give a brief motivation on why a MIG is useful (see Section 3.1). Second, we explain the structure of a MIG and what it represents (see Section 3.2). Third, we explain how a MIG can be used to facilitate feature model analyses (see Section 3.3). Fourth, we introduce a build process that can construct a MIG from a feature model (see Section 3.4). Fifth, we describe optimizations to the build process to enable incremental updates to a MIG (see Section 3.5). Finally, we evaluate the performance of a MIG within feature model analysis and the overhead introduced by the regular and incremental build process (see Section 3.6). This chapter is mainly based on two publications that introduce the original concept of

MIGs [Krieter et al., 2018] and present an incremental build process to speed up the construction of MIGs for evolving feature models [Krieter et al., 2021]. However, beyond these publications, we also present novel ideas regarding the application of MIGs in feature model analyses (see Section 3.3.2) and further details on the regular and incremental build process of MIGs (see Section 3.5 and Section 3.4).

3.1 Motivation

Many of the typically employed analyses, such as finding dead features and redundant constraints, are NP-hard problems, which means that currently there is no known algorithm that is able to solve them efficiently. Still, we attempt to perform analyses on feature models in feasible time by utilizing two of their properties. First, most feature model analyses can be solved by reducing them to multiple instances of the satisfiability problem (SAT) [Mendonça, 2009] or other well-known problems, such as the constraint satisfaction problem (CSP) or the sharp satisfiability problem (#SAT). For instance, determining all dead features of a feature model can be done by constructing and solving a SAT problem for each feature. This reduction allows us to strip the complex part of an analysis and solve it with an optimized solver. Second, when reducing analyses of many real-world feature models the resulting SAT instances are relatively easy to solve [Mendonça et al., 2009b]. In summary, SAT-based analysis solves multiple instance of SAT using a state-of-the-art SAT solver and interpret the results to answer the actual question of the analysis. Thus, a potential efficiency improvement for these analyses lies in decreasing the number of SAT instances necessary to find an answer. With our work, we try to build on this fact and increase the efficiency of SAT-based analyses.

When we consider the SAT reductions used in state-of-the-art SAT-based analyses, we can see that the resulting SAT instances often represent similar problems. For example, regarding our example in Figure 2.1 (on Page 9), if we want to know whether the feature `File System (FS)` is a dead feature, we transform it to the SAT problem $\neg SAT(\phi(\mathcal{M}) \wedge \phi^{+}(FS))$. Analogous, if we want to know whether the feature `Operating System (OS)` is a dead feature, we transform it to the similar SAT problem $\neg SAT(\phi(\mathcal{M}) \wedge \phi^{+}(OS))$. Due to the similarities between such SAT instances, in most cases the corresponding solving processes share many redundancies, even to a point where one SAT instances can be answered simply by reusing an intermediate result from a previous one. Similarly, there can be questions that are transformed into a SAT instances, but are in fact trivial to answer, which results in unnecessary computational effort. By modeling the relationships between features within a MIG, we attempt to address both of these issues.

3.2 Structure of Modal Implication Graphs

A MIG is a directed graph that represent the relationships between the assignments of feature variables of a feature model. It is based on a regular implication graph, but extends it by adding an additional type of edge. This new edge type represent a relationship between two features that is not a direct implication. A regular implication graph is only capable of representing implications between two features. Thus,

when we construct an implication graph for a feature model, all other relationships between features would not be represented within it. In contrast, the extension of a regular implication graph to a MIG enables us include all relationships between all features defined in a feature model within the graph. Thus, a MIG is able to store all information of a feature model formula.

3.2.1 Strong and Weak Edges

The structure of a MIG consists of a set of vertices and two sets of directed edges. A MIG can be constructed for every non-void feature model. For such a feature model, the vertices of a corresponding MIG represent all valid assignments of its feature variables (i.e., all literals). For each Boolean feature, the graph contains two vertices, one that represent a *positive* literal and one that represent a *negative* literal. Note that features that are either core or dead can only have one valid assignment (i.e., *true* for core features and *false* for dead features). For this reason, it is unnecessary to put them into relation to other features. Thus, there are no vertices in the graph that refer to features that are core or dead. An edge between two vertices in a MIG describes a (potential) implication between two literals and is based on some propositional formula that can be derived from the constraints of a feature model. There are two types of edges, *strong* and *weak*. A *strong edge* represents a *direct* implication between two literals. The relationship that a strong edge (v, v') represents can be expressed as the propositional formula $v \Rightarrow v'$. A *weak edge* represents a *potential* implication between two literals, which means there is an implication between both literals under some additional condition. A potential implication can also be expressed as a propositional formula in the following form: $\phi \Rightarrow (v \Rightarrow v')$, where $\phi = (l_1 \wedge ... \wedge l_k)$ is some conjunction of literals with $k \geq 1$. If no feature variable has an assigned value, the condition of a weak edge is usually not satisfied, which means that an inclusion of v in a configuration *does not* imply an inclusion of v'. However, if a partial configuration assigns values to some feature variables such that the condition of a weak edge is satisfied, then the implication of that weak edge is no longer potential. In this case, an inclusion of v in the configuration *does* imply an inclusion of v'.

Formally, we define a MIG as follows:

Definition 3.1 Modal Implication Graph**.** *A modal implication graph* $\mathcal{G} = (\mathcal{V}, \mathcal{D}^U, \mathcal{D}^S, \mathcal{D}^W)$ *is a 4-tuple consisting of a set of vertices* $\mathcal{V}$*, a set of unit clauses* $\mathcal{D}^U$*, a set of 2-clauses* $\mathcal{E}^S$*, and a set of complex clauses* $\mathcal{D}^W$*.*

- $\mathcal{V}$ *contains all vertices of the graph. Each vertex represents a literal of the corresponding feature model.*
- $\mathcal{D}^U$ *contains all literals that represent core and dead feature variables in the corresponding feature model. Note that:*
  - *No core or dead feature variable is represented by a regular vertex (i.e.,* $\mathcal{V} \cap \mathcal{D}^U = \emptyset$*)*
  - *A feature can be either core or dead, but not both (i.e.,* $\nexists v \in \mathcal{D}^U : \neg v \in \mathcal{D}^U$*).*

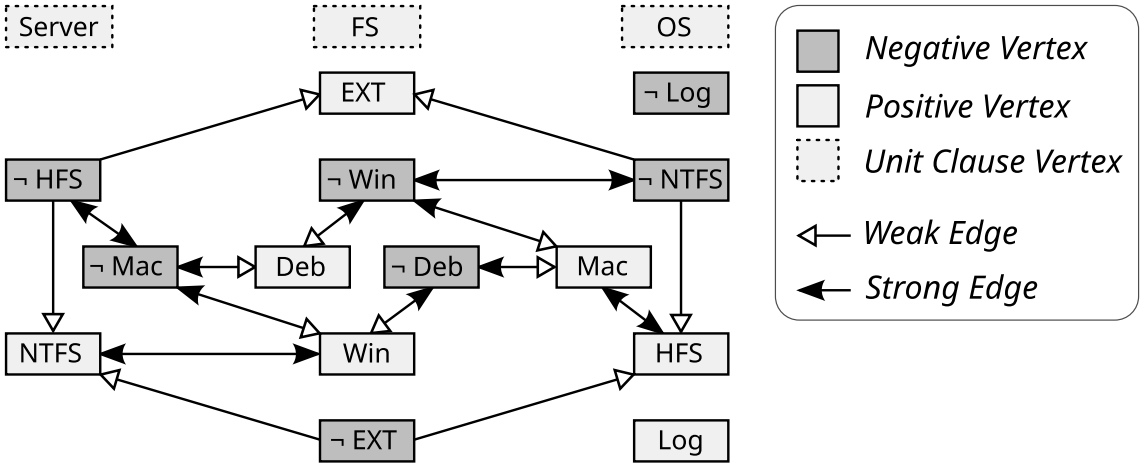

Figure 3.1: MIG for the *Server* feature model from Figure 2.1 (on Page 9)

- $\mathcal{D}^S$ *contains all clauses of the corresponding feature model that contain exactly two literals. All strong edges can be derived from* $\mathcal{D}^S$*:*
  - *A strong edge* $e = (v, v')$ *with* $v, v' \in \mathcal{V}, v \neq v'$ *is a directed edge between two vertices of the graph.*
  - *The set of all strong edges* $\mathcal{E}^S(\mathcal{G})$ *is implicitly inferred from* $\mathcal{D}^S$ *as follows:* $\mathcal{E}^S(\mathcal{G}) = \{(\neg v, v') \mid \{v, v'\} \in \mathcal{D}^S, \neg v \neq v'\}$.
- $\mathcal{D}^W$ *contains clauses with three or more literals from the corresponding feature model. All weak edges can be derived from* $\mathcal{D}^W$*:*
  - *A weak edge* $w = (v, v')$ *with* $v, v' \in V, v \neq v'$ *is a directed edge between two vertices of the graph.*
  - *The set of all weak edges* $\mathcal{E}^W(\mathcal{G})$ *is implicitly inferred from* $\mathcal{D}^W$ *as follows:* $\mathcal{E}^W(\mathcal{G}) = \{(\neg v, v') \mid \{v, v', ...\} \in \mathcal{D}^W, \neg v \neq v'\}$.

In Figure 3.1, we show a depiction of a MIG for our example feature model in Figure 2.1. Each feature has two vertices in the MIG, a vertex representing its positive literal and a vertex representing its negative literal. The features *Server*, *Operating System*, and *File System* are an exception. They do not appear in the MIG as regular vertices, because they are core features. Thus, they are part of the set $\mathcal{D}^U$. There are multiple strong edges, for example from the vertex *Deb* to the vertex $\neg Win$ and from $NTFS$ to WIN and vice-versa. This means that if a valid configuration contains the literal *Deb* (i.e., the feature variable *Deb* is assigned the value *true*) then it must also contain the literal $\neg Win$ (i.e., the feature variable Win must be assigned the value *false*). There are also multiple weak edges, for instance from $\neg Win$ to *Deb* and $\neg Deb$ to Win. This means that if a configuration contains the literal $\neg Win$ then under some condition it must also contain the literal *Deb*. In this case, the condition is satisfied for all partial configurations that contain the literal $\neg Mac$. This can also be expressed as $\neg Mac \Rightarrow (\neg Win \Rightarrow Deb)$.

3.2.2 Strong and Weak Paths

In addition to directed edges, we can also define relationships of literals via paths within the graph. Formally, we define a path in a MIG as follows:

Definition 3.2 MIG Path**.** *A path $p = (e_1, \ldots, e_k)$ within a MIG $\mathcal{G}$ is a finite, non-empty sequence of k edges with $p \in (\mathcal{E}^S(\mathcal{G}) \cup \mathcal{E}^W(\mathcal{G})^k$, in which each edge occurs at most once and for which there is a sequence of vertices $(v_1, \ldots, v_{k+1}) \in \mathcal{V}(\mathcal{G})^{k+1}$ such that $\forall i \in [1, k] : e_i = (v_i, v_{i+1})$. Further, we define the following properties of a path:*

- *Let $p = (e_1, \ldots, e_k)$ and $e_1 = (v_1, v_2)$ and $e_k = (v_k, v_{k+1})$, then we call v_1 the source vertex and v_{k+1} the target vertex of the path p.*
- *A* strong path *is a path that consists of only strong edges (i.e., $\forall e \in p : e \in \mathcal{E}^S(\mathcal{G})$). We denote the set of all strong paths within a MIG with $\mathcal{P}^S(\mathcal{G})$.*
- *A* weak path *is a path that contains at least one weak edge (i.e., $\exists e \in p : e \in \mathcal{E}^W(\mathcal{G})$). We denote the set of all weak paths within a MIG with $\mathcal{P}^W(\mathcal{G})$.*

A path between two vertices represents the same type of relationships between two literals as edges. To this end, we can partition all paths into two categories, *strong paths* and *weak paths*. A *strong path* is a path between two vertices that consists of only strong edges. It has the same effect as a strong edge. If the source vertex of a strong path is included in a configuration, the target vertex of the path must also be included in that configuration. This is due to the transitivity of implications (i.e., if $v_1 \Rightarrow v_2$ and $v_2 \Rightarrow v_3$ then $v_1 \Rightarrow v_3$). For instance, in Figure 3.1, there is a strong edge from Deb to $\neg Win$ and a strong edge from $\neg Win$ to $\neg NTFS$. Thus, there is a strong path $p = ((Deb, \neg Win), (\neg Win, \neg NTFS))$ from Deb to $\neg NTFS$, which translates into the propositional formula $Deb \Rightarrow \neg NTFS$.

In contrast, a *weak path* is a path that contains at least one weak edge. A weak path is analogous to a weak edge, meaning that it represents a potential implication between two literals. This potential implication is also dependent on a condition, which is the conjunction of the conditions of all weak edges contained in a weak path. If the condition is satisfied, then the inclusion of the literal represented by the source vertex implies the inclusion of the literal represented by the target vertex. For example, in Figure 3.1, there is a weak path $p = ((\neg NTFS, \neg Win), (\neg Win, Deb))$ from $\neg NTFS$ to Deb. This is the opposite traversal direction from our previous example, but in this case the edge $(\neg Win, Deb)$ is weak, and thus the entire path is also weak. The potential implication represented by this path is $\neg NTFS \Rightarrow Deb$, which is *true* under the condition $\neg Mac$. As this path contains only one weak edge the condition is the same as the condition from this edge.

Furthermore, if there is no path between two given vertices it means that their corresponding literals have no relationship at all and can *always* be included independently from each other. For example, the vertices Log and $\neg Log$ are disconnected from all other vertices in the MIG, and thus unreachable from any other vertex via any combination of edges. Therefore, the feature Log can be configured independently from any other feature. Another less obvious example is that the vertex Win is not reachable from the vertex EXT. Thus, the literal Win can be included in any

configuration independently from whether the literal EXT is included in the same configuration or not.

In general, if we want to determine whether there is an implication from one given literal v to another literal v' under some partial configuration c, we can follow the following procedure: First, we check whether there is any path at all that leads from v to v'. If there is no path, then v is independent from v'. Otherwise, there is a non-empty set of paths, which all lead from v to v'. Second, we check whether the set of paths contains at least one strong path. If this is the case, then $v \Rightarrow v'$ is *true*. Otherwise, all paths from v to v' are weak. Third, we check whether there is a weak path for which its condition is satisfied by c. If there is any, then $v \Rightarrow v'$ is *true*. Otherwise, v is independent from v' in c. The iterative application of this procedure is the basis for using a MIG to facilitate the process of decision prorogation, which we explain in the detail in the following section.

3.2.3 Completeness and Minimality Property

In Figure 3.1, we show a correct MIG for the feature model in Figure 2.1. We can use this MIG to determine the implication between two literals, as we described above. However, the effectiveness of a MIG during its application may be increased by optimizing its sets of edges. In general, we consider a MIG to be more effective the more strong paths and the fewer weak paths it contains. One of the main applications of a MIG is to find vertices that are connected to a given vertex by traversing along the edges. When traversing along weak paths, this may results in solving a SAT instance in order to check whether the condition of a weak path is met. Thus, the effort of traversal can be simplified by reducing the number of weak and increasing the number of strong edges in the MIG.

The reduction of weak edges accomplishes two things. First, the branching degree and thereby the total number of paths is reduced, which speeds up traversal. Second, the number of weak paths is reduced, which reduces the potential number of SAT instance that need to be solved. The increase of strong edges accomplishes a similar goal. If the inclusion a new strong edge creates a strong path between two vertices that were previously only connected via weak paths, it again reduces the potential number of SAT instance. For example, in Figure 3.1 the edges of the MIG are not optimized. There is a weak path from Deb to EXT (e.g., via $\neg Win$, Mac, HFS, and $\neg NTFS$). Although there is no strong path between both vertices the statement $Deb \Rightarrow EXT$ is always *true* due to the other clauses in the feature model formula. Thus, there exists an implicit strong edge from Deb to EXT that is not yet part of the MIG. If we add this strong edge to the MIG, there is no SAT instance required to check whether Deb implies EXT, because this implication would be explicitly represented by the strong edge. For the MIG in Figure 3.1 one other implicitly strong edge from $\neg EXT$ to $\neg Deb$ can be derived.

Completeness To express whether a MIG is optimized, we define the two properties *completeness* and *minimality* for a MIG. The completeness property of a MIG regards its set of strong edges. A modal implication graph $\mathcal{G}$ is *complete* if each direct implication that can be inferred from the underlying feature model $\mathcal{M}$ is represented

as a strong edge (i.e., $\mathcal{E}^S(\mathcal{G}) = \{(v, v') \in \mathcal{V}(\mathcal{G})^2 \mid TAUT(\phi(\mathcal{M}) \Rightarrow (v \Rightarrow v'))\}$). If a MIG is complete, it means that there is no implicit strong edge that is not explicitly contained in its set of strong edges. As we have shown, the MIG in Figure 3.1 contains such implicit strong edges, and thus it is not complete. We can measure the degree of completeness of a MIG by computing the ratio between the size of its set of strong edges and the number of all implications that can be inferred from the feature model formula. We express this with the following function:

$$\mathcal{E}_S^* = \{(v, v') \in \mathcal{V}(\mathcal{G})^2 \mid TAUT(\phi(\mathcal{M}) \Rightarrow (v \Rightarrow v'))\}$$

$$completeness(\mathcal{G}, \mathcal{M}) = \begin{cases} 1 & if\ |\mathcal{E}_S^*| = 0 \\ \dfrac{|\mathcal{E}^S(\mathcal{G})|}{|\mathcal{E}_S^*|} & \text{otherwise} \end{cases}$$

The function *completeness* returns a rational value between 0 and 1, where a value of 1 means that the MIG is complete and a value of 0 means that all strong edges that could be inferred from the feature model are missing. A complete MIG means that there is no implicit strong edge that is not explicitly contained in the set of strong edges. Note that, if there are no strong edges that could be inferred from a given feature model, we consider the corresponding MIG as complete.

Minimality While the property of completeness is an indicator of whether the set of strong edges is optimized, the property of minimality is an indicator for an optimized set of weak edges. We define a modal implication graph to be *minimal* if each of its weak edges has a corresponding clause that is implied by the feature model formula (i.e., $\mathcal{E}^W(\mathcal{G}) = \{(v, v') \in \mathcal{V}(\mathcal{G})^2 \mid \exists d : d = (\neg v \vee v' \vee v'' \vee \ldots) \wedge TAUT(\phi(\mathcal{M}) \Rightarrow (\bigvee(d)))\}$). For examples, the MIG in Figure 3.1 is minimal. The weak edge $(\neg HFS, EXT)$ can be derived from the clause $d = HFS \vee EXT \vee NTFS$ and d is implied by the feature model formula (i.e., $\phi(\mathcal{M}) \Rightarrow (\bigvee(d))$). This is true for every weak edge in the MIG. Contrary, if the MIG would contain a weak edge $(\neg EXT, Log)$, there would exists no clause $d' = \{EXT, Log, ...\}$ for which $TAUT(\phi(\mathcal{M}) \Rightarrow (\bigvee(d')))$ is *true*, and thus the MIG would not be minimal. Analogous to the degree of completeness, we define the degree of minimality of a MIG as the ratio between all pairs of literals for which a clause exists that is implied by the feature model formula and the size of its set of weak edges:

$$\mathcal{E}_W^* = \{(v, v') \in \mathcal{V}(\mathcal{G})^2 \mid \exists d : d = (\neg v \vee v' \vee v'' \vee \ldots) \wedge TAUT(\phi(\mathcal{M}) \Rightarrow (\bigvee(d)))\}$$

$$minimality(\mathcal{G}, \mathcal{M}) = \begin{cases} 1 & if\ |\mathcal{E}^W(\mathcal{G})| = 0 \\ \dfrac{|\mathcal{E}_W^*|}{|\mathcal{E}^W(\mathcal{G})|} & \text{otherwise} \end{cases}$$

The function *minimality* returns a rational value between 0 and 1, where a value of 1 means that the MIG is minimal and a value less than 1 means that there are unnecessary weak edges in the MIG that could be removed or replaced with a strong edge. A value of 0 means that a MIG contains weak edges, but only needs to contain

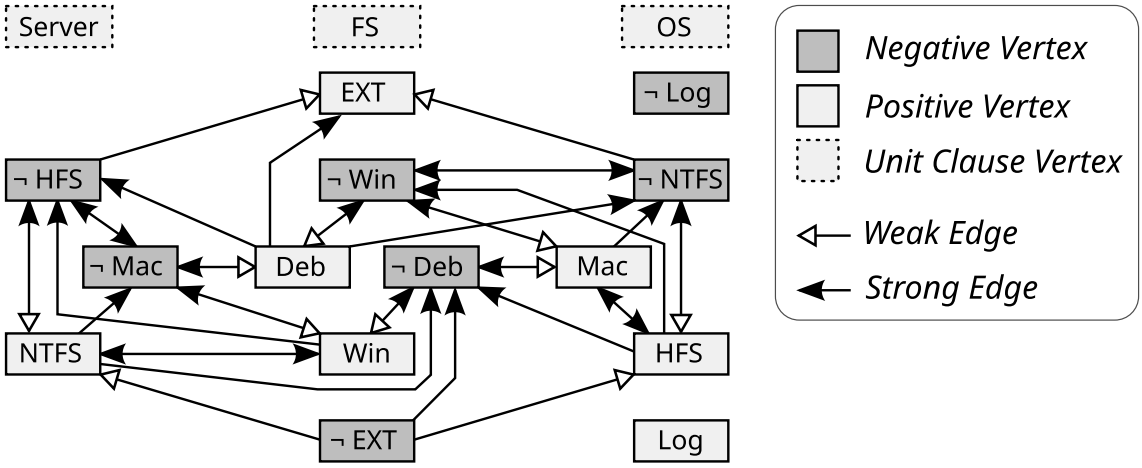

Figure 3.2: *Complete* MIG for the *Server* feature model from Figure 2.1 (on Page 9)

strong edges. In this case, a MIG could be expressed as a regular implication graph. Note that, if a MIG contains no weak edges at all, we consider it to be minimal.

In Figure 3.2, we depict a complete and minimal version of the MIG in Figure 3.1. We can see that this MIG has numerous transitive strong edges, which were not present in the previous MIG. For instance, the edge $(HFS, \neg Deb)$ is a transitive strong edge that can be derived from the strong path $((HFS, Mac), (Mac, \neg Deb))$. The new MIG also contains the two implicit strong edges $(\neg EXT, \neg Deb)$ and (Deb, EXT). In addition, there is no weak edge in the new MIG that could be removed without affecting its correctness with regard to its corresponding feature model.

Both, the degree of completeness and the degree of minimality, serve as an indicator for the effectiveness of a MIG in its applications. The smaller the degree of completeness and minimality of a MIG the lower its effectiveness. If a MIG is both, complete and minimal, its potential effectiveness is maximized.

3.3 Applications of Modal Implication Graphs

In our work, we consider multiple applications for MIGs. Originally, we designed MIGs to facilitated the process of *decision propagation* in an interactive configuration process. However, the information on feature relationships that are exposed by MIGs are beneficial for other applications as well. For instance, we can utilize MIGs for configuration sampling in an automated configuration process. Furthermore, we can use MIGs for some feature model analyses, such as finding atomic sets and operations that modify a feature model (e.g., feature model decomposition and feature model slicing). In this section, we focus on the originally intended usage of MIG to facilitated decision prorogation. We present some more details on the utilization of MIGs within these operations in the next chapters (Chapters 4–7).

Algorithm 3.1 Naïve decision-propagation algorithm

Require:
$\mathcal{M}$ – Feature Model
$c_{current}$ – Current Configuration
Return:
c_{new} – New Configuration

```
1:  function DECISIONPROPAGATION(M, c_current)
2:      c_solution ← SAT( M, c_current )
3:      if c_solution = ∅ then
4:          return ∅
5:      c_new ← c_current
6:      for all l_test ∈ L(M) do
7:          c_temp ← c_new ∪ {l_test}
8:          c_solution ← SAT( M, c_temp )
9:          if c_solution = ∅ then
10:             c_new ← c_new ∪ {¬l_test}
11:     return c_new
```

3.3.1 Decision Propagation

In this thesis, we employ decision propagation (cf. Section 2.3.2) in an interactive configuration process. An interactive configuration process consists of a sequence of decisions made by user. Starting with an empty configuration, each decision assigns a value to a feature variable, which leads to a partial and eventually to a complete configuration. Thus, each decision further restricts the current valid problem space until only one valid configuration remains.

For example, we take an empty configuration $c = \emptyset$ for the server system in Figure 2.1 and make the decision to assign the value *true* to the feature variable Win, which results in the partial configuration $c' = \{Win\}$. We then apply decision propagation, which computes the following new partial configuration $c'' = \{Win, Server, FS, OS, NTFS, \neg HFS, \neg Mac, \neg Deb\}$. The literals $Server$, FS, and OS are included, because the corresponding feature variables are core. The literal $NTFS$ is implied by the inclusion of Win through the cross-tree constraint $Win \Leftrightarrow NTFS$. The inclusion of the literals $\neg Deb$ and $\neg Mac$ are a result of the alternative group with Win. The literal $\neg HFS$ is included, because $\neg Mac$ is included and the cross-tree constraint $Mac \Leftrightarrow HFS$ must be satisfied.

Naïve SAT-Based Decision Prorogation

In general, decision propagation can be reduced to multiple SAT problems, similar to determining dead and core features. If we want to determine whether a given feature variable f can still be assigned the value *true* under a partial configuration c, we solve the SAT problem $SAT(\phi(\mathcal{M}) \wedge \bigwedge(c) \wedge \phi^{+}(f))$. Analogous, for the value *false*, we solve the problem $SAT(\phi(\mathcal{M}) \wedge \bigwedge(c) \wedge \phi^{-}(f))$. In Algorithm 3.1, we show a naïve algorithm that implements decision propagation by applying this principle. The algorithm takes two arguments, a feature model $\mathcal{M}$ and a (partial) configuration

Algorithm 3.2 Advanced decision-propagation algorithm

Require:
$\mathcal{M}$ – Feature Model
$c_{current}$ – Current Configuration
Return:
c_{new} – New Configuration

```
1:  function DECISIONPROPAGATION(M, c_current)
2:      c_solution ← SAT( M, c_current )
3:      if c_solution = ∅ then
4:          return ∅
5:      L_unknown ← {l ∈ L(M) | l ∉ c_solution ∧ ¬l ∉ c_current}
6:      c_new ← c_current
7:      for all l_test ∈ L_unknown do
8:          c_temp ← c_new ∪ {l_test}
9:          c_solution ← SAT( M, c_temp )
10:         if c_solution = ∅ then
11:             c_new ← c_new ∪ {¬l_test}
12:         else
13:             L_unknown ← L_unknown \ c_solution
14:     return c_new
```

$c_{current}$ for $\mathcal{M}$ and then computes a new (partial) configuration c_{new} that is an extension of $c_{current}$ containing all implicitly implied literals. As a first step, the algorithm calls the function `SAT`, which attempts to find a satisfying assignment for $\mathcal{M}$ under the assumption of all assignments defined by $c_{current}$ (Line 2). If the function returns no satisfying assignment, then $c_{current}$ is already invalid and decision propagation cannot be applied (Lines 3–4). Otherwise, the algorithm continues by creating an initial version of c_{new} by copying all literals from $c_{current}$ (Line 5). In the next step, the algorithm checks for every possible literal whether it conflicts with $c_{current}$ or not. For each literal in $\mathcal{L}(\mathcal{M})$, the algorithm constructs a new temporary configuration c_{temp} by adding l_{test} to c_{new} (Line 7). Then, the algorithm calls `SAT` to find a satisfying assignment of $\mathcal{M}$ under the assumption of c_{temp} (Line 8). If there is no satisfying assignment, then l_{test} conflicts with c_{new} and consequently with $c_{current}$. As we use Boolean features, we know that if l_{test} is conflicting with $c_{current}$, then its complement $\neg l_{test}$ must be implied by $c_{current}$ (Lines 9–10). Therefore, the algorithm adds $\neg l_{test}$ to c_{new}. In case that `SAT` finds a satisfying assignment $c_{solution}$, we know that l_{test} does not conflict with $c_{current}$, but we cannot infer that it is also implied by $c_{current}$. Finally, after all literals in $\mathcal{L}_(\mathcal{M})$ were processed, the algorithm returns c_{new}, which now contains all implied literals (Line 11).

Advanced SAT-Based Decision Prorogation

The naïve algorithm uses two SAT queries for every feature variable, and thus requires a high amount of computational effort. In Algorithm 3.2, we show an advanced version of this algorithm, which uses the solution of any solved SAT query to drastically reduce the total amount of SAT queries. The general structure of the

algorithm is the same as before, it also takes two arguments, a feature model $\mathcal{M}$ and a (partial) configuration $c_{current}$ and computes a new (partial) configuration c_{new}. Again, the first step of the algorithm is to call the function `SAT` for $\mathcal{M}$ under the assumption of $c_{current}$ (Line 2). There are two possibilities, either there is no or at least one satisfying assignment. If the function returns no assignment (i.e., $\emptyset$), then $c_{current}$ is invalid and the algorithm stops (Lines 3–4). Otherwise, the function returns a satisfying assignment as a set of literals $c_{solution}$, which is then used in the following steps. Next, the algorithm constructs a set $\mathcal{L}_{unknown}$ that contains all literals for which *it is unclear* whether they conflict with $c_{current}$ (Line 5). As we know that $c_{solution}$ is satisfiable and that $c_{current} \subseteq c_{solution}$, we know that all literals in $c_{solution}$ are definitely not conflicting with $c_{current}$. Analogous, we know that the complements of all literals in $c_{current}$ definitely do conflict with $c_{current}$. For all other literals it is not immediately clear whether they have a conflict with $c_{current}$, Consequently, $\mathcal{L}_{unknown}$ contains all literals from $\mathcal{L}_(\mathcal{M})$ minus all literals of $c_{solution}$ and minus all complementary literals from $c_{current}$. In the next step, the algorithm creates the initial version of c_{new} by copying all literals from $c_{current}$ (Line 6). Then, the algorithm tests each literals from $\mathcal{L}_{unknown}$ individually and within each iteration may modify c_{new} and $\mathcal{L}_{unknown}$ (Line 7). Again, the algorithm constructs a new temporary configuration c_{temp} by adding l_{test} to c_{new} (Line 8) and uses this temporary configuration in a SAT query (Line 9). If `SAT` cannot find a satisfying assignment then l_{test} conflicts with c_{new} and consequently with $c_{current}$. The algorithm adds $\neg l_{test}$ to c_{new} (Lines 10–11). In case that `SAT` does find a satisfying assignment $c_{solution}$, we know that l_{test} does not conflict with $c_{current}$. Furthermore, we can use $c_{solution}$ to reduce the set $\mathcal{L}_{unknown}$. Because $c_{solution}$ is a satisfying assignment with $c_{current} \subseteq c_{solution}$, every assignment in $c_{solution}$ is definitely not conflicting with $c_{current}$ and we can remove them from $\mathcal{L}_{unknown}$ without further testing (Lines 12–13). Finally, after all literals in $\mathcal{L}_{unknown}$ were processed, the algorithm returns the modified c_{new} (Line 14).

The algorithm described above attempts to reduce the amount of SAT instance that are needed to solve during its execution. However, applying this algorithm after each decision still requires much computational effort, because it still requires solving multiple NP-complete problems. By employing a MIG, we want to improve on this algorithm by reducing the total amount of required SAT instances even further. The basic idea of a MIG-assisted algorithm is to traverse along the strong and weak paths of the MIG in order to determine directly implied and potentially implied literals. One improvement is to avoid SAT instances that test literals that are obviously implied by another literal (i.e., are connected by a strong path). Another improvement is to avoid SAT instances that test literals that are independent of the current decision (i.e., no reachable by any path). We can easily identify these cases using a MIG, and thus potentially reduce the amount of SAT queries necessary for the a complete decision propagation. In our scenario of an iterative configuration process, a decision is the assignment of a value to a single feature variable. We can map every decision to a literal that we add to the current configuration. Further, we know that there is a vertex in a MIG that represents this literal. Starting from this vertex, we can traverse the graph using a standard graph searching algorithm (e.g., breadth-first search) to find all strong paths to other vertices. This way, we find all literals that are directly implied by the current decision without employing any

Algorithm 3.3 MIG-assisted decision propagation algorithm

Require:
$\mathcal{M}$ – feature model
$\mathcal{G}$ – modal implication graph
$c_{current}$ – current configuration
l_{new} – literal of most recent decision
Return:
c_{new} – new configuration

1: **function** DECISIONPROPAGATION($\mathcal{M}$, $c_{current}$)
2: $\mathcal{G} \leftarrow$ UPDATEGRAPH($\mathcal{G}$, $c_{current}$)
3: $\mathcal{L}_{strong} \leftarrow \{l \in \mathcal{L}(\mathcal{M}) \mid \exists(l_{test}, l) \in S\}$
4: $c_{new} \leftarrow c_{current} \cup \mathcal{L}_{strong}$
5: $\mathcal{L}_{unknown} \leftarrow \{l \in \mathcal{L}(\mathcal{M}) \mid \neg l \notin c_{current} \ \wedge \ \exists(l_{new}, l) \in W\}$
6: **if** $\mathcal{L}_{unknown} \neq \emptyset$ **then**
7: $c_{solution} \leftarrow$ SAT($\mathcal{M}$, $c_{current}$)
8: **if** $c_{solution} = \emptyset$ **then**
9: **return** $\emptyset$
10: $\mathcal{L}_{unknown} \leftarrow \mathcal{L}_{unknown} \setminus \{l \mid \neg l \in c_{solution}\}$
11: **for all** $l_{test} \in \mathcal{L}_{unknown}$ **do**
12: $c_{temp} \leftarrow c_{new} \cup \{\neg l_{test}\}$
13: $c_{solution} \leftarrow$ SAT($\mathcal{M}$, c_{temp})
14: **if** $c_{solution} = \emptyset$ **then**
15: $\mathcal{G} \leftarrow$ UPDATEGRAPH($\mathcal{G}$, c_{temp})
16: $\mathcal{L}_{strong} \leftarrow \{l \in L \mid \exists(l_{test}, l) \in S\}$
17: $c_{new} \leftarrow c_{new} \cup \{\neg l_{test}\} \cup \mathcal{L}_{strong}$
18: $\mathcal{L}_{unknown} \leftarrow \mathcal{L}_{unknown} \setminus \mathcal{L}_{strong}$
19: **else**
20: $\mathcal{L}_{unknown} \leftarrow \mathcal{L}_{unknown} \setminus c_{solution}$
21: **return** c_{new}

SAT instances. In addition, we can perform another graph traversal along all weak paths to find all vertices that are *potentially* implied by the current decision. The literals represented by these vertices we have to check using SAT instances, however every other literal can be ignored, because it is not reachable via a strong or weak path, and thus independent from the current decision. Effectively, we split the set of all literals that are not part of the current configuration into three disjoint subsets. First, the set of literals that are represented by vertices that are reachable via a strong path. Second, the set of literals that are represented by vertices that are only reachable via a weak path. Third, the set of literals that are represented by vertices that are not reachable. The first set we can infer directly without using any SAT instances. The third we can ignored as they have no relationship with the current selection. Only for the second set we must resort to employing SAT instances.

MIG-Assisted Decision Prorogation

We depict our MIG-assisted decision propagation algorithm in Algorithm 3.3. Again, the general structure of this algorithm is similar to the naïve and advanced algorithm.

However, it requires two additional parameters, a MIG $\mathcal{G}$ and a literal l_{new} that represents the latest decision in the configuration process (i.e., the most recent variable assignment). The first step of the algorithm is to update $\mathcal{G}$ by checking whether the condition of any weak edge is fulfilled, which is then treated as a strong edge (Line 2). This can happen because we assign concrete values to feature variables during the configuration process and also during the decision propagation. Therefore, if the condition of a weak edge is fulfilled by the current partial configuration, the edge no longer indicates a potential but a direct implication (i.e., becomes strong). In the second step, the algorithm traverses $\mathcal{G}$ along all strong paths, starting from the vertex of l_{new}, to find the set of all literals $\mathcal{L}_{strong}$ that are directly implied by the current decision (Line 3). The algorithm then constructs c_{new} by copying all literals from $c_{current}$ and adding all literals from $\mathcal{L}_{strong}$ (Line 4). Next, the algorithm traverses $\mathcal{G}$ along all weak edges, starting from the vertex of l_{new}, to determine the set of all potentially implied literals $\mathcal{L}_{unknown}$ that are not already contained in c_{new} (Line 5). In case that $\mathcal{L}_{unknown}$ is empty, the algorithm has nothing more to do and simply returns c_{new} (Lines 6, 21). Otherwise, the algorithm continues by performing an initial SAT query on $\mathcal{M}$ under the assumption of c_{new} (Line 7). If this SAT query cannot find a satisfying assignment, $c_{current}$ was already conflicting and the algorithm stops by returning an empty set (Lines 8–9). Otherwise, the algorithm uses the resulting satisfying assignment $c_{solution}$ to reduce $\mathcal{L}_{unknown}$. Every literal in $c_{solution}$ is not conflicting with $c_{current}$, because $c_{current} \subseteq c_{solution}$. Therefore, the complement of each literal in $c_{solution}$ cannot be implied by $c_{current}$. Consequently, the algorithm removes the complement of each literal in $c_{solution}$ from $\mathcal{L}_{unknown}$ (Line 10). Then, the algorithm iterates over every remaining literal l_{test} in $\mathcal{L}_{unknown}$ to test each individually as follows. The algorithm creates a temporary configuration c_{temp} that is a union of c_{new} and the complement of l_{test} (Line 12). Next, the algorithm calls `SAT` with c_{temp} (Line 13). If there is a satisfying assignment $c_{solution}$, then the algorithm uses it again to reduce $\mathcal{L}_{unknown}$ by removing the complement of every literal in $c_{solution}$ from $\mathcal{L}_{unknown}$ (Line 20). In this case, also l_{test} is not implied by $c_{current}$, because its complement does not conflict with $c_{current}$. In contrast, if `SAT` cannot find a satisfying assignment, the complement of l_{test} does conflict with $c_{current}$, and thus l_{test} is implied by $c_{current}$. The algorithm updates $\mathcal{G}$ using c_{temp} (Line 15). Afterwards, it traverses the updated MIG along its strong edges, starting from the vertex representing l_{test}. The set of all strongly connected literals $\mathcal{L}_{strong}$ are implied by l_{test}, and thus also by $c_{current}$. The algorithm adds l_{test} and all literals in $\mathcal{L}_{strong}$ to c_{new} (Line 17) and removes all literals in $\mathcal{L}_{strong}$ from $\mathcal{L}_{unknown}$ (Line 18). Finally, after the algorithm processed all literals of $\mathcal{L}_{unknown}$, it returns c_{new} (Line 21).

With the MIG-assisted algorithm, we can enabled the utilization of MIGs in decision propagation. We expect that the usage of a MIG speeds up the decision propagation compared to the naïve and advanced SAT-based algorithms.

3.3.2 Feature Model Analysis

Another application for MIGs is to increase the performance of certain feature model analyses. In particular, analyzing the set of strong and weak paths can reveal some information about the relationship between two features, which is useful for any analysis that relies on these information. This makes it especially helpful for analyses

such as *detecting void feature models*, *detecting unsatisfiable configurations*, and *finding atomic sets*.

Detecting Void Feature Models Technically, we cannot create a MIG for a void feature model, because the resulting MIG would contain one or more contradictions. In addition, such a MIG would not be useful, as it cannot describe relationships between literals, if the underlying feature model formula is not satisfiable However, for detecting whether a feature model is void, we can attempt to create a MIG and during its construction check whether we include any contradictions. If we add a strong edge $e = (v, \neg v)$ the graph contains a contradiction (i.e., $v \Rightarrow \neg v$) and therefore the feature model must be void. If we added all strong edges without detecting such a contradiction, we can traverse the MIG along its strong edges to find contradicting strong paths. Any strong path where the source vertex represents the complement of the literal of the target vertex also represents a contradiction, and thus a void feature model. While this technique works in theory, its practical applications are limited. The existence of a contradictory strong edge or path is a sufficient, but not a necessary condition. A contradiction within a feature model may also be caused by a combination of complex clauses.

Detecting Unsatisfiable Configurations A more useful technique than detecting a void feature model is the related analysis of checking whether a given partial configuration is satisfiable. This technique works with the same principle as detecting void feature models by checking whether there exists a contradiction within the MIG. Given a partial configuration c for a feature model $\mathcal{M}$ and a MIG $\mathcal{G}$ for $\mathcal{M}$, we first update $\mathcal{G}$ as we do in our MIG-assisted decision propagation algorithm (cf. Section 3.3.1). The assigned values in c may satisfy the conditions of some weak edges in $\mathcal{G}$, and thus the updated MIG $\mathcal{G}'$ may contain more strong edges. We then traverse along the strong edges of $\mathcal{G}'$ to find any strong path with a source vertex v and a target vertex $\neg v$. If we find any such path, the conjunction of the feature model $\mathcal{M}$ and the partial configuration c contains a contradiction. Thus, assuming that $\mathcal{M}$ itself is satisfiable, c must be unsatisfiable. Similar to the detecting void feature model, the existence of such an edge or path is a sufficient, but not a necessary condition. A configuration can be unsatisfiable without the existence a contradictory strong edge or path.

Finding Atomic Sets A more complex analysis is to find the atomic sets of a feature model (cf. Section 2.1.3). When using a SAT-based approach, this analysis requires much computational effort. However, whether two features are in the same atomic set can be efficiently determined in a *complete MIG* (cf. Section 3.2.3). To this end, we iterate over all strong edges of a complete MIG. If there exists a strong path from a vertex v to another vertex v' and also from v' to v, then we know that $v \Rightarrow v'$ and that $v' \Rightarrow v$, and consequently that $v \Leftrightarrow v'$. In this case, the corresponding features of v and v' must be in the same atomic set. Analogous, if there is no strong path from v to v' or from v' to v, then the corresponding features of v and v' must be in different atomic sets.

3.4 Construction of a Modal Implication Graph

A MIG can be constructed from any non-void feature model. As long as the feature model is not changed a corresponding MIG can be used indefinitely in any application. Thus, any computation effort to construct a MIG may be amortized over time, when it is frequently used. However, it is still desirable to have an efficient building process to construct a MIG from a feature model. In the following, we describe the details of the basic build process and a collection of optimizations that enables a faster build process at the cost of losing some effectiveness of the resulting MIG. To this end, we also explain the property of completeness and minimality for a MIG and show how this relates to a MIG's effectiveness and its required build time.

3.4.1 Basic Build Process

The basic build process of a MIG consists of two phases, *analyzing the input feature model* and *constructing vertices and edges*. In the first phase, we prepare the input feature model formula by analyzing it and removing certain anomalies. In particular, there are three possible anomalies that we need to identify, a *void feature model*, *core and dead features*, and *internally redundant clauses*. A void feature model does not have any valid configurations, and thus it is not reasonable to build a MIG for this model at all. As core and dead feature variables can only be assigned one value, it is not reasonable to create vertices for them, but to ignore them and only create vertices for configurable feature variables. Internal redundancies in clauses may increase the number of total clauses or the number of literals within a clause, which can affect the derivation of strong and weak edges later. Thus, we remove these redundancies to simplify the remaining build process. In the second phase, we derive the MIG graph structure from the modified feature model formula in two steps. First, we *derive the vertices* from the set feature variables. Second, we *derive the edges* from the clauses of the feature model formula. In the following, we describe the steps od all phases in detail.

Detecting Void Feature Model Formulas First, we analyze whether the feature model formula is void by performing the corresponding analysis (cf. Section 2.1.3). For a given feature model formula $\phi(\mathcal{M})$, we solve the SAT instance $SAT(\phi(\mathcal{M}))$. If we cannot find a satisfying assignment c_{void}, then the formula is void and we stop the build process. Otherwise, we proceed and use the found satisfying assignment c_{void} in the upcoming analysis for dead and core feature variables.

Detecting Core and Dead Feature Variables Second, we compute the set of core and dead feature variables using the a slightly modified version of the analysis we described in Section 2.1.3. The analysis detects core and dead feature variables by solving multiple SAT instances. A minor change to the standard analysis is that we can use the satisfying assignment c_{void} from the void analysis of the previous step. Thus, we can omit solving the initial SAT instance of the standard analysis. As a result, the analysis returns a set of literals that represent all core and dead feature variables. This set is equivalent to the set $\mathcal{D}^U$ of a MIG.

Detecting Internal Clause Redundancies Third, we detect internally redundant clauses and fix them accordingly. This step creates a new feature model formula ϕ', which is equivalent to the original formula $\phi(\mathcal{M})$, but without any internal redundancies. We detect internal redundancies for each clause in the feature model formula individually by iterating over the set of clauses. A clause is internally redundant if it contains a literal and its complement (cf. Section 2.1.3). In this case, we remove the entire clause from the feature model formula, because it is an obvious tautology. In addition, we also remove duplicate literals from each clause. After processing all clauses, we continue the build process with the new formula ϕ'.

Deriving Vertices Fourth, we derive the set of vertices $\mathcal{V}$. To this end, we iterate over all configurable feature variables from the feature model formula. For each configurable feature variable f, we add two vertices to $\mathcal{V}$, which each represents one of the two possible assignments of the feature variable (i.e., $\phi^+(f)$ and $\phi^-(f)$).

Deriving Edges Fifth, we derive the set of 2-clauses $\mathcal{D}^S$ and the set of complex clauses $\mathcal{D}^W$. To this end, we iterate over all clauses of the feature model formula. For each clause, we differentiate whether it contains one literals, two literals, or more than two literals. First, if a clause contains exactly one literal (i.e., a unit clause), the corresponding feature variable is either core if the literal is positive or dead if the literal is negative. Since we determined the set of core and dead features $\mathcal{D}^U$ in the previous step, we can ignore unit clauses in this step. Second, if a clause contains exactly two literals we add it to $\mathcal{D}^S$. Each 2-clause can be converted into an implication between two literals, which translates into a strong edge. In fact, every 2-clause $a \vee b$ is equivalent to two different implications, $\neg a \Rightarrow b$ and $\neg b \Rightarrow a$. Consequently, for each 2-clause $a \vee b$, we infer two strong edges $e_1 = (\neg a, b)$ and $e_2 = (\neg b, a)$. For example, in Figure 2.1, we add the clauses $\neg Win \vee NTFS$ to $\mathcal{D}^S$ and infer the two strong edges $(Win, NTFS)$ and $(\neg NTFS, \neg Win)$ (cf. Figure 3.1). Third, if a clause contains more than two literals we add it to the set of complex clauses $\mathcal{D}^W$. All weak edges can be inferred from $\mathcal{D}^W$. Storing complex clauses rather than weak edges directly has two advantages. First, as each clause translates two multiple weak edges, we reduce the storing of redundant information. Second, in addition to deriving weak edges themselves, we can also derive the condition under which they are *true*. For each clause in $\mathcal{D}^W$ with k unique literals that is not a tautology or contradiction, we can infer $k \cdot k - 1$ weak edges. Every clause $d = l_1 \vee \ldots \vee l_n$ can be written as $\neg l_i \Rightarrow \bigvee_{1 \leq j \leq n, j \neq i} l_j$ for each literal l_i in the clause. Thus, for each pair of literals within a clause $d = l_1 \vee \ldots \vee l_n$, two weak edges can be inferred, such that $e_{i,j} = (\neg l_i, l_j)$ and $e_{j,i} = (\neg l_j, l_i)$ for all $1 \leq i < j \leq n$. For instance, the clause $a \vee b \vee c \vee d$ translates into 12 weak edges. In Table 3.1, we list all weak edges that result from this clause, their respective condition, and an equivalent propositional formula that explains and visualized the weak edge and its condition. Regarding our example in Figure 2.1, for the clause $Deb \vee Win \vee Mac$, the following six weak edges are added: $e_{1,2} = (\neg Deb, Win)$, $e_{2,1} = (\neg Win, Deb)$, $e_{1,3} = (\neg Deb, Mac)$, $e_{3,1} = (\neg Mac, Deb)$, $e_{2,3} = (\neg Win, Mac)$, and $e_{3,2} = (\neg Mac, Win)$ (cf. Figure 3.1).

The basic build process does not optimize the constructed MIG. Thus, it is likely that this MIG is neither complete nor minimal. To achieve both properties, we introduce

Table 3.1: All 12 weak edges and their conditions derived from the clause $a \vee b \vee c \vee d$

Equivalent Formula	Condition	Weak Edge
$(\neg a \wedge \neg b) \Rightarrow (\neg c \Rightarrow d)$	$\neg a \wedge \neg b$	$(\neg c, d)$
$(\neg a \wedge \neg b) \Rightarrow (\neg d \Rightarrow c)$	$\neg a \wedge \neg b$	$(\neg d, c)$
$(\neg a \wedge \neg c) \Rightarrow (\neg b \Rightarrow d)$	$\neg a \wedge \neg c$	$(\neg b, d)$
$(\neg a \wedge \neg c) \Rightarrow (\neg d \Rightarrow b)$	$\neg a \wedge \neg c$	$(\neg d, b)$
$(\neg a \wedge \neg d) \Rightarrow (\neg b \Rightarrow c)$	$\neg a \wedge \neg d$	$(\neg b, c)$
$(\neg a \wedge \neg d) \Rightarrow (\neg c \Rightarrow b)$	$\neg a \wedge \neg d$	$(\neg c, b)$
$(\neg b \wedge \neg c) \Rightarrow (\neg a \Rightarrow d)$	$\neg b \wedge \neg c$	$(\neg a, d)$
$(\neg b \wedge \neg c) \Rightarrow (\neg d \Rightarrow a)$	$\neg b \wedge \neg c$	$(\neg d, a)$
$(\neg b \wedge \neg d) \Rightarrow (\neg a \Rightarrow c)$	$\neg b \wedge \neg d$	$(\neg a, c)$
$(\neg b \wedge \neg d) \Rightarrow (\neg c \Rightarrow a)$	$\neg b \wedge \neg d$	$(\neg c, a)$
$(\neg c \wedge \neg d) \Rightarrow (\neg a \Rightarrow b)$	$\neg c \wedge \neg d$	$(\neg a, b)$
$(\neg c \wedge \neg d) \Rightarrow (\neg b \Rightarrow a)$	$\neg c \wedge \neg d$	$(\neg b, a)$

an advanced build process that extends the basic build process by three steps with the goal to increase the effectiveness of the MIG.

3.4.2 Advanced Build Process

The goal of the advanced build process is to optimize the sets $\mathcal{D}^S(\mathcal{G})$ and $\mathcal{D}^W(\mathcal{G})$, such that the set of weak edges is minimized and the set of strong edges is complete. To this end, the advanced build process extends the basic build process by adding three more steps: *detecting external clause redundancies*, *finding implicit strong edges*, and *building the strong hull*. We add the first additional step at the end of the first phase of the basic build process (i.e., *analyzing the input feature model*). It further analyzes the input feature model formula and removes an additional type of anomalies (i.e., externally redundant clauses). The other two steps we add at the end of the second phase of the basic build process (i.e., *constructing vertices and edges*). Both steps refine the initial structure of the graph by modifying the sets of weak and strong edges. It is important to note that, none of these three additional steps modifies the sets $\mathcal{D}^S(\mathcal{G})$ and $\mathcal{D}^W(\mathcal{G})$ in such a way that the correctness of the MIG is affected. In the following, we present each step in more detail.

Detecting External Clause Redundancies In this step, we detect externally redundant clauses within the formula (cf. Section 2.1.3) resulting from the first three steps of the basic building process. The goal of this step is to increases the degree of minimality of the resulting MIG by removing redundant clauses, which would lead to unnecessary weak edges. To this end, we construct a new feature model formula by starting with an empty formula ϕ'' and then iteratively adding each clause of the current feature model formula ϕ' one-by-one. Before we add a clause d to ϕ'', we first check whether it is already implied by ϕ'' (i.e., whether $SAT(\phi'' \wedge \neg(\bigvee(d)))$ is *false*). If d is not implied by ϕ'', it is not redundant and we add it to ϕ''. Otherwise d is not added to ϕ''. We then continue the build process using the clauses from the newly

constructed formula ϕ''. While this step effectively removes any redundancies in the set of dependencies, it also computationally expensive, because for each clause we check, we must solve a SAT instance.

Finding Implicit Strong Edges After we built the initial graph structure of a MIG in the basic build process, there still can be implicit strong edges that are not contained explicitly in the MIG. We show this in our example in Section 3.2.3. In this step, we detect implicit strong edges and add them to the MIG. Therefore, we increases the degree of completeness of the MIG with this step.

To find implicit strong edges, we employ a breadth-first search that investigates all pairs of weakly connected vertices. We start the search with an arbitrary vertex a and consider each vertex b that is weakly connected to it. For each vertex b, we then solve the SAT instance $SAT(\phi(\mathcal{M}) \Rightarrow (a \Rightarrow b))$ to determine whether a implies b. In that case, we add a strong edge from a to b and continue the search. We repeat the search with all possible starting vertices until we checked every pair of weakly connected vertices.

Building the Strong Hull As a last step, we build the strong hull of the MIG. This means that we compute all transitive strong edges and add them to the existing set of strong edges. By doing this, we facilitate traversing along strong paths and also increase the degree of completeness of the MIG. In fact, if we performed the previous step and added all implicit strong edges, after building the strong hull, the MIG is now guaranteed to be complete. Finding all transitive strong edges can be done efficiently by traversing the graph along its strong edges. For each vertex in the graph, a breadth-first search finds all other vertices that are reachable via a strong path. If there exists such a path, but no direct strong edge from the start to the end vertex (i.e., a transitive strong edge), this edge is added to the graph.

3.5 Incremental Modal Implication Graphs

While a once built MIG can be reused an unlimited number of times in any feature model analysis and configuration processes, it must be specifically tailored to encode the configuration logic of a particular feature model, which can entail significant computational cost. This can be an issues when a feature model, for which we built a MIG, evolves. Feature model evolution may change a feature model or its constraints and, subsequently, invalidate an existing MIG that then represents outdated configuration logic. For feature models that evolve frequently, it is costly to perform the advanced build process to fully rebuild a complete MIG after each evolution step. Thus, in the light of frequent feature model evolution, reusing the benefits of a MIG for interactive configuring or sampling configurations is, currently, severely hampered if not outright infeasible. In addition to the basic and advanced build process, we present an incremental build process to create a MIGs for an evolving feature model. The core idea of the incremental build is to reuse information on anomalies from an older version of the MIG and the information on the feature model change to reduce the number of SAT instances. To this end, we first explain how we *compute the feature model change* and how it is helpful in speeding up

certain analysis during the build process. Then, we reason about which are the most *computationally expensive building steps* in the advanced build process and what potential options we have to increase their performance in the incremental build process. Finally, we describe the *modified building steps* in the incremental build process in more detail and reason about how these modifications affect the MIG properties of completeness and minimality.

3.5.1 Computing the Feature Model Change

For the incremental build process, we first infer the change Δ between the two feature models $\mathcal{M}$ (i.e., before the evolution) and $\mathcal{M}'$ (i.e., the current version).

Definition 3.3 Feature Model Change**.** *The feature model change between two feature models $\mathcal{M}$ and $\mathcal{M}'$ is described by a tuple $\Delta(\mathcal{M}, \mathcal{M}') = (\mathcal{F}^+, \mathcal{F}^-, \mathcal{D}^+, \mathcal{D}^-)$ consisting of four sets.*

- *The set of added feature variables $\mathcal{F}^+ = \mathcal{F}(\mathcal{M}') \setminus \mathcal{F}(\mathcal{M})$ contains all features that are only contained in $\mathcal{M}'$, but not in $\mathcal{M}$. Compared to $\mathcal{M}$ these features are newly introduced in $\mathcal{M}'$.*
- *The set of removed feature variables $\mathcal{F}^- = \mathcal{F}(\mathcal{M}) \setminus \mathcal{F}(\mathcal{M}')$ contains all features that are only contained in $\mathcal{M}$, but not in $\mathcal{M}'$. Compared to $\mathcal{M}$ these features have been removed in $\mathcal{M}'$.*
- *The set of added dependencies $\mathcal{D}^- = \mathcal{D}(\mathcal{M}) \setminus \mathcal{D}(\mathcal{M}')$ contains all dependencies that are only contained in $\mathcal{M}'$, but not in $\mathcal{M}$.*
- *The set of removed dependencies $\mathcal{D}^+ = \mathcal{D}(\mathcal{M}') \setminus \mathcal{D}(\mathcal{M})$ contains all dependencies that are only contained in $\mathcal{M}$, but not in $\mathcal{M}'$.*

The change $\Delta(\mathcal{M}, \mathcal{M}')$ between two feature models can be computed by comparing their sets of features and dependencies, respectively. In our work, we compute the sets of changed feature variables $\mathcal{F}^+$ and $\mathcal{F}^-$ by comparing the feature names in $\mathcal{F}(\mathcal{M})$ and $\mathcal{F}(\mathcal{M}')$. We infer the set of removed clauses and added clauses by computing $\mathcal{D}^- = \mathcal{D}(\mathcal{M}) \setminus \mathcal{D}(\mathcal{M}')$ and $\mathcal{D}^+ = \mathcal{D}(\mathcal{M}') \setminus \mathcal{D}(\mathcal{M})$. For this, we take into account renamings of feature variables, if this information is available. If a feature variable was renamed as part of the feature model evolution, we treat it as the same variable as before, instead of considering it to be a removed and an added variable. Similarly, if a clause contains one or more renamed variables, but is otherwise unchanged, we treat it as the same clause.

From $\Delta(\mathcal{M}, \mathcal{M}')$, we can reason about the type of change that is necessary to update the MIG. In particular, we characterize $\Delta(\mathcal{M}, \mathcal{M}')$ depending on the contents of the sets of changed dependencies $\mathcal{D}^-$ and $\mathcal{D}^+$. A feature model evolution can either *do no change* (i.e., $\mathcal{D}^- = \mathcal{D}^+ = \emptyset$), *add constraints* (i.e., $\mathcal{D}^- = \emptyset$ and $\mathcal{D}^+ \neq \emptyset$), *remove constraints* (i.e., $\mathcal{D}^- \neq \emptyset$ and $\mathcal{D}^+ = \emptyset$), or *replace constraints* (i.e., $\mathcal{D}^- \neq \emptyset$ and $\mathcal{D}^+ \neq \emptyset$). For the characterization, we do not consider the sets $\mathcal{F}^+$ and $\mathcal{F}^-$, as changing the set of features does not change the relationships between existing features. Naturally, adding or removing features to a specific feature model

representation, such as a feature diagram, also changes related constraints. However this is then also reflected in the sets of changed dependencies $\mathcal{D}^-$ and $\mathcal{D}^+$. When the set of features changes, it suffices to update the MIG by adding and removing the respective vertices, which is part of the second building step *deriving vertices and edges*. Adding constraints to $\mathcal{D}(\mathcal{M})$ may cause new anomalies within a feature model, but will not remove existing ones. Removing constraints from $\mathcal{D}(\mathcal{M})$ may fix old anomalies, but will never cause new ones. Replacing constraints can be seen as simultaneous addition and removal of constraints to a feature model, and thus may introduce *and* fix anomalies.

In addition to the feature model change $\Delta(\mathcal{M}, \mathcal{M}')$, we also require information on anomalies of the previous feature model $\mathcal{M}$. We determine the set of anomalies from the feature model formula and MIG of $\mathcal{M}$. Core and dead feature variables, as well as implicit strong edges, are saved within a MIG, and, thus, we can access them without further computation. We derive the set of previously externally redundant clauses by comparing the set of clauses in the CNF with the edges in the MIG. If there is a clause with no corresponding edge, it was redundant.

3.5.2 Computationally Expensive Building Steps

The advanced build process consists of many single building steps. However, only certain steps of are computationally demanding. In fact, all building steps that rely on solving SAT instances (i.e., the detection of *void feature model*, *core and dead features*, *external clause redundancies*, and *implicit strong edges*) are among the most time-consuming building steps, as these analyses solve (multiple) NP-complete problems. A notable outlier within these steps is the detection of void feature models, as this analysis requires only a single SAT query, which is relatively fast for most feature models [Mendonça et al., 2009b]. Thus, we focus on the other three building steps. In the following, we discuss the potential options to improve the efficiency of these steps either directly or with the help of the additional information provided by the feature model change. All three steps are similar in nature, as they all analyze employ multiple similar SAT instances. This allows us to reason about potential efficiency improvements of these three steps together. In summary, we see two possible improvements: *reducing* the amount of *SAT instances* and entirely *skipping the build step*.

Reducing SAT Queries One way to speed up a building step is by reducing the number of overall SAT instances as these are the most expensive parts. Each SAT instances in every step can be mapped to exactly one particular anomaly (i.e., a core feature, dead feature, redundant clause, or implicit strong edge). If we knew or could estimate these anomalies in advance (e.g., whether a feature is core or a clause redundant), we could avoid the respective SAT queries. Regarding the additional information from the incremental build, we can think of two ways of achieving this. First, we can reason about whether and how the set of anomalies for the previous feature model has changed. Second, given the feature model change, we can use heuristics to estimate whether a new anomaly will occur.

Given a feature model and corresponding MIG of the previous version, we can derive its core and dead features, redundant clauses, and implicit strong edges. Then, by analyzing the feature model change, we can determine whether these anomalies will change. Adding constraints to the feature model can add more anomalies, but does not remove existing anomalies. This includes new core and dead features, redundant clauses, and implicit strong edges. In contrast, removing constraints may only remove existing anomalies, but never adds new ones.

Instead of investigating all features and constraints for anomalies, by using a heuristic, we can test only the ones that are most likely to be affected by a change (e.g., by considering only features that appear in added or removed constraints). Testing only a subset of features and constraints decreases the number of SAT instances. However, using heuristics introduces the risk of decreasing completeness of a computed MIG, i.e., that the graph contains fewer strong edges than possible and more weak edges than necessary. In turn, this may decrease the MIGs effectiveness in terms of facilitating decision propagation.

Skipping Operations Another options is to skip an entire operation during the build process. Similar to using heuristics, this can also reduce the completeness and minimality of the MIG. Based on the usage scenario, it may or may not be reasonable to skip operations. For instance, if a MIG (version) is created only once and it is used for many thousands of configuration processes, the additional effort to not skip the operations may pay off in the long term. In contrast, if a MIG (version) is only used for few configuration processes, it pays off to skip them. With the basic build process, we already have the option to partially skip building steps compared to the advanced build process, which may result in an incomplete and non-minimal MIG, which can be less effective during its application. Thus, we resort to skipping operations in the incremental build process as well.

3.5.3 Modified Building Steps

To apply an incremental build process instead of the basic or advanced one, there must be a MIG built for a feature model and a new version of this feature model (e.g., stemming from evolution). The incremental build process is based on the advanced build process and follows the same major structure, but uses a modified variant of the most time-consuming building steps. As the creation of the initial graph structure (i.e., adding vertices and edges) is a fast building step operation, we rebuild the graph from scratch. This simplifies the implementation compared to an implementation that needs to be able to add and remove edges and vertices to and from a graph. In the following, we describe only the modifications to three steps *detecting core and dead features*, *detecting external clause redundancies*, and *finding implicit strong edges*.

Detecting Core and Dead Features Detecting core and dead features, is based on solving multiple SAT instance. However, it is orders of magnitudes faster compared to the following two steps. This is due to the re-use of already found SAT solutions to substantially reduce the number of SAT queries [Mendonça, 2009]. For this reason, we make only small adaptions to this step for the incremental build

process. A central change is that we split the step into two parts: First, we check whether the core and dead features from the previous MIG are still core or dead. Second, we check whether any previously configurable features are now core or dead. This allows us to avoid unnecessary SAT instances, if constraints were only added or only removed. We execute the first part only if Δ is removing or replacing constraints and execute the second part only if Δ is adding or replacing constraints. If Δ makes no change to the constraints at all, we can skip both parts.

We do not consider using any heuristics for this operation or skipping it entirely, for two reasons. First, the operation is by far the fastest compared with the other operations that solve SAT instance. Second, an inaccurate result from this operation could severely harm the graph structure as it could add vertices that cannot be part of a valid configuration (i.e., false negative) or neglect vertices that are necessary (i.e., false positive).

Detecting External Clause Redundancies Similar to the detection of core and dead features, we split this step into two parts, checking whether the old externally redundant clauses are still redundant and checking whether there are new externally redundant clauses. Iff Δ removes or replaces constraints, it is necessary to check whether old redundant clauses are still redundant. Checking redundancy of previously redundant clauses is mandatory. If a former redundant clause is now required and is not added to the MIG, the resulting MIG would be incorrect.

Checking whether previously non-redundant clauses have become redundant is optional, as it does not impact the correctness of the MIG, but only increases its number of edges. Therefore, we consider three different options if Δ adds or replaces constraints. First, checking all previously non-redundant clauses for redundancy. Second, using a heuristic to test only a subset of the previously non-redundant clauses. In this case, a clause is checked for redundancy only if it contains at least one feature from any clause in the set of added constraints $\mathcal{D}^+$. Third, skipping the second part of the operation and do not check any previously non-redundant clauses. Clearly, only the first option guarantees completeness for finding all externally redundant clauses. However, it also does not have any performance improvement compared to the advanced build process. Thus, we favor the second and third option, which both may introduce redundancy to the new MIG. In our evaluation, we test performance and potential loss of completeness of both options.

Detect Implicit Strong Edges For this step, we make very similar adaptions as for the step of detecting external clause redundancies for the same reasons. We split the step into two parts, checking if previously implicit strong edges are still valid and checking if we can derive new implicit strong edges. If Δ removes or replaces constraints, we must check whether every previously implicit strong edge is still valid. If Δ adds or replaces constraints, we again consider the three options of running the entire second part of the operation, using a heuristic to investigate only a subset of weak edges, or skip the second part of the operation altogether. For the second option, we use a similar heuristic as for detecting redundancies. In particular, we only test weak edges that contain at least one feature from any clause in $\mathcal{D}^+$. Again, the first option does not have any performance benefits, but is the only one that

leads to completeness of the resulting MIG. Thus, we evaluate the performance and loss in completeness of the second and third option in our evaluation.

3.6 Evaluation

By using a MIG-assisted decision propagation algorithm, we effectively split the process of decision propagation into two phases, an initial *offline phase*, in which the MIG is constructed and a repeatable *online phase*, in which the configuration process is performed. With MIGs, we aim to speed-up the online phase of decision propagation by doing further computations during the offline phase (i.e., building the MIG). Therefore, we need to evaluate the performance of different algorithms for decision propagation. We compare the offline and online execution time of decision propagation with differently built MIGs against SAT-based decision propagation (cf. Section 3.3.1). In particular, we aim to answer the following research questions:

- RQ_1 Does the choice of a decision propagation algorithm affect the execution time of the *offline* phase?
- RQ_2 Does the choice of a decision propagation algorithm affect the execution time of the *online* phase?
- RQ_3 Given a number of configuration processes, which decision propagation algorithm is superior to others in terms of overall execution time and memory consumption?

Further, we presented an incremental build process for MIGs that aims to improve the performance overhead of the offline phase for frequently evolving feature models (cf. Section 3.5). To evaluate whether the concept for incrementally building MIGs is able to decrease the performance overhead, we compare it to the advanced build process. In particular, we pose three more research questions that enable us to assess the incremental build process. First, we examine whether the incremental build process improves the MIG build time compared to the advanced build process. Second, as the incremental build process uses some heuristic analysis, we are also interested in whether usage of an incrementally built MIG is less efficient than a MIG built with the advanced build process. Third, considering this potential trade-off between build time and completeness, we investigate under which circumstances it is suitable to use the incremental instead of the advanced build process. In particular, we want to know whether there is an indicator within the change information that lets us reason about the suitability of the incremental build process. In summary, we aim to answer the following additional research questions:

- RQ_4 Does the incremental build process perform faster compared to the advanced build process?
- RQ_5 Does the loss in completeness of an incrementally built MIG affect its effectiveness in analyses?
- RQ_6 Can the usefulness of an incremental build process be inferred by the characteristics of a feature model change?

3.6.1 Setup of Experiments

To answer our research questions, we perform three experiments in total. The first two experiments regard the potential performance increase of decision propagation and the initial performance overhead when using a MIG built with the basic or advanced build process. In these two experiments, we compare four different algorithms for decision propagation using 120 real-world systems. In addition to execution time, we measure the memory consumption of the derived modal implication graph for each feature model. Our third experiment regards the performance of incremental build process and the resulting MIG. In this experiment, we use the evolution histories of four real-world systems to build a MIG for many different feature-model versions with the advanced and the incremental build process. Then, we use the resulting MIGs in decision propagation analysis to examine their performance.

In the following, we describe which configurable systems and decision-propagation algorithms we consider, how we designed the individual runs, and what values we measure during a single run. First, we present our subject systems and their respective evolution histories. Second, we explain the different algorithms and parameters we use in our experiments. Third, we explain the design of our experiments. We describe which are the relevant variables we measure in order to answer our research questions. Lastly, we provide relevant details on our implementation of the incremental build process and the evaluation environment.

Subjects Systems

For our first two experiments, we use the feature models of 120 real-world configurable systems with varying sizes and complexity, which have been used in prior studies [Berger et al., 2013b; Knüppel et al., 2017]. The majority of these feature models (117) contain between 1,166 and 1,397 features. Of these 117 models, 107 comprise between 2,968 and 4,138 cross-tree constraints, while one has 14,295 and the other nine have between 49,770 and 50,606 cross-tree constraints. The remaining three models contain an even higher number of features. The feature models from the systems Automotive01, Automotive02, and Linux contain 2,513, 18,616, and 6,889 features and 2,833, 1,369, and 80,715 constraints, respectively.

In the remaining experiments, we require feature models in different versions over time. For this, we use the version histories of four real-world systems. For each system, we have one feature model per version in the configurable system's history. *Busybox* (500 – 700 features) and *Linux* (16,000 features) are software systems from the operating system domain. The evolution history of *Linux* contains 14 versions, ranges from November 2013 to December 2013, and has short and long times between versions (i.e., within a day and within a month). In case of *Busybox*, we have two version histories from May 2007 to May 2010 for the same system that differ in the time between versions. For *Busybox (Commits)*, each of its 187 versions corresponds to the state of the system after a commit in its version control system that changed the feature model. In contrast, the history of *Busybox (Monthly)* contains 37 monthly snapshots of the system. *FinancialServices01* (500 – 700 features) comes from the financial domain and represents a family of financial products rather than software. Its evolution history spans from May 2017 to March 2019 and contains 20 monthly

snapshots. *Automotive02* (14,000 – 19,000 features) is a cyber-physical system from the automotive domain. Its evolution history contains 6 monthly snapshots.

Evolution Histories

In order to perform an incremental build, at least one MIG from a previous version is required. For each of our subject systems, we have an evolution history that contains multiple consecutive versions (i.e., $\vec{\mathcal{H}} = (V_1, ..., V_n)$). For any pair of versions (V_a, V_b), we can use the MIG of the first version V_a to incrementally build the MIG for the second one V_b. Note that it is not necessary to use the MIG from the directly preceding version, but any preceding version.

To see the impact of different evolution steps, we test different scenarios that lead to a different list of version pairs. The most natural approach would be to evaluate the incremental build process with regard to all consecutive feature model versions (i.e., V_1 and V_2, V_2 and V_3, and so on). However, this would require that a regular MIG is available for each version, which is then used to incrementally build the MIG for the next version. Thus, we also consider the scenario that there is only one regular MIG from the first version of a model. In particular, we consider three different lists of version pairs derived for any given evolution history, namely *consecutive*, *accumulative*, *sequential*.

Consecutive With the consecutive version pair list, we aim to show the impact of building an incremental MIG for every new version based on the MIG of the directly preceding version. For a given evolution history, we construct version pairs, such that each version is combined with its direct predecessor (i.e., $(V_1, V_2), (V_2, V_3), ..., (V_{n-1}, V_n)$). For incrementally building a MIG for any version, we use the regular MIG from the preceding version as input.

Accumulative With the accumulative version pair list, we aim to demonstrate the impact of only using an incremental build process over several consecutive versions. We use the same version pairs as for the consecutive version pair list (i.e., $(V_1, V_2), (V_2, V_3), ..., (V_{n-1}, V_n)$). However, instead of using the regular MIG from the preceding version, we use an incrementally built MIG as input. Only for the first pair (V_1, V_2), we use the regular MIG of V_1.

Sequential With the sequential version pair list, we aim to show the impact of skipping some versions. For a given evolution history, we construct version pairs, such that each version is combined only with the initial version (i.e., $(V_1, V_2), (V_1, V_3), ...,$ (V_1, V_n)).

Decision-Propagation Algorithms

In the first two experiments, we compare the following algorithms for decision propagation:

Short Name	Name
NSAT	Naïve SAT-based
ASAT	Advanced SAT-based
IMIG	MIG-assisted using an *incomplete* MIG
CMIG	MIG-assisted using a *complete* MIG

To ensure a fair comparison of all algorithms, we employ a white-box evaluation, where each algorithm uses the same base implementation as described in Section 3.3.1.

Each algorithm performs certain tasks during its offline phase. Both SAT-based algorithms determine the core and dead features (i.e., initial decision propagation). In addition to computing core and dead features, both MIG-assisted algorithms (cf. Section 3.3.1) derive a modal implication graph. While CMIG derives a complete and minimal MIG using the advanced build process, IMIG derives an incomplete and non-minimal MIG using the basic build process. The modal implication graph is implemented as an adjacency list, due to reasons of memory efficiency. It is stored to and loaded from persistent memory using Java's serialization mechanism.

During their online phase, all algorithms calculate implied literals, as described in Section 3.3.1. While the SAT-based algorithms solely query the SAT solver, the MIG-assisted algorithms additionally traverse through a MIG to avoid SAT queries.

Parameters of the Incremental Build Process

As stated in Section 3.5, we can choose different options for the two most time-consuming operations of detecting *external clause redundancies* and *implicit strong edges*. These options influence the execution time of the build process and the completeness of the resulting MIG. For the advanced build process, we have two options for each operation, either performing it (✓) or skipping it entirely (×). For the incremental build process, the options depend on the advanced build process. If we skip an operation in the advanced build process (×), we also have to skip this operation in the incremental build as well (×). This is due to the fact that we use the result of the operations from the MIG of an earlier version and if an operation was skipped the required information is missing. If we performed the operation in the advanced build process (✓), we have two options, using a heuristic to speed up the second part of the operation (*heuristic*) or skipping the second part of the operation entirely (*skip*). Combining the different values for these options, results in the following five parameter settings that we use for our experiments:

ID	Original		Incremental	
	Redundancy	*Strong*	*Redundancy*	*Strong*
1	×	×	×	×
2	✓	×	*skip*	×
3	✓	×	*heuristic*	×
4	✓	✓	*heuristic*	*heuristic*
5	✓	✓	*skip*	*skip*

Offline Phase (Experiment 1)

In our first experiment, we measure the execution time for each algorithm's offline phase. As stated above, the offline phase of each algorithm consists of all tasks after receiving the CNF of a feature model and before starting the actual configuration process. As the process of creating a CNF is independent from the chosen algorithm, we do not include it as part of the offline phase, but use the CNF as initial parameter for each algorithm. To avoid computational bias and calculate a representable mean value for each feature model and algorithm, we repeat the experiment 200 times. Furthermore, we compensate for the warm-up effect of the Java virtual machine (JVM) by performing an initial execution without any measurement.

In order to be able to draw meaningful conclusions, we formulate the following null hypotheses from RQ_1: H_0^{RQ1}: *The execution time of the offline phase is the same for all investigated decision-propagation algorithms.* To test our hypotheses we use a paired Wilcoxon-Mann-Whitney test with a confidence interval of 95%. Our expectation is that the offline phase of both SAT-based algorithms is faster than the offline phase of the MIG-assisted algorithms, but that they perform worse during the online phase. This is due to the difference in effort of the algorithms during the offline phase. NSAT and ASAT do the least amount of precomputations, while IMIG additionally derives an incomplete modal implication graph and CMIG even derives a complete modal implication graph. This means that, during the online phase, CMIG can access more information than IMIG, while IMIG has more information than ASAT and NSAT. Hence, we expect that for the offline phase NSAT and ASAT are faster than IMIG, which is again faster than CMIG.

Online Phase (Experiment 2)

In our second experiment, we measure the execution time for each algorithm's online phase. We simulate a configuration process by using random decisions, as we cannot know which decisions users would make in their configurations and want to avoid relying on false assumptions. The simulated configuration process consists of the following steps:

1. Start with an empty configuration
2. Randomly choose an undefined feature
3. Randomly define the feature (i.e., select or deselect)
4. Apply decision propagation
5. Repeat 2–4 until all features are defined

We measure the execution time for each individual application of decision propagation. In the experiment, we neglect the time that a user would need to make configuration decisions (i.e., reasoning and input), as these values highly depend on the user and, thus, would bias our results. Furthermore, this is not an issue for automated configuration processes, such as t-wise sampling [Al-Hajjaji et al., 2016a; Johansen et al., 2012a].

We use a pseudo random generator, which has the advantage that we can fix the random seed for each iteration of the experiment. Therefore, we ensure that all

algorithms get the same series of random decision and, thus, that the resulting configurations are equal. To get meaningful results, we repeat the experiment 200 times with different random seeds. Analogous to Experiment 1, we compensate for the JVM warm-up effect.

Analogous to experiment 1, we formulate the following null hypotheses from RQ_2: H_0^{RQ2}: *The execution time of the online phase is the same for all investigated decision-propagation algorithms.* For the online phase, we expect that the fastest algorithm is CMIG, followed by IMIG, ASAT, and NSAT, in that order.

Build Incremental MIGs (Experiment 3)

In our third experiment, we measure two different variables: the time to build a MIG and the time of a decision propagation using the built MIG. In the first part of each run, we build a MIG using the advanced build process and an incremental MIG for a given feature model version of a subject system. We separately measure how much time it takes for both build processes to finish.

In the second part of each run, after building both MIGs, we use each MIG in a series of decision propagations to measure if there is any difference in their performance. To this end, we choose 200 random literals from the feature model and perform a decision propagation with each of the chosen literals as a starting point (i.e., including it in an empty configuration). We measure the time it takes for each decision propagation to finish and sum up the results for each MIG. We do not allow duplicates in the random literal list and use the same literals for every MIG on the same version of a feature model.

Implementation

We base our implementation on the open-source framework FeatureIDE [Krieter et al., 2017; Meinicke et al., 2017]. The implementation is written in Java and uses Sat4J (Version 2.3.5) [Le Berre and Parrain, 2010] as a SAT solver. With Sat4J, we are able to employ incremental SAT solving. For each feature model we create a separate solver, which is able to deduce new clauses while solving one query and reuse these clauses in subsequent queries. In order to ensure a fair comparison between different algorithms, we base all decision propagation algorithms and MIG build process are based on the same basic implementation and make sure that they use the same underlying software framework (e.g., for loading feature models and using decision propagation with a MIG).

Evaluation System

We run our first two experiments on a notebook with the following specifications:

- *CPU*: Intel Xeon E3-1505Mv5 (2.80 GHz)
- *Physical Memory*: 64 GB
- *JVM Max Memory*: Xmx: 4 GB
- *OS*: Windows 7
- *JVM*: JRE 1.8.0_121 (64-Bit)

Table 3.2: Offline and online time of evaluated algorithms for a selection of feature models (mean value over 200 experiments)

Feature Model	**#Features**	**#Clauses**	**MIG Memory (Byte)**	**Offline time in s (⌀)**		
				ASAT	*IMIG*	*CMIG*
FreeBSD 8.0.0	1,397	14,295	243,168	0.04	0.42	6.89
Automotive01	2,513	2,833	1,098,248	0.07	0.70	11.60
Linux 2.6.28.6	6,889	80,715	2,157,320	0.27	16.75	399.98
Automotive02	18,616	1,369	5,088,720	2.30	42.47	296.73
All models (⌀)	–	–	–	0.03	0.62	6.99

For our third experiment, we use the following hardware:

- *CPU*: Intel Core i7-5500U (2.4 GHz)
- *Physical Memory*: 16 GB
- *JVM Max Memory*: Xmx: 4 GB
- *OS*: Linux 5.10.23-1-MANJARO
- *JVM*: OpenJDK 64 Bit 15.0.2

3.6.2 Results of Experiments

In the following, we present and analyze our evaluation results and answer our research questions.

Offline Phase (Experiment 1)

In Table 3.2, we give an excerpt of the aggregated evaluation results for a selection of our subject systems. For brevity, we do not list the results from all feature models. For each feature model, we list its number of features and constraints, memory consumption of the MIG, and aggregated measurements of our experiments. We show the execution time that each algorithm needs during its offline phase. All shown results represent the mean value over the 200 conducted experiments. We also show the mean of all values over all feature models and conducted experiments at the bottom of the table. However, we omit the results of NSAT, as its offline phase is equal to ASAT and its online phase execution time is orders of magnitude larger than all other algorithms (e.g., over 100 times larger compared to ASAT).

We display the results of our first experiment in Figure 3.3 in the first diagram for most feature models. We excluded the four largest feature models (i.e., Automotive01, Automotive02, FreeBSD, and Linux) from the diagram, as they are visually hard to compare to the other models due to their size. Nevertheless, we state the results for these models in Table 3.2. Each data point represents the offline time of a particular algorithm and feature model. On the y-axis, we show the execution time in milliseconds and on the x-axis, the number of features in the feature model. Our data reveals that the execution times from the different algorithms differ in orders

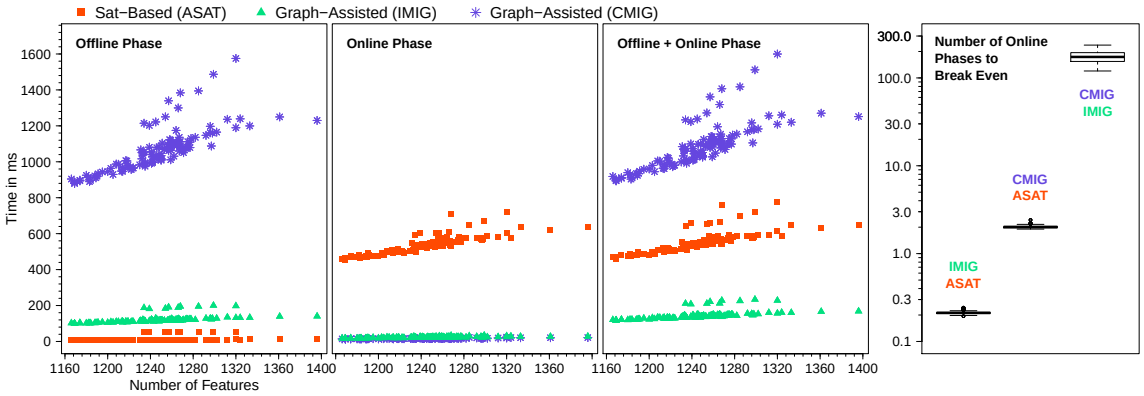

Figure 3.3: Execution time of offline (1), online (2), and combined offline and online phase (3) of all algorithms for multiple feature models (sorted by size). Break-even point (4) of two algorithms indicates the number of online phases one algorithm's overall execution time becomes faster than another (e.g., CMIG is faster than ASAT for two or more online phases)

of magnitude. For instance, for the feature model Automotive01, ASAT required 67 ms, IMIG 696 ms, and CMIG 11,596 ms for the offline phase. In terms of memory consumption, the additional memory required to store a modal implication for IMIG and CMIG lies between 0.25 MB for the feature model FreeBSD and 5.1 MB for Automotive02 with a mean value of 0.9 MB over all 120 feature models.

In Table 3.3, we show the p-values of the Wilcoxon-Mann-Whitney test for all pairwise combinations of algorithms. In all cases, we received a p-value of less than 10^{-15} and, thus, we can reject our null hypothesis H_0^{RQ1}. For all feature models ASAT needs significantly less time for its offline phase than the two MIG-assisted algorithms. Likewise, IMIG needs significantly less time than CMIG.

RQ_1 Therefore, we can answer RQ_1: Yes, there is a significant difference in the time required for the offline phase of the different algorithms. These results are expected, as the algorithms' offline phases differ in the amount of precomputations. While ASAT only detects core and dead features, IMIG has to derive an incomplete modal implication graph in addition. Moreover, CMIG does all of the above and also computes implicit strong edges within the modal implication graph to make it complete.

Online Phase (Experiment 2)

We depict the aggregated results of our second experiment in Figure 3.3 in the second diagram for most feature models. We provide the results for the remaining feature models in Table 3.4. Analogous to the diagram for the offline time, each data point represents the mean execution time over 200 experiments for a particular algorithm to define 100% of the variant features of one feature model. We show the execution time of the online phase when 3%, 10%, and 100% of variant features were defined. On the y-axis, we show the execution time in milliseconds and, on the x-axis, the number of features in the feature model. From our data we can see that for every feature model ASAT requires more online time than both MIG-assisted algorithms.

Table 3.3: Pairwise comparison of algorithms

Attribute	**ASAT / IMIG**	**ASAT / CMIG**	**IMIG / CMIG**
p-value H_0^{RQ1}	$< 10^{-15}$	$< 10^{-15}$	$< 10^{-15}$
p-value H_0^{RQ2}	$< 10^{-15}$	$< 10^{-15}$	$< 10^{-15}$

Table 3.4: Offline and online time of evaluated algorithms for a selection of feature models (mean value over 200 experiments)

Feature Model	**$\sum$ Online time in s for relative number of defined features (⌀)**								
	3%			**10%**			**100%**		
	ASAT	IMIG	CMIG	ASAT	IMIG	CMIG	ASAT	IMIG	CMIG
FreeBSD 8.0.0	0.21	0.05	0.04	0.44	0.07	0.05	1.82	0.10	0.08
Automotive01	1.54	0.36	0.35	3.10	0.56	0.54	6.05	0.76	0.74
Linux 2.6.28.6	11.78	5.75	4.24	26.98	8.39	5.63	80.67	10.07	6.66
Automotive02	329.29	38.08	37.84	535.70	58.77	58.45	821.48	68.03	67.68
All models (⌀)	2.98	0.38	0.36	4.96	0.58	0.55	8.10	0.68	0.64

In contrast, there is no big difference in the time required by both MIG-assisted algorithms. Nevertheless, using CMIG indicates slight improvements over IMIG.

To illustrate the results of the second experiment in more detail, we depict the execution time for each individual decision propagation for the feature model of Linux in Figure 3.4. Each data point originates from *one* of the 200 conducted experiments and represent the execution time of decision propagation by a particular algorithm. On the y-axis, we depict the time in milliseconds and, on the x-axis, the number of defined variant features before decision propagation was executed. The regression curves indicate the mean execution time over 200 experiments. For ASAT the data points for a particular x-value spread wide around the regression curve. However, most data points lie above the data points from CMIG and IMIG. While CMIG shows slightly better results than IMIG, the difference between both is mostly in the range of a few milliseconds. It is also notable that, for both MIG-assisted algorithms, there are many data points that are close to zero.

We list the results from the Wilcoxon-Mann-Whitney test for Experiment 2 in Table 3.3. Again, we can reject our null hypothesis H_0^{RQ2} as we received a p-value of less than 10^{-15} for each algorithm combination. ASAT needs significantly more time than IMIG and CMIG, whereas CMIG is faster than IMIG to a significant, but rather small degree.

RQ_2 Therefore, we can answer RQ_2: Yes, there is a significant difference in the execution time of decision propagation for the different algorithms. These results correspond to our expectations, as all algorithms can access a different amount of information during their online phase. CMIG traverses a complete modal implication graph, while IMIG uses a graph that might lacks some strong edges. By contrast, ASAT does not use any additional data structure and has to infer all needed feature

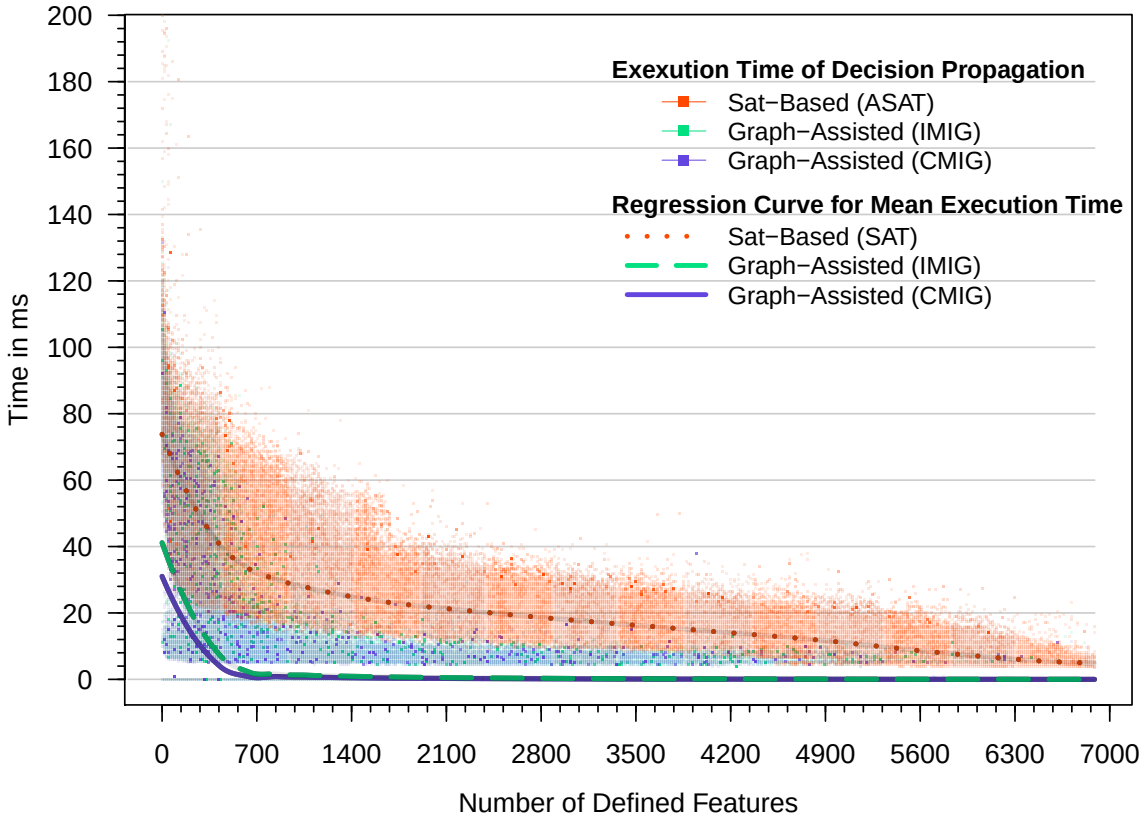

Figure 3.4: Execution time during online phase with ASAT, IMIG, and CMIG for the feature model of Linux

dependencies on-the-fly. The low performance increase of CMIG over IMIG is due to the small number of implicit strong edges that can be derived in the offline phase.

Comparison of Offline and Online Phase

As the number of configuration processes and changes to the feature model might differ for each configurable system, we are interested in the combined cost of offline and online phase. For a better comparison of execution times for offline and online phase, we present the diagrams for both results side by side in Figure 3.3 with both diagrams sharing the same y-axis. Moreover, we visualize the combined cost of offline and online phase in the third diagram of Figure 3.3. In this diagram, we add the time needed for the offline phase to the time required to execute the online phase once. Again, we depict the result of the algorithms for all but the largest feature models. We visualize the number of online phases necessary for each algorithm to break even with the other algorithms in the fourth diagram of Figure 3.3.

The visualization clearly indicates that IMIG needs less time than ASAT even for just one iteration of the online phase. Regarding CMIG, we see that its higher offline time compared to ASAT is amortized after two iterations of the online phase. In our evaluation, we experienced only two exceptions of this observation for the feature models FreeBSD (three iterations) and Automotive02 (five iterations) (cf. Table 3.2). As the online time for both MIG-assisted approaches only differs slightly, CMIG needs many more online phases in order to amortize its initial costs when compared

Table 3.5: Absolute and relative build and usage times for all systems and parameter settings

System	Parameter	Build Time					Usage Time				
		Orig (s)	Inc (s)	Ratio			Orig (s)	Inc (s)	Ratio		
		∅	∅	*Min*	∅	*Max*	∅	∅	*Min*	∅	*Max*
Busybox	1	0.002	0.003	0.478	0.823	1.690	0.034	0.034	0.771	0.997	1.052
(Commits)	2	0.012	0.004	0.311	3.176	6.333	0.034	0.034	0.662	0.997	1.050
	3	0.012	0.006	0.275	2.416	5.266	0.034	0.034	0.959	1.002	1.070
	4	0.086	0.018	0.283	12.732	32.996	0.034	0.034	0.955	1.000	1.204
	5	0.086	0.004	0.301	21.094	47.502	0.034	0.034	0.948	1.001	1.191
Busybox	1	0.003	0.003	0.589	0.766	0.919	0.040	0.040	0.973	0.999	1.026
(Monthly)	2	0.013	0.004	1.998	2.981	4.226	0.040	0.040	0.972	0.999	1.031
	3	0.013	0.008	0.911	1.990	3.490	0.040	0.040	0.970	0.997	1.021
	4	0.102	0.025	1.296	8.614	26.702	0.041	0.041	0.966	1.014	1.058
	5	0.102	0.005	10.534	19.867	37.456	0.041	0.040	0.986	1.034	1.056
Financial-	1	0.161	0.165	0.925	0.977	1.031	0.198	0.197	0.929	1.006	1.069
Services01	2	0.345	0.182	1.336	1.913	2.348	0.195	0.197	0.934	0.988	1.030
	3	0.341	0.333	0.841	1.038	1.392	0.196	0.197	0.956	0.995	1.049
	4	18.809	13.218	0.969	1.604	2.992	0.195	0.193	0.963	1.009	1.060
	5	18.837	6.792	0.853	9.975	23.033	0.195	0.192	0.974	1.017	1.051
Auto-	1	3.109	3.535	0.822	0.882	0.958	7.490	7.502	0.992	0.999	1.005
motive02	2	8.583	3.873	1.672	2.120	3.395	7.451	7.499	0.985	0.994	0.999
	3	8.561	9.120	0.690	1.080	1.616	7.447	7.458	0.993	0.998	1.005
	4	2090.844	1547.998	1.069	4.244	13.733	7.353	7.365	0.992	0.998	1.005
	5	2095.159	4.270	110.037	424.784	1221.410	7.362	7.471	0.977	0.986	1.001
Linux	1	29.326	29.446	0.829	0.998	1.127	15.686	15.677	0.992	1.001	1.010
	2	2218.864	132.196	15.514	16.806	18.269	15.667	15.664	0.993	1.000	1.004
	3	2228.968	2845.951	0.748	0.783	0.815	15.671	15.665	0.993	1.000	1.008

to IMIG. In detail, we measured between 112 and 801 necessary iterations, with a mean value of 185 over all feature models.

RQ_3 Considering the observations, we made from the evaluation results, we can answer RQ_3: While IMIG and CMIG need more offline time than ASAT, this extra initial cost is amortized relatively quickly. Regarding IMIG, it is already faster than ASAT for even one complete configuration process. Only when considering an incomplete online phase (i.e. creating only a partial configuration) ASAT is faster than IMIG due to its fast Offline Phase. Thus, in most cases (i.e., for less than 112 online phases), IMIG seems to be preferable over the other three algorithms, as it provides a good trade-off between the time required for offline and online phase. ASAT and CMIG are superior over other algorithms in some extreme cases. When the feature model evolves more frequently than the configurations, ASAT can be superior, due to its efficient offline phase. In case that configurations are updated frequently while the feature model does not evolve for a longer period of time, CMIG can be superior as its online phase requires less time than ASAT and IMIG. Regarding memory consumption, in our experiments we found that the memory required to store a modal implication graph was at maximum 5.1 MB, which is relatively small compared to the available main memory on modern hardware. Thus, the additional memory consumption can be neglected for most applications.

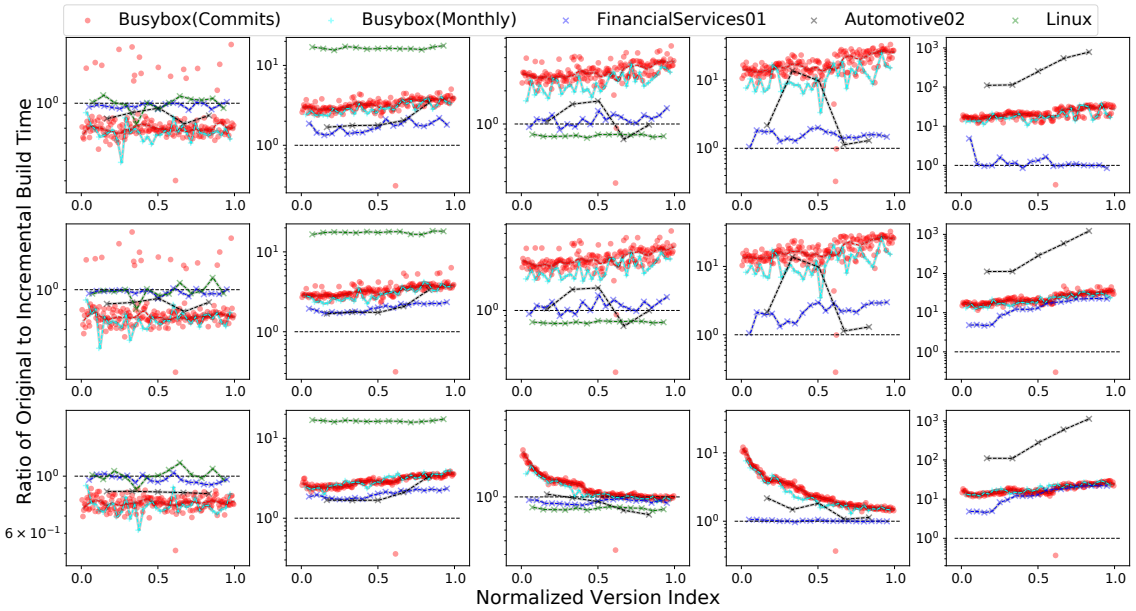

Figure 3.5: Build time ratio (regular/incremental) for all systems and versions (X: Normalized versions difference; Y: Ratio; Columns: Parameters 1–5; Rows: Consecutive, Accumulative, Sequential)

Incremental Build Process (Experiment 3)

In Table 3.5, we show an overview of our measurements in our third experiment for all four subject systems and all parameter settings. All values in the table are aggregated over all version pairs. We show the mean time in seconds for building MIGs with the advanced build process and incremental MIGs and the mean time in seconds for executing decision propagation with the regular and the incremental MIGs. In addition, we show the ratio between regular and incremental MIG for the build and usage time. For the ratio, we state the min, mean, and max values, respectively. Note that, we omit the data for the system *Linux* with parameter settings 4 and 5, because these runs did not finish within 24 hours.

In addition to Table 3.5, we show a more detailed plot of the build time ratio in Figure 3.5. The figure contains one scatter plot per parameter setting (columns) and version pair list (rows). Each plot contains the data of all measured values for the particular parameter setting and version pair list. The y-axis shows the ratio on a log scale (i.e., the higher the more time was saved by the incremental build). The x-axis shows the index of the used version divided by the number of versions in the evolution history of the system. This normalization is done in order to evenly space out all systems over the x-axis. Additionally, each plot contains a regression curve per system to visualize any trend in the data.

From our data table and plots we can make three observations. First, the differences in the data for different version pair lists are relativity small. All plots in a column roughly show the same behavior with the exception of the sequential version pairs (third row) for parameter settings 3 and 4. We can see that the performance improvement of the incremental build process is better if the versions are closer together. For example, for parameter setting 4, *Busybox (Commits)* (V_1, V_2) has a

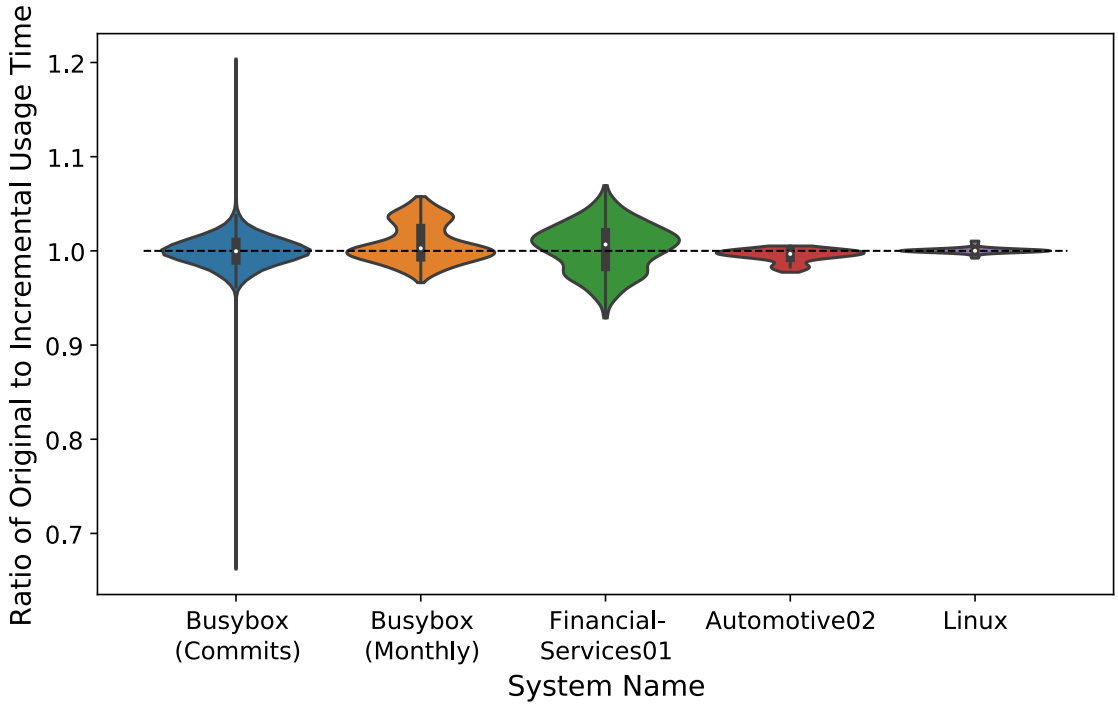

Figure 3.6: Aggregated usage time ratio (regular/incremental) for all systems

speed-up of factor 10, in contrast to factor 2 for (V_{186}, V_{187}). Second, for parameter setting 1, we can see no benefit of using an incremental build process over the advanced one. There is even a small overhead (i.e., factor 0.76 – 0.99 on average) that increases the overall build time. This can be explained by the time it takes the incremental build process to compute the feature model change. Further, none of the two most time-consuming operations that were modified in the incremental build can be utilized as they are skipped for both build processes. Third, there is a moderate improvement for parameter settings 2 and 3 (i.e., factor 0.78 – 16.8 on average) and a high improvement for parameter settings 4 and 5 (i.e., factor 1.6 – 424.78 on average). This is expected as settings 2 and 3 enable the detection of externally redundant clauses and settings 4 and 5 also enable the detection of implicit strong edges. Both operations are partially skipped during the incremental build process. There is also a clear difference between skipping the second part of the respective operations and using a heuristic. When using a heuristic, it substantially decreases the build time improvement of the incremental build compared to skipping (e.g., from factor 4.24 to factor 424.78 for *Automotive02*).

In Figure 3.6, we show the ratio of the usage time of an MIG built with the advanced build process and an incremental MIG aggregated for each system. The y-axis shows the ratio and the x-axis the system. We aggregated the data, as there is almost no difference in the data distribution for different parameter setting and version pair lists. These results lead us to the following observation. The difference in usage time for regular and incremental MIGs is almost non-existent. Although, we can see some relative differences for *Busybox* and *FinancialServices01* however in absolute terms the difference is within a few milliseconds (e.g., the maximum time difference for *Automotive02* is 109 ms). Both the ratio and the actual difference are negligible for a

single configuration process. The practical difference in the MIG's completeness also seems to be independent from the chosen parameter settings and version distance.

RQ$_4$ The incremental build process is able to drastically outperform the advanced build process. This is dependent on the concrete parameter settings, though. When using the basic build process that does not detect redundancies or implicit strong edges, and thus computes an incomplete MIG, the incremental build process has no benefit. Only when the advanced build process aims for a complete MIG, the incremental build process is able to achieve substantially lower building times (varying from a factor of 10 to 400 on average). In these cases, it seems to be more efficient to not rely on a heuristic, but skipping entire parts of certain operations within the incremental build, as this always improves the performance (e.g., in case of *Automotive02* from factor 4 to factor 400 on average).

RQ$_5$ The incremental build process does barely affect the resulting MIGs effectiveness within decision propagation. An incremental MIG cannot be guaranteed to be complete regardless of the considered parameter settings. Thus, there is a theoretical loss in completeness for the incremental build process. However, the practical performance impact when using an incremental MIG compared to a complete MIG is almost not noticeable in our experiments (within a factor of 1.01 on average). This is in line with our findings from our first paper on MIGs, where we found that a complete MIG only slightly improves decision propagation compared to an incomplete MIG [Krieter et al., 2018].

RQ$_6$ The difference in usage time between complete, incomplete, and incremental MIGs is relatively small. On the other hand, the build time varies dramatically for different parameter settings and whether an advanced or incremental build process is used. Thus, it makes sense to use the incremental build frequently to keep build times low and to use incrementally built MIGs as input for the next incremental build process. One could argue that it is more beneficial to only use incomplete MIGs all together (i.e., using the advanced build process with parameter setting 1) as this results in the fastest build process and only small loss in effectiveness. However, there are circumstances, where the incremental build process is still superior, for instance, when a complete MIG is already present (e.g., from other analyses). In this case, using an incremental build is more suitable than rebuilding an (in)complete MIG from scratch. Overall, for the incremental build process shows a substantial speed-up for all parameter settings (except 1) without noticeable loss in effectiveness, even for large evolution steps.

3.6.3 Threats to Validity

In the following, we reason about possible threats to validity within our evaluation and explain what we did to mitigate any potential biases. To this end, we categorize the possible threats into internal and external threats to validity.

Internal

A number of issues might threaten the internal validity of our results. The results could be biased in favor of our proposed algorithms. This is due to the fact, that we implemented all evaluated algorithms by ourselves. However, we use the same base implementation for all algorithms and, for each algorithm, we only do the necessary modifications as described in Section 3.3.1.

Random input data might lead to unrepresentative results. To simulate a configuration process, we used a series of random decisions, which might not correspond to a real-world configuration. This may result in random bias, where we by chance only picked features that result in certain corner cases of decision propagation. However, a randomized approach gave us the capability to efficiently do multiple iterations with distinct random seeds and, thus, gather more data. To avoid random bias, we evaluate each setting with 200 repetitions. Further, we made sure that every MIG is tested against the same list of features.

There may be several causes for a computational bias. First, the JVM may influence the required time for consecutive runs due to just-in-time compilation. To tackle this issue, we performed warm-up computations prior to the first measurement. Second, Java regularly frees the memory of not referenced objects by means of garbage collection. To mitigate this effect, we instructed the JVM to run the garbage collector before building and using a MIG. Third, there may be a general computational bias, which can cause minor differences in measured execution times. To reduce the mitigate this bias, we performed three repetitions for each computation and used the median of those.

For the entire empirical evaluation, we use the CNF transformation implemented in FeatureIDE. As both, the advanced and incremental build process, rely on a CNF input, using a different CNF may change the internal structure of a MIG, and thus may influence measured execution times.

Bugs in our implementations might cause wrong results. We mitigate this issue by deploying unit tests to test each algorithm individually. Furthermore, we compared the resulting configurations of all algorithms and found no difference during all conducted experiments. Additionally, we use matured open-source tools such as Sat4J to further reduce the possibility of bugs.

External

There are some threats that may affect the generalizability of our results. Our results might not transfer to real configuration applications. Our simulated configuration process is likely to be different from a manual configuration by a user with domain knowledge. In addition, starting with a partial configuration may mitigate the problem of a slow initial decision propagation (cf. Figure 3.4). However, a manual configuration process strongly depends on the particular user, which could bias the results as well.

The tested feature models might not be representative of feature models used in practice. Similarly, our evaluation results may not be generalized for evolution histories of other configurable systems. There are only very few histories of industrial systems publicly available and we limited ourselves to such systems (contrary to artificial ones) for more expressive results on real-world scalability. To mitigate this issue, we tested 120 real-world feature models with a varying number of features and constraints that have been used in prior studies [Berger et al., 2013b; Schröter et al., 2016]. We also include the largest real-world feature models referenced in literature at the moment. Furthermore, we evaluated the history of four systems from very different domains that are widely used for empirical evaluations [Nieke et al., 2018; Pett et al., 2019, 2021; Schröter et al., 2016].

In our implementation, we only employ Sat4J as SAT solver. However, we use Sat4J as a black-box, such that other solvers (e.g., SAT, CSP, BDD, MDD) could also be plugged-in. As we shift SAT calls from the online to the offline phase, faster solvers should improve both, online and offline computation.

3.7 Summary

In this chapter, we presented our graph-based data structure *Modal Implication Graph (MIG)* for storing and accessing information about relationships of all pairs of features within a feature model. We gave a brief motivation to highlight the benefits of using a MIG. We then demonstrated the usage of MIGs within decision propagation. Further, we explained the process of building a MIG and described the possibilities to customize the build process in order to trade-off build time for completeness and minimality of the resulting MIG. Additionally, we presented an incremental build process to enable the efficient updating of an existing MIG after feature model evolution. We evaluated how the performance of decision propagation is affected, when using different algorithms with an with a MIG. Further, we evaluated the efficiency of our different build processes and the effectiveness of the resulting MIGs and then argued about the feasibility of building a MIG under different circumstances. We find that MIGs are able to significantly increase the performance of the semi-automated configuration process compared to pure SAT-based approaches. For most use case scenarios, even an incomplete MIG is sufficiently effective, which can be efficiently build and updated after a feature model evolution. Thus, we are able to facilitate the semi-automated configuration process by providing a MIG-assisted decision propagation algorithm and an efficient build process to create the required MIG. In addition, MIGs can also be used in an automated configuration process, such as t-wise sampling, which we describe in Chapter 4 and Chapter 5. Furthermore, we discuss additional applications of MIGs in Chapter 7.

4. Advanced T-Wise Interaction Sampling

This chapter introduces the concept of an adaptable sampling algorithm. It shares material with the following two publications: YASA: yet another sampling algorithm (VaMoS'20) *[Krieter et al., 2020] and* Large-scale T-wise interaction sampling using YASA (SPLC'20) *[Krieter, 2020].*

In this chapter, we present an efficient and flexible t-wise sampling algorithm called *YASA (Yet Another Sampling Algorithm)*. YASA is a general purpose algorithm for creating t-wise configuration samples, which is aims to be more efficient and flexible than state-of-the-art algorithms. We plan to employ YASA in multiple different application scenarios with different requirements on sampling time and properties of the resulting sample. For instance, we use YASA as sampling algorithm for computing our new coverage criterion presence condition coverage, which we introduce in Chapter 5. To facilitate the adaptability of YASA to different scenarios, we enable its customization by means of multiple parameters. To this end, YASA is capable of handling of partial samples and partial configurations as input and output, ignore specific feature interactions, use different covering strategies for individual interactions, and provides some control over sample size and sampling time. Within in YASA, we employ *Modal Implication Graphs (MIGs)*, which we introduced in Chapter 3, to improve its performance when checking for valid interactions.

In the following, we first, give a brief motivation of our new sampling algorithm (see Section 4.1). Second, we present all input parameters of YASA and explain their influence on the sampling process and the resulting sample (see Section 4.2). Third, we explain the main algorithm for creating a sample (see Section 4.3). Fourth, we discuss several optimizations and possibilities to adapt YASA for optimizing sampling time and sample size (see Section 4.4). Finally, we evaluate the efficiency of YASA by comparing it to existing algorithms in generating samples for feature models from multiple real-world systems (see Section 4.5). The content of this chapter is mainly based on two publications, the first introduction of YASA [Krieter et al., 2020] and its application in a challenge of the *Software Product Line Conference* in 2020 [Krieter,

2020]. In addition to these publications, we present more details on the parameters of YASA (see Section 4.2) and the possibilities to adapt the algorithm for different application scenarios (see Section 4.4).

4.1 Motivation

In the last chapter, we presented an approach for a semi-automated configuration process. This approach is well suited for the creation of configurations by a user. However, many scenarios require a fast and automated generation of one or more configurations. One important scenario is software testing of configurable systems. Software testing in general is an important task in software engineering to increase software quality and to check intended software behavior [Ammann and Offutt, 2016; McGregor, 2010]. However, extensive testing can be quite costly and binds resources that could be used in other phases of development. This is especially an issue for testing of highly-configurable systems. As the problem space for configurable systems typically grows exponentially with the number of features [Engström and Runeson, 2011; Lee et al., 2012]. For example, the feature model in Figure 4.1 is an excerpt of the feature model for the system Busybox. This subset of the entire feature model has only 9 features, but already represents 85 valid configurations.

A straight-forward testing strategy for configurable systems is product-based testing, in which the test cases of a system are executed for different configurations [Thüm et al., 2014a]. As testing every possible configuration is usually not feasible, sampling strategies, have been developed to generate a small but representative set of products to test [Oster et al., 2010; Perrouin et al., 2010]. One such sampling strategy is t-wise interaction sampling, which aims to generate a small set of configurations that covers all possible interactions of t configuration options (e.g., none selected, only one selected, all selected, ...) [Cohen et al., 2008; Marijan et al., 2013]. Using t-wise interaction sampling, developers can ensure that each possible combination of t configuration options is indeed contained in at least one configuration in the generated sample.

T-wise interaction sampling is a promising approach, because even when using small values for t (i.e., $t \in \{2, 3\}$) it achieves effective results with a relatively small sample [Abal et al., 2018; Kästner et al., 2009; Marijan et al., 2013]. However, even when using small values of t and a state-of-the-art sampling algorithm, sampling can take an infeasible amount of time, especially regarding large-scale systems with thousands of features [Pett et al., 2019].

In this chapter, we want to tackle the scalability problem of t-wise interaction sampling for large-scale systems. To this end, we introduce YASA, a t-wise interaction sampling algorithm that aims to be efficient and more scalable and flexible than existing approaches. For this, we focus on three improvements compared to existing approaches. First, we employ many optimizations in YASA, such as applying heuristics, caching, and using MIGs (cf. Chapter 3), to increase the sampling performance. Second, we introduce additional parameters to adapt some of our used heuristics to enable more control over the trade-off between sampling time and sample size. These parameters also allow to incorporate domain knowledge of a system in the sampling process to make it more efficient and the resulting sample more suitable for

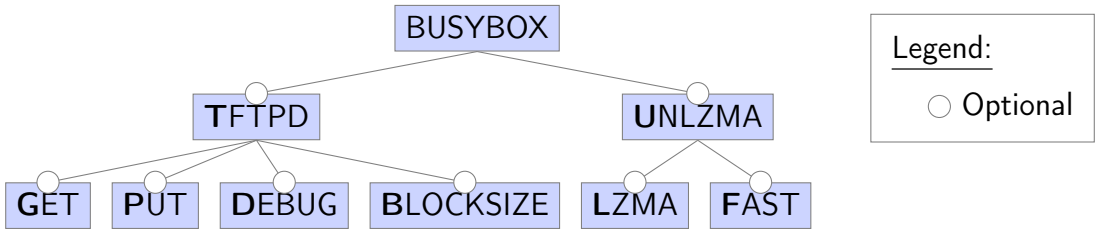

Figure 4.1: Excerpt of *BusyBox* feature diagram

the desired purpose. Third, we enable further modifications of the algorithm in order to adapt it to specific scenarios and give the user even more control over sampling time and properties of the final sample.

Throughout this chapter, we use the feature model in Figure 4.1 as an example to demonstrate our sampling approach. The feature model is an excerpt of the feature model for the system Busybox. It consists of the nine features *BUSYBOX*, *TFTPD (T)*, *GET (G)*, *PUT (P)*, *DEBUG (D)*, *BLOCKSIZE (B)*, *UNLZMA (U)*, *LZMA (L)*, and *FAST (F)*. For all following propositional formulas using these features, we use their provided abbreviated names.

4.2 Parameters of YASA

YASA consists of a basic algorithm that starts with an empty sample and then iterates over all t-wise interactions one at a time. For each valid interaction, it either adds a new partial configuration with the literals of the interaction to the sample or it adds the literals to an existing configuration. We enhance this basic algorithm by applying different heuristic and caching methods that aim to improve its sampling time. We explain the details of the sampling in algorithm in Section 4.3 and Section 4.4. In the following, we start by describing all available input parameters of YASA. These can be used to affect the properties of the resulting sample as well as the sampling time.

In total, there are five parameters that can be given to YASA as input, a *feature model*, an *interaction size* (i.e., a value for t), an *initial sample*, a *feature subset*, and a *resampling limit*. The first two parameters are mandatory, and thus have to be specified in order for the algorithm to run. They are also common to other t-wise interaction sampling algorithms, as they provide the basic information for creating a t-wise sample (i.e., the set of features, their interdependencies, and a value for t). In contrast, the remaining three parameters are optional, which means that they do not have to be specified and we will use a fixed default value in that case. These three parameters enable us to fine-tune YASA for a given configurable system, such that it produces suitable samples in a reasonable time. In the following, we explain every parameter in more detail.

4.2.1 Feature Model

With the feature model, we specify the set of features and all of their interdependencies, which in turn defines the set of all valid configurations and interactions. If no set of expressions is specified (see Section 4.2.4), then the feature set is used as basis for building the set of interactions, which are covered by the resulting sample. From the feature set $\mathcal{F}(\mathcal{M})$, we can derive the set of literals $\mathcal{L}(\mathcal{M})$, which represents all values for each feature. If we then construct all sets of size t containing the literals of different features, we get the list of all t-wise feature interactions $\vec{\mathcal{I}}$. Typically, some interaction from $\vec{\mathcal{I}}$ cannot be covered, as they conflict with the dependencies $\mathcal{D}(\mathcal{M})$ from the given feature model $\mathcal{M}$. Analogues to configurations, we call these interactions *invalid*. An interaction i is invalid, if there exists no valid configuration c for the feature model, which contains i (i.e., $\nexists c \in \mathcal{C}(\mathcal{M}) : valid(c) \wedge i \subseteq c$). Thus, the set of dependencies $\mathcal{D}(\mathcal{M})$ from the feature model defines the set of valid interactions, which is a subset of $\vec{\mathcal{I}}$. Furthermore, the feature model $\mathcal{M}$ is also used to compute a MIG (cf. Chapter 3), which facilitates the detection of invalid interactions (see Section 4.3).

4.2.2 Interaction Size

The parameter t defines the interaction size (i.e., how many features are considered in an interaction). For example, a value of $t = 2$ considers all pair-wise interactions. For two Boolean features a and b the following four interactions are considered: $i_1 = \{\neg a, \neg b\}$, $i_2 = \{a, \neg b\}$, $i_3 = \{\neg a, b\}$, and $i_4 = \{a, b\}$. In the resulting sample all valid interactions of all combinations of two features are covered, such that for each valid interaction i, the sample $\mathcal{S}$ contains at least one configuration that contains i (i.e., $\exists c \in \mathcal{S} : i \subseteq c$). In a three-wise sample all three combinations of three features are considered and so on. Note that a complete t-wise sample covers all interactions of size t and also all interactions with a size smaller than t. For instance, a three-wise sample covers all valid three-wise, pair-wise, and one-wise interactions. For most applications, a larger value for t means that the resulting sample has a greater effectiveness. For example, in testing, typically a three-wise sample is able to detect more faults than a pair-wise sample, because more interactions are tested.

In theory, YASA allows to set any integer value greater than zero for t. Practically however, the value of t is limited by the sampling time, which increases drastically for higher values of t, and the sample size, which also increases with higher values of t. Increasing the value of t directly increases the number of interactions (i.e., $\vec{\mathcal{I}}$) that have to be considered. The size of $\vec{\mathcal{I}}$ can be computed with the following formula, where n is the number of features:

$$|\vec{\mathcal{I}}| = 2^t \cdot \binom{n}{t}$$

Thus, increasing t will increase the number of interactions exponentially. While raising the number of features leads to a polynomial growth that also depends on the value of t. A larger set of interactions means two things. First, the time required for checking and covering all interactions increases. In order to guarantee that a sample has a 100% t-wise coverage, we have to make sure that every valid interaction is

present in at least one configuration, which means we have to look at every interaction at least once during the sampling process and either cover it in a configuration or show that it is invalid. Thus, we assume a linear correlation between interaction size and sampling time, which means than the sampling time also increases exponentially with the value of t. Second, to cover more and also larger interactions requires more configurations. Thus, by increasing the value of t, we also increase the sample size.

4.2.3 Initial Sample

YASA is able to use a predefined initial sample during the sampling process. The initial sample is used as a basis of the final sample. It is not required to meet any specific degree of coverage beforehand and may even contain partial configurations. Instead of starting with an empty sample, YASA takes into account all interactions that have been covered by the initial sample and will complete the sample by adding new configurations and completing any partial ones. As this is an optional parameter, it can omitted and, in this case, the default value is simply an empty sample.

There are many scenarios, in which providing an initial sample can be useful. It allows to include certain configurations in the final sample (e.g., user-defined configurations or all-yes and all-no configurations) and is a way to decrease the sampling time. Furthermore, it enables us to interrupt the sampling process resulting in a partial sample and than later continue the sampling process with this partial sample. It also allows us to update an already computed sample after a feature model evolution. In such a case, the old sample can be kept to a certain degree (i.e., such that no conflicts with the evolved feature model are present) and can than be used as a starting point for a new sample [Pett et al., 2021]. Another possibility is to use a sample from a different sampling algorithm, which does not guarantee a 100% t-wise coverage, as input for YASA. This makes it possible to combine YASA with other sampling algorithms, such as random sampling or any evolutionary approach for t-wise sampling.

4.2.4 Feature Subset

Typically, t-wise interaction sampling algorithms consider interactions between *all* features of a feature model. YASA allows to ignore certain features and only consider combinations between features within a subset of the entire set of features. With this parameter, we can provide a set of features $\mathcal{F}^*$, which is a subset of $\mathcal{F}(\mathcal{M})$. Instead of combining every feature from $\mathcal{F}(\mathcal{M})$ with each other, YASA only constructs all t-sized subsets from $\mathcal{F}^*$ to build the list of interactions $\vec{\mathcal{I}}$. Again, this parameter is optional and can be omitted. In this case, the feature set from the feature model is used to build the set of interactions (cf. Section 4.2.1).

Specifying a feature subset can make sense in large systems, where many features may not interact at all, maybe because they belong to different subsystems or maybe because there is no data or control flow between them. In case that users are aware of such circumstances, it can be helpful to ignore the combinations between these features for sampling, as it can save sampling time and reduce the sample size. Furthermore, this parameter allows us to divide the entire feature set of a feature model into multiple feature subsets and create a sample that considers only

interactions resulting between feature in the same subset and does not have to consider any interaction between features in different subsets. For this, we combine this parameter with the possibility to specify an initial sample (cf. Section 4.2.3). We run YASA repeatedly with different feature subsets and use the sample from a previous run as initial sample for the next run. To this end, YASA is able to return a sample containing partial configurations (see Section 4.4).

For instance, regarding our example in Figure 4.1, we might be aware that the features from the subtrees *TFTPD* and *UNLZMA* are not interacting. In this case, we can divide the feature set into the two smaller subsets $\mathcal{F}_1^* = \{BUSYBOX, T, G, P, D, B\}$ and $\mathcal{F}_2^* = \{BUSYBOX, U, L, F\}$. Note that it is possible for the subsets to overlap. We then run YASA once with $\mathcal{F}_1^*$ as feature subset. Then, we run YASA a second time with $\mathcal{F}_2^*$ as feature subset and the sample from the first run as initial sample for the second one. The final sample contains all interactions resulting from $\mathcal{F}_1^*$ and $\mathcal{F}_2^*$.

The actual distribution of features into subsets must be derived from domain knowledge about the system or through source code analyses (e.g., data flow analysis). Naturally, this parameter increases the risk of missing actual interactions that were not anticipated, and thus are then overlooked, for instance during testing. However, when applied carefully, providing one or more feature subsets has the potential to drastically reduce the amount of interactions that need to be covered, which decreases sampling time as well as sample size. We further generalize the concept behind this parameter in Chapter 5.

4.2.5 Resampling Limit

In YASA, we include a mechanism for refining the resulting sample by removing and resampling some configurations (cf. Section 4.3.4), which can be controlled with an additional parameter m. As YASA is a greedy algorithm and employs heuristics, the resulting sample size is most likely not minimal. To further decrease the sample size, we integrate a mechanism into our algorithm to refine the sample by removing configurations that only cover a small amount of interactions. Afterwards, we reiterate over all interactions that are now uncovered and complete the sample again (cf. Section 4.3.4). We can repeat the process of removing and resampling several times to improve the overall result. However, increasing the number of iterations also increases the sampling time significantly. During the development and testing of our algorithm, we noticed that increasing the number of iterations above a certain point no longer yields substantial improvements on the sample size. Though, how many iterations are reasonable, depends on the sampled product lines. Thus, we included the parameter m to set a limit on the number of iterations for sampling. Users can choose for themselves by how much they are willing to increase the sampling time in order to reduce the sample size. If no value for m is provided, YASA uses the value $m = 1$, which means that after the initial sampling, there is no resampling at all.

Algorithm 4.1 Main sampling algorithm of YASA

Require:
- $\mathcal{M}$ – Feature Model
- t – Interaction Size
- $\mathcal{S}_{init}$ – Initial Sample (*Default*: $\mathcal{S}_{init} = \emptyset$)
- $\mathcal{F}^*$ – Feature Subset (*Default*: $\mathcal{F}^* = \mathcal{F}(\mathcal{M})$)
- m – Resampling Limit (*Default*: $m = 1$)

Return:
- $\mathcal{S}$ – Configuration Sample

1: **function** CREATESAMPLE($\mathcal{M}$, t, $\mathcal{S}_{init}$, $\mathcal{F}^*$, m)
2: $\mathcal{S} \leftarrow \mathcal{S}_{init}$
3: $\mathcal{G} \leftarrow$ CREATEMIG($\mathcal{M}$)
4: $\mathcal{L}_I \leftarrow$ GETLITERALS($\mathcal{F}^*$, $\mathcal{G}$)
5: $\mathcal{S} \leftarrow$ SAMPLE($\mathcal{M}$, $\mathcal{G}$, t, $\mathcal{S}$, $\mathcal{L}_I$)
6: **for** 1 **to** m **do**
7: $\mathcal{S} \leftarrow$ TRIM($\mathcal{S}$)
8: $\mathcal{S} \leftarrow$ SAMPLE($\mathcal{M}$, $\mathcal{G}$, t, $\mathcal{S}$, $\mathcal{L}_I$)
9: $\mathcal{S} \leftarrow$ AUTOCOMPLETE($\mathcal{S}$)
10: **return** $\mathcal{S}$

11: **function** SAMPLE($\mathcal{M}$, $\mathcal{G}$, t, $\mathcal{S}$, $\mathcal{L}_I$)
12: $\vec{\mathcal{I}} \leftarrow$ COMBINE($\mathcal{L}_I$, t)
13: **for** $i \in \vec{\mathcal{I}}$ **do**
14: $\mathcal{S} \leftarrow$ COVER($i, \mathcal{S}, \mathcal{M}, \mathcal{G}$)
15: **return** $\mathcal{S}$

4.3 Constructing a Configuration Sample

In the following, we describe the basic process of our sampling algorithm. In the next sections, we go into detail of the functions `getLiterals` and `combine` for *building the list of interactions*, `cover` for *covering a single interaction* in the sample, and `trim` for *resampling*. Further, we explain certain optimizations to this basic process and how it can be adapted to specific scenarios in Section 4.4.

4.3.1 Basic Sampling Process

We depict the basic outline of our algorithm in pseudo code in Algorithm 4.1. We start by initializing the sample $\mathcal{S}$, the MIG $\mathcal{G}$ and a set of literals $\mathcal{L}_I$. We set the sample $\mathcal{S}$ to the value provided with the parameter $\mathcal{S}_{init}$ or to an empty set if the parameter is omitted (Line 2). We create a MIG $\mathcal{G}$ for the feature model using our basic build process (cf. Section 3.4). The MIG is used to build the set of interactions and during the coverage of each interaction (Line 3). The set $\mathcal{L}_I$ contains literals corresponding to the features in the feature subset $\mathcal{F}^*$ or to the features in the entire feature set $\mathcal{F}(\mathcal{M})$ if this parameter is omitted (Line 4). It serves as basis for building the list of interactions $\vec{\mathcal{I}}$. We create $\mathcal{L}_I$ by calling the function `getLiterals`, which we describe in more detail in Section 4.3.2. After these initializations, we create the sample by calling the function `sample` (Line 5). In this function, we create an ordered

list of interactions $\vec{\mathcal{I}}$, which contains subsets of $\mathcal{L}_I$ with size t, by calling the function `combine` on the set $\mathcal{L}_I$ (Line 12). Then, we iterate over every interaction $i \in \vec{\mathcal{I}}$ and try to cover it in the function `cover` (Lines 13–14). We explain the details of this function below in Section 4.3.3. Once every interaction is processed, all interactions are covered within the sample $\mathcal{S}$.

After creating the first sample, we repeat the sampling process $m-1$ times to reduce the number of configurations in the sample (Line 6). If no value for m is provided or its value is one than the resampling is skipped entirely. For the resampling, we first call the function `trim` that removes all configurations from $\mathcal{S}$ that cover only a few interactions (Line 7). We explain the details of this process in Section 4.3.4. Then, we call the function `sample` again using the trimmed sample (Line 8). Finally, we complete any partial configurations in $\mathcal{S}$ by calling the function `autocomplete`. For each partial configuration in $\mathcal{S}$, this function assigns a value to each feature variable that is currently undefined such that the complete configuration is valid. We describe this function in more detail in Section 4.4.2.

4.3.2 Building the List of Interactions

We create the list of interactions $\vec{\mathcal{I}}$ in two steps. In the first step, we create a set of literals $\mathcal{L}_I$, which represents every assignment for the features in the feature set. Then, in the second step, we list all subsets of $\mathcal{L}_I$ with a size of t to create the list of interactions $\vec{\mathcal{I}}$.

We create the set of literals $\mathcal{L}_I$ in the function `getLiterals`. First, we filter out all core and dead feature variables in the feature set. A core feature is always selected in any valid configuration, thus any interaction that contains the positive literal of a core feature will be covered automatically by any configuration in the sample. For instance, regarding our example in Figure 4.1, the three-wise interaction $i = \{BUSYBOX, B, T\}$ contains the positive literal of the core feature $BUSYBOX$. Thus, any valid configuration that contains the literals B and T also contains $BUSYBOX$, and thus covers i. As there are other interaction that contain B and T, such as $i' = \{B, T, G\}$, there will be configurations in the sample that cover i', and thereby also i. On the other hand, any interaction that contains the negative literal of a core feature variable must be invalid, as there is no valid configuration that deselects a core feature. Therefore, we can ignore all core feature variables in the set $\mathcal{L}_I$. Analogous, we can also ignore all dead feature variables for a similar reasoning. If an interaction contains the negative literal of a dead feature variable, it will be automatically covered in some configuration in the sample and if an interaction contains the positive literal of a dead feature variable, it is invalid. Thus, we can also ignore all dead feature variables for $\mathcal{L}_I$. We can retrieve the set of literals representing the core and dead features variables $\mathcal{D}^U(\mathcal{G})$ from the MIG we computed earlier without any further analysis. For each remaining feature, we add its positive and negative literal to $\mathcal{L}_I$. For example, assume that we call YASA for our example feature model in Figure 4.1 with the two feature subsets $\mathcal{F}_1^* = \{BUSYBOX, B, T, G, P, D, B\}$ and $\mathcal{F}_2^* = \{BUSYBOX, U, L, F\}$. For this, we create the literal sets $\mathcal{L}_{I_1} = \{\neg T, \neg G, \neg P, \neg D, \neg B, T, G, P, D, B\}$ and $\mathcal{L}_{I_2} = \{\neg U, \neg L, \neg F, U, L, F\}$.

Algorithm 4.2 Basic covering strategy of YASA

Require:
i – Interaction
$\mathcal{S}$ – Configuration Sample
$\mathcal{M}$ – Feature Model
$\mathcal{G}$ – MIG for $\mathcal{M}$
Return:
$\mathcal{S}$ – Configuration Sample

```
1: function COVER(i, S, M, G)
2:     c_i ← i
3:     if ∄c ∈ S : c_i ⊆ c then
4:         if ¬satisfiable(c_i, M) then
5:             return S
6:         for c ∈ S do
7:             c' ← c ∪ c_i
8:             if satisfiable(c', M) then
9:                 S ← (S \ {c}) ∪ {c'}
10:                return S
11:        S ← S ∪ c_i
12:    return S
```

With the function `combine`, we list all subsets of $\mathcal{L}_I$ with a size of t to create the list of interactions $\vec{\mathcal{I}}$. For a large set $\mathcal{L}_I$ or higher values of t, this list can be quite long. To avoid any memory issues, rather than creating the entire list and store it in memory, we create an iterator, that sequentially generates every interaction in a defined order. By default the iterator uses a lexicographical ordering. However, by modifying the iterator it is also possible to use another ordering, such as an random order. Continuing our example from before, for $t = 2$ we would create the lists of interactions $\vec{\mathcal{I}}_1 = (\{\neg T, \neg G\}, \{\neg T, \neg P\}, ...)$ and $\vec{\mathcal{I}}_2 = (\{\neg U, \neg L\}, \{\neg U, \neg F\}, ...)$.

4.3.3 Covering an Interaction

YASA processes all interactions in $\vec{\mathcal{I}}$ sequentially with a specific *covering strategy* implemented in the function `cover`. For each given interaction, the applied covering strategy determines how the interaction is processed. In total, there are four possible outcomes for each given interaction. The interaction is either *already covered*, *invalid*, *coverable in an existing configuration*, or *coverable in a new configuration*. The function `cover` processes an interaction by trying to cover it in the sample. It takes four parameters, a single interaction i, the current sample $\mathcal{S}$, the feature model $\mathcal{M}$, and a MIG $\mathcal{G}$. To demonstrate the concept of using a covering strategy, we first describe our basic covering strategy, which is similar to the established algorithm IPOG for combinatorial interaction coverage [Lei et al., 2007]. However, in YASA, we enable the usage of a customized covering strategy, which can affect the sampling time and sample size. Thus, we enable the fine-tuning of the sampling process to specific needs. In the implementation we use in our thesis, we employ an advanced covering strategy that is based on the basic covering strategy, but has been mostly

optimized for decreasing sampling time. We further describe this advanced covering strategy and its optimization compared to the basic covering strategy in Section 4.4.1.

We depict our basic covering strategy in pseudo code in Algorithm 4.2. First, we check whether there exists a configuration c in the sample $\mathcal{S}$ that already covers the given interaction i (Line 3). In that case, we do not modify the sample and return, continuing with the next interaction to cover (Line 12). Otherwise, we check if i is valid by computing whether it conflicts with the feature model using a SAT solver (Line 4). If i is invalid, we return without modifying the sample (Line 5). In contrast, if i is valid, we iterate through each configuration c in $\mathcal{S}$ and check whether c would still be satisfiable when adding the literals from i (Line 8). In case that we find such a configuration c, we add the literals of i to c and return $\mathcal{S}$ with the modified configuration in it (Line 10). If there is no configuration in $\mathcal{S}$ that is able to cover i, we create a new configuration c_i containing only the literals from i and add this new configuration to $\mathcal{S}$ (Lines 11, 12).

The basic covering strategy is able to process any given interaction and either cover it in a configuration of the sample or determine that it is invalid. When we apply this covering strategy to every interaction in $\vec{\mathcal{I}}$, we compute a complete sample with 100% interaction coverage.

4.3.4 Resampling the Sample

The *resampling process* aims to reduce the sample size of an already computed sample. It consists of two steps, trimming and recomputing the sample. Within the function `trim`, we take a complete sample and remove certain configurations from it. Then, we complete the now partial sample by calling the function `sample` again to covering any missing interactions.

To determine which configurations to remove, we compute a score for each configuration in the current sample by considering their number of uniquely covered interactions. In detail, we use the following formula to give every configuration $c \in \mathcal{S}$ a score:

$$score(c, \mathcal{S}) = \frac{|\{i \in \mathcal{L}(\mathcal{F}^*)^t \mid i \subseteq c, \forall c' \in \mathcal{S} \setminus \{c\} : i \not\subseteq c'\}|}{|c|^t}$$

For each configuration $c \in \mathcal{S}$ we count the number of interactions that are only covered by this configuration and none other. As the sample may contain partial configurations, we also factor in the number of defined features for each configuration. A configuration that defines more features has a higher chance of covering an interaction. Thus, for each configuration, we divide its unique interaction count by its number of defined features to the power of t. After calculating a score for every configuration in $\mathcal{S}$, we compute the arithmetic mean of all scores. Then, we remove all configurations that have a score less than this mean value.

After trimming the sample, we repeat the sampling process to complete the sample. For every repetition of the sampling process, we use a different random ordering for iterating over the interactions. During our experiments, we found that having a

different ordering every time helps in finding a good distribution for the interactions within the configuration sample. However, recomputing the sample means that it is also possible for the sample size to increase instead of decrease. To solve this problem, we always remember the smallest sample achieved so far and return it in the end. In case there are more than one smallest samples, we return the first one we found.

After we apply the resampling process as often as specified by the resampling limit parameter, we get a refined sample. Because this latest sample may still contain some partial configurations, we finish the sampling process by calling the function `autocomplete`, which then computes the final sample.

4.4 Optimization and Adaptation of YASA

In Section 4.3, we present the general sampling process of YASA. This sampling process produces a complete sample and can be fine-tuned using its five parameters. However, we aim to make YASA even more adaptable to other application scenarios, in which certain properties of a sample, such as the similarities or order of configurations in the sample, are important. For instance, this can be the case for regression testing or measuring non-functional properties of a configurable system. To this end, we choose a modular architecture that lets us alter two strategies for building the sample, the covering strategy and the auto-completion strategy. For this, we must replace the implementation of the functions `cover` and `autocomplete`, respectively. In the following, we present alternative strategies for both functions.

4.4.1 Advanced Covering Strategy

A modification of the covering strategy can affect the sampling time and properties of the resulting sample, such as sample size. For instance, a covering strategy may attempt to complete as many configurations as soon as possible in the sample or may try to balance the size of each configuration in a sample. In our thesis, we aim for an efficient covering strategy. Thus, here, we propose an advanced covering strategy that is based on the basic covering strategy, we presented earlier. In the following, we highlight all the changes we made compared to the basic covering strategy. In Algorithm 4.3, we show our advanced strategy in pseudo code.

The most potential for decreasing sampling time lies in the check for validity of an interaction and the check whether a configuration is still satisfiable when including a given interaction. For both, we have to check the validity of partial configurations by solving a SAT instance, which is an expensive operation. Therefore, one optimization is to restructure the order of operations. We move the more expensive operations to the end of the algorithm, because it is sometimes possible to cover an interaction before needing to check its validity using a SAT solver. In addition, we create a filtered sample $\mathcal{S}'$ by filtering all configurations that clearly contradict the given interaction, because they contain at least one complementary literal (Line 3). Thus, we do not iterate through every configuration, which reduces the amount of SAT instances we need to solve.

Algorithm 4.3 Advanced covering strategy of YASA

Require:
- i – Interaction
- $\mathcal{S}$ – Configuration Sample
- $\mathcal{M}$ – Feature Model
- $\mathcal{G}$ – MIG for $\mathcal{M}$
- $\mathcal{C}_{History}$ – Configuration History

Return:
- $\mathcal{S}$ – configuration sample

1: **function** COVER(i, $\mathcal{S}$, $\mathcal{M}$, $\mathcal{G}$)
2: **if** $\nexists c \in \mathcal{S} : i \in c$ **then**
3: $\mathcal{S}' \leftarrow \{c \in \mathcal{S} \mid c \cap \{\neg l \mid l \in i\} = \emptyset\}$
4: **for** $c \in \mathcal{S}'$ **do**
5: **if** isValid_NoSAT($c \cup i$, $\mathcal{C}_{History}$) **then**
6: $\mathcal{S} \leftarrow (\mathcal{S} \setminus \{c\}) \cup \{c \cup i\}$
7: **return** $\mathcal{S}$
8: **if** $\neg$ isValid_NoSAT(i, $\mathcal{C}_{History}$) **then**
9: **if** isInvalid_MIG(i, $\mathcal{G}$) **then**
10: **return** $\mathcal{S}$
11: **if** $\neg$ isValid_SAT(i, $\mathcal{M}$, $\mathcal{C}_{History}$) **then**
12: **return** $\mathcal{S}$
13: **for** $c \in \mathcal{S}'$ **do**
14: $c_i \leftarrow c \cup i$
15: **if** $\neg$ isInvalid_MIG(c_i, $\mathcal{G}$) **then**
16: **if** isValid_SAT(c_i, $\mathcal{M}$, $\mathcal{C}_{History}$) **then**
17: $\mathcal{S} \leftarrow (\mathcal{S} \setminus \{c\}) \cup \{c_i\}$
18: **return** $\mathcal{S}$
19: $c_i \leftarrow i$
20: $c_i \leftarrow$ decisionPropagation(c_i, $\mathcal{M}$, $\mathcal{G}$)
21: $\mathcal{S} \leftarrow \mathcal{S} \cup \{c_i\}$
22: **return** $\mathcal{S}$

23: **function** isInvalid_MIG(c, $\mathcal{G}$)
24: $\mathcal{G}' \leftarrow$ updateGraph($\mathcal{G}$, c)
25: **return** $\exists p \in \mathcal{P}^S(\mathcal{G}) : p = ((v, ...), ..., (.., \neg v))$

26: **function** isValid_NoSAT(c, $\mathcal{C}_{History}$)
27: **return** $\exists c' \in \mathcal{C}_{History} : c \subset c'$

28: **function** isValid_SAT(c, $\mathcal{M}$, $\mathcal{C}_{History}$)
29: $c_{solution} \leftarrow$ SAT($\mathcal{M}$, c)
30: **if** $c_{solution} = \emptyset$ **then**
31: **return** *false*
32: **else**
33: $\mathcal{C}_{History} \leftarrow \mathcal{C}_{History} \cup c_{solution}$
34: **return** *true*

Storing a Configuration History As another optimization, we store a configuration history $\mathcal{C}_{History}$ that contains configurations computed by any SAT solver query. Whenever we find that a SAT instance is solvable, the corresponding solver will return a matching configuration, which we add to our history (Line 33). We use the configuration history in the new function `isValid_NoSAT` (Lines 26–27). This function takes a partial configuration c and return either *true*, if there is a configuration $c' \in \mathcal{C}_{History}$ that is a super set of c or *false*, if no such configuration exists in $\mathcal{C}_{History}$. If a partial configuration or an interaction is unsatisfiable the function will always return *false*, otherwise it may return *true*. We use this function two times in the advanced covering strategy. First, we iterate over every configuration c in the current sample $\mathcal{S}$ and check whether `isValid_NoSAT` return *true* for the combination of c and the current interaction i (Lines 4–5). If such a configuration c exists, we add the literals of i to c, update c in $\mathcal{S}$, and return (Lines 6–7). Second, before determining the validity of an interaction with a SAT solver, we first use `isValid_NoSAT` to check whether it occurs in any configuration in $\mathcal{C}_{History}$ (Line 8). In that case, we can skip the SAT solver call (Line 11).

Using a MIG In order to avoid even more SAT instances, we employ a MIG to check whether a given interaction is invalid. The function `isInvalid_MIG` takes a partial configuration c and a MIG $\mathcal{G}$ and returns *true* if c leads to a contradiction in $\mathcal{G}$ (Lines 23–25). For this, the function first creates an updated MIG version $\mathcal{G}'$ using c (cf. Section 3.3.1). Then, it checks whether $\mathcal{G}'$ contains a strong path leading from a vertex v to the complementary vertex $\neg v$. In that case, c must be unsatisfiable and the function returns *true*. If no contradiction can be found, then the function returns *false*. If a partial configuration or an interaction is satisfiable, the function will always return *false*, otherwise it may return *true*. We use the function `isInvalid_MIG` two times. First, we try to determine whether an interaction is invalid, before using a SAT solver (Line 9). Second, we try to determine whether a configuration including the current interaction leads to a contradiction before checking it with a SAT solver (Line 15).

Applying Decision Propagation Yet another optimization for reducing the number of SAT instances is to increase the amount of already covered interactions. We try to increase this number by applying decision propagation on every new configuration. That means that we compute the literals that are implied by the literals already contained in a new partial configuration. As decision propagation is itself an expensive operation we only apply it to new configurations, instead of applying it whenever a configuration is modified. Furthermore, we use a lightweight version of decision propagation that only relies on a MIG and does not use a SAT solver. Thus, this lightweight version does not determine every implied literal, but only a subset (i.e., strongly connected vertices) (cf. Section 3.3.1). Nevertheless, it helps to complete new partial configurations and consequently reduces the number of SAT instances for checking the validity of partial configurations and interactions.

4.4.2 Alternative Completion Strategies

The automated completion of partial configurations in the final sample is controlled by the chosen completion strategy. Employing a specific completion strategy can be useful to affect the properties of a sample. For instance, a sample may contain rather similar or dissimilar configurations depending on this strategy. In YASA, we enable the following completion strategies: *default*, *random*, *all-yes*, *all-no*, or *none*. The default strategy uses a SAT solver to compute a complete configuration using a partial one. All partial configurations in the sample are satisfiable (cf. Section 2.1), thus for each configuration $c \in \mathcal{S}$ a SAT solver computes a complete configuration c' that is a super set of c. The actual values assigned to currently undefined features are dependent on the internal behavior of the employed SAT solver. Although this strategy uses a SAT solver call per partial configuration, it is relatively fast. Finding a satisfying assignment for a given satisfiable partial configuration is more efficient than computing a valid configuration for a feature model, because the partial configuration already defines values, which satisfy some clauses of the feature model formula. Furthermore, if we use a configuration history in our covering strategy, we can also use it here to find already computed configurations that can complete a given partial configuration, and thus save a SAT solver call. Thus, this is the recommended strategy, if the the concrete configurations in the sample do not matter for the application scenario.

Another strategy is to randomly assign values to each partial configuration. This method is similar to the default method, as we use a modified SAT solver to compute a random configuration given a partial configuration as input. Although the modified SAT solver creates a random configuration, the result is still dependent on its internal behavior. Therefore, the configurations that are randomly generated in this way are not uniformly distributed over the valid problem space. Still, the final sample contains much more dissimilar configurations than with the default strategy and is thus suited for application scenarios that require most dissimilar configurations within a sample (e.g., for regression testing). Another possible strategy would be to employ a solver able to produce uniformly distributed random configurations. However, currently, running these solvers is more computationally expensive and would increase the sampling time significantly.

The strategies all-yes and all-no try to complete a partial configuration by selecting as many or as few features as possible, respectively. This strategy is comparable with solving a Max-SAT or Min-SAT problem for each partial configuration, although we do not compute an exact solution of these problems, but only an approximation. The execution time of both is similar to the random strategy. Depending on the application scenario, it can be useful to either create a sample that represents products with a small code base (e.g., for faster compile times) or products with a larger code base (e.g., for testing).

Another possible strategy is to disable auto-completion entirely. This is the fastest method, as the sample is returned as is, potentially containing partial configurations. This strategy can be useful if the application scenario does not require complete configurations. We also uses this method when we sample multiple times for multiple feature subsets, as we described above (cf. Section 4.2).

4.5 Evaluation

With YASA we aim to provide an efficient and flexible sampling algorithm that scales even to large systems. Thus, we test how YASA behaves for highly-configurable systems. Further, we evaluate YASA by comparing it with other t-wise sampling algorithms and itself with different parameter settings with regard to the following metrics:

- Sample size (i.e., the number of configurations in a sample)
- Sampling time (i.e., the time required to compute a sample)

In particular, we aim to answer the following research questions:

RQ_1 Does the choice of a sampling algorithm affect the sampling time?

RQ_2 Does the choice of a sampling algorithm affect the sample size?

RQ_3 Is there a limit for the scalability of YASA?

We want to investigate, whether our advanced covering strategy in YASA affects the sample size compared to other approaches, as our hypothesis is that our approach is at least as efficient as other sampling strategies. Even more, we hypothesize that we can achieve a similar sample size as other approaches with shorter sampling time, and thus increasing sampling efficiency.

4.5.1 Setup of Experiments

In our evaluation for YASA, we run two experiments. First, we compare YASA to other state-of-the-art t-wise interaction sampling algorithms by computing samples for multiple configurable systems. We use the data of this experiment to answer our first two research questions. Second, we use YASA to sample a set of highly configurable systems. With this experiment we aim to answer our third research question. In the following, we describe the setup for our experiments. First, we present the subject systems, for which we generate samples. Second, we introduce the algorithms, which we compare against each other. Finally, we describe our measuring methods for our two evaluation criteria, sampling time and sample size.

Subject Systems

We use multiple subject systems in our evaluation from different sources with varying sizes. In Table 4.1, we list the systems together with their number of features. In detail, we use 11 small to middle-sized systems from FeatureIDE, which have between 27 and 366 features. Using the tool Kclause [Oh et al., 2019], we extracted feature models from 5 real-world systems that employ Kconfig as a configuration tool. These feature models range from 71 to 1,018 features. Furthermore, we use a selection of 39 feature models from subsystem of the eCos system provided by Knüppel et al. [Knüppel et al., 2017] ranging from 1,165 to 1,267 features. In addition, we use a feature model for FreeBSD with 1,396 features and for the Linux kernel with 6,888 features provided by She et al. [She et al., 2011]. Finally, we use the feature models of three real-world systems in different versions provided by Pett et al. in their 2019 SPLC sampling challenge [Pett et al., 2019].

Table 4.1: Subject systems for the evaluation of YASA

Source	System Name	#Features
FeatureIDE Example	gpl	27
FeatureIDE Example	dell	46
FeatureIDE Example	berkeleyDB1	53
FeatureIDE Example	SmartHome22	60
FeatureIDE Example	violet	88
FeatureIDE Example	berkeleyDB2	99
FeatureIDE Example	BattleofTanks	144
FeatureIDE Example	BankingSoftware	176
FeatureIDE Example	eShopFIDE	192
FeatureIDE Example	eShopSplot	287
FeatureIDE Example	DMIE	366
Kconfig Project	fiasco	71
Kconfig Project	axtls	95
Kconfig Project	uclibc-ng	270
Kconfig Project	toybox	323
Kconfig Project	busybox-1_29_2	1,018
Knüppel et al.	eCos Subsystems	1,165 – 1,267
She et al.	FreeBSD 8.0.0	1,396
She et al.	Linux Kernel 2.6.28	6,889
Pett et al.	FinancialServices01 (10 Versions)	1,001 – 1,148
Pett et al.	Automotive02_V1	14,010
Pett et al.	Linux_2013-11-06	49,247

Algorithms

We use several state-of-the-art algorithms for t-wise interaction sampling as comparison, which were also used in previous evaluations [Al-Hajjaji et al., 2016a, 2019; Johansen et al., 2012a], namely *Chvátal* [Chvatal, 1979], *ICPL* [Johansen et al., 2011, 2012a], and *IncLing* [Al-Hajjaji et al., 2016a]. The implementation of these algorithms is provided within the FeatureIDE framework [Al-Hajjaji et al., 2016b; Meinicke et al., 2017], which we employ in our evaluation. Consequently, we implemented YASA in Java and integrated it into the FeatureIDE library as well. Within the implementation of YASA, we employ the SAT solver *Sat4J* [Le Berre and Parrain, 2010] to check for validity of configurations and interactions.

Not all algorithms support all values for t. IncLing is designed as a strict pair-wise interaction coverage algorithm, and thus only works for $t = 2$. ICPL supports values for t up to 3 and Chvátal up to 4. In theory, we can run our algorithm with any value for t. However, due the restrictions of the other algorithms for the parameter t, in our experiments, we use $t = 2$.

Regarding YASA, we use the following parameters. We test multiple values for the resampling limit m to evaluate its impact. In particular, we choose the values 1, 5,

and 10. In order to ensure a fair comparison of the sample sizes, we neither use an initial sample nor any feature subsets. As covering strategy, we use our advanced covering strategy (cf. Section 4.4.1).

In summary, we compare results from the following algorithms:

Short Name	Name	Supported t
Chvatal	*Chvátal* [Chvatal, 1979]	$1 \leq t \leq 4$
ICPL	*ICPL* [Johansen et al., 2011, 2012a]	$1 \leq t \leq 3$
IncLing	*IncLing* [Al-Hajjaji et al., 2016a]	$t = 2$
YASA_1	*YASA* with $m = 1$	$t \geq 1$
YASA_5	*YASA* with $m = 5$	$t \geq 1$
YASA_10	*YASA* with $m = 10$	$t \geq 1$

Evaluation System

For a fair comparison, we run all algorithms for our first experiment on the same hard- and software:

- *CPU*: Intel(R) Core(TM) i5-8350U
- *Physical Memory*: 16 GB
- *JVM Max Memory*: Xmx: 14 GB
- *OS*: Manjaro (Arch Linux)
- *JVM*: OpenJDK 1.8.0_222

For our second experiment, we use the following system:

- *CPU*: AMD Ryzen 7 3700X
- *Physical Memory*: 32 GB
- *JVM Max Memory*: Xmx: 16 GB
- *OS*: Manjaro (Arch Linux)
- *JVM*: OpenJDK 14.0.1

Measurement Process

Sample Size Regarding the sample size, we count the number of configurations in each sample computed by each algorithm. To this end, we count partial and complete configurations within a sample.

Sampling Time For measuring sampling time, we take the time that is needed for generating a sample with each algorithm. To be precise, we use the Java method `System.nanoTime()` to get a timestamp in nanoseconds directly before calling a sampling algorithm and use the same method to get a timestamp directly after the algorithm returns. We then compute the difference between both timestamps and derive the elapsed time in milliseconds. Additionally, we set a timeout of 24 hours for every sampling process. If a sampling algorithms fails to return within this period, we cancel the computation.

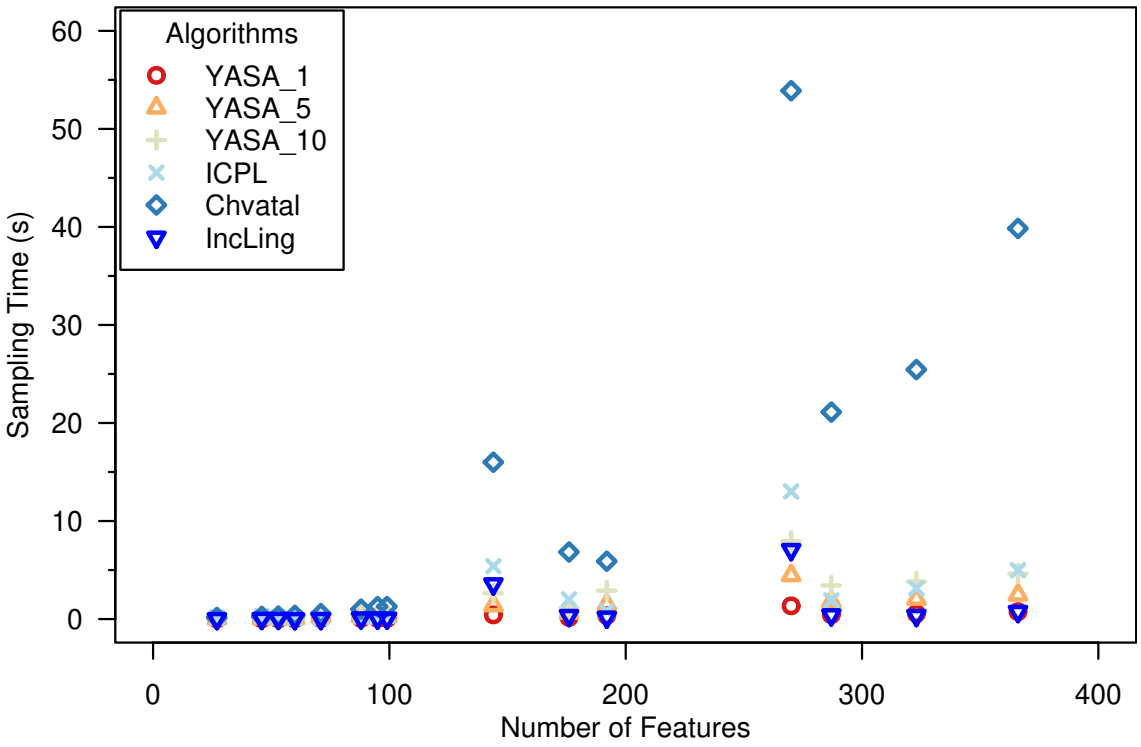

Figure 4.2: Sampling times for systems with a number of features less than 1,000

Memory Consumption Measuring the precise memory consumption of a JVM is more complex than measuring execution time. We measure memory consumption using the Java command `Runtime.getRuntime().totalMemory()`, which returns the currently allocated memory of the JVM. This is different from the actual used memory and can be seen as an upper bound. In detail, we called the method every second and stored the highest value returned. This gives us a fairly accurate approximation on an upper bound of used memory of the JVM.

4.5.2 Results of Experiments

We structure our findings according to our two experiments. For each experiment, we analyze and discuss our results and answer the corresponding research questions.

Experiment 1 (Comparing Sampling Times and Sample Sizes)

In Figure 4.2 and 4.3, we depict the absolute sampling time for all systems for $t = 2$. On the y-axis we show the sampling time in seconds (s). We relate the values to the number of features within each feature model, which is shown at the x-axis. In order to achieve a better scaling, we present the values in two diagrams with different scales for the y-axis. In Figure 4.2, we show all values for feature models with less than 1,000 features and in Figure 4.3 all values for feature model with more than 1,000 features. Note that, the data in Figure 4.3 is plotted on a logarithmic scale. Additionally, we show the result for *Linux* in Table 4.2.

For small systems the sampling time of most algorithms lies below 10 s for our setup. An exception is Chvátal, which takes up to 60 s for some feature models. For larger

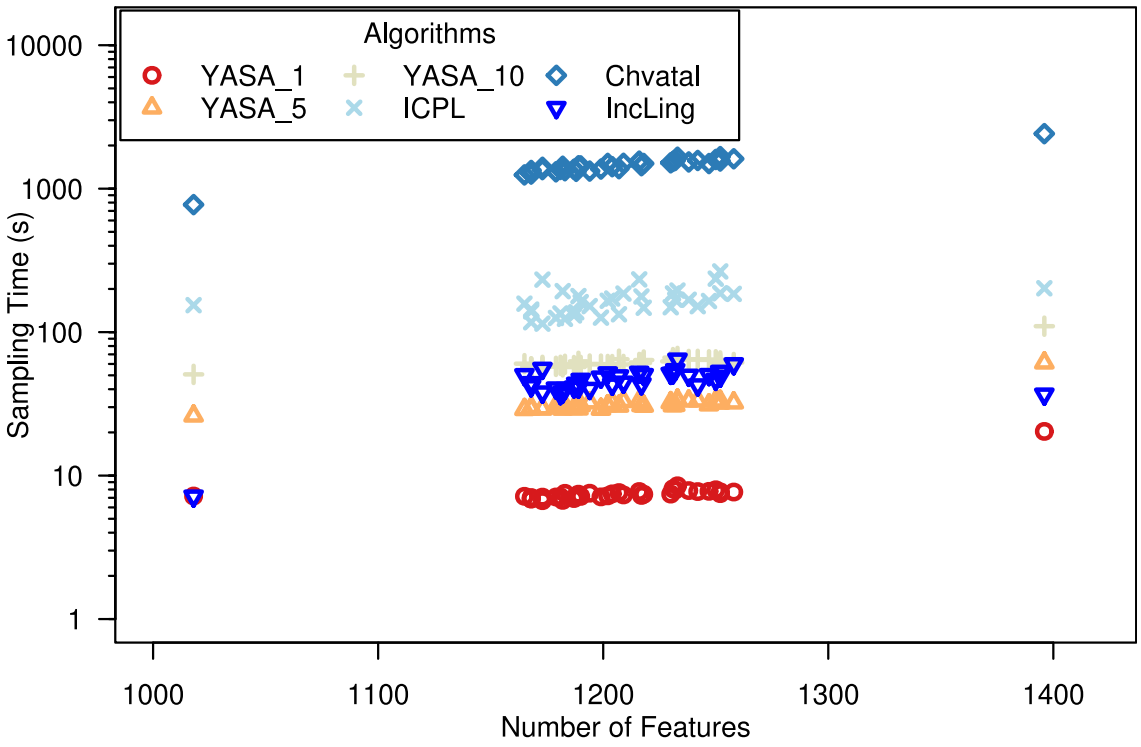

Figure 4.3: Sampling times for systems with a number of features larger than 1,000

Table 4.2: Results for *Linux 2.6.28* with $t = 2$

System Name	Algorithm	Sample Size	Sampling Time
Linux 2.6.28	YASA_1	545	36m 21s
Linux 2.6.28	YASA_5	489	2h 23m 03s
Linux 2.6.28	YASA_10	487	4h 43m 20s
Linux 2.6.28	ICPL	482	4h 00m 14s
Linux 2.6.28	Chvatal	–	*timeout*
Linux 2.6.28	IncLing	781	1h 50m 39s

systems we can see a clearer distinction of the sampling times. Chvátal is the slowest algorithm for all feature models with sampling times between 774 s and 2,418 s. ICPL is the second slowest for almost all models, but is already substantially faster with sampling times ranging from 114 s to 264 s. YASA_10 requires between 50 s and 109 s of sampling time. YASA_5 (26 – 61 s) and IncLing (7 – 64 s) have relatively similar sampling times for most models, while YASA_1 is distinctively faster with sampling times ranging from 6 s to 20 s.

In Figure 4.4, we show the sampling times for YASA_5, YASA_10, ICPL, Chvátal, and IncLing relative to YASA_1 (i.e., YASA_1 is 100%). With this figure, we visualize the differences in runtime for the different algorithms. Note that, the data is plotted on a logarithmic scale. While there are some cases where YASA_1 is outperformed by IncLing, on average all other algorithms require more sampling time. Using the Mann-Whitney test, we can see that this is significant with p-values all $< 10^{-7}$ (i.e., comparing YASA_1 with all other algorithms). Chvátal requires

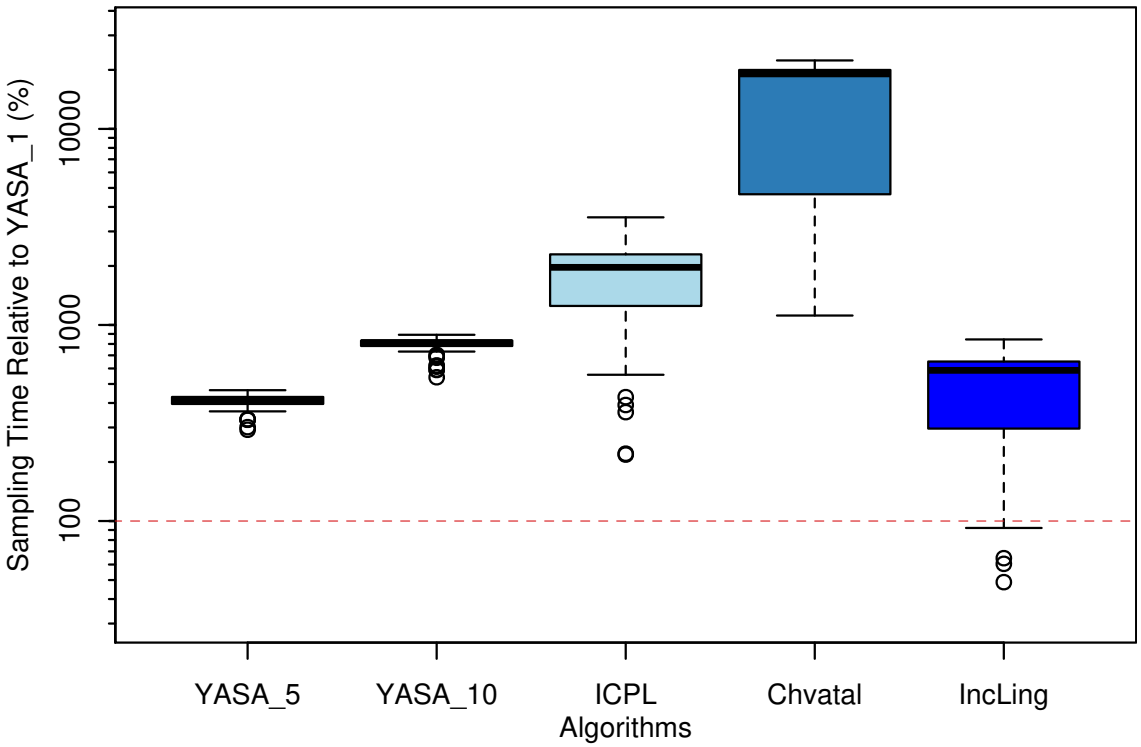

Figure 4.4: Aggregated sampling times relative to YASA_1 ($m = 1$)

the most sampling time and, on average, takes more than 200 times longer than YASA_1. ICPL is much faster than Chvátal, but still, on average, takes about 20 times longer than YASA_1. The sampling time of IncLing is close to YASA_5, but is still out performed by YASA_1 by factor 5. We can see that the parameter m has a clear influence on the sampling time. The higher the value of m, the higher the sampling time.

RQ_1 While with YASA we mostly aimed for a more efficient sampling, we also were able to produce relatively small samples for most systems. Thus, we can answer our research questions, as follows. Regarding RQ_1, we found out that our approach is at least as efficient as other state-of-the-art algorithms and can even outperform them. This however, depends on the value of parameter m. We are able to lower the sampling time significantly by decreasing the parameter m, although this has a negative effect on the sample size. In theory, by dividing the feature set into smaller subsets we could decreases the sampling time even further. Nevertheless, we showed that our approach has indeed the potential to efficiently generate samples.

We depict the sample size for most systems in Figure 4.5 and 4.6. On the y-axis we show the sample size. Again, we show the number of features of each feature model on the x-axis. Analogous to the sampling time, in Figure 4.2 we show all values for feature models with less than 1,000 features and in Figure 4.3 all values for feature models with more than 1,000 features. We depict the sample size for *Linux* in Table 4.2.

From both diagrams we can see that the sample size varies greatly for every system and also for the different algorithms. However, for larger systems we can also see a

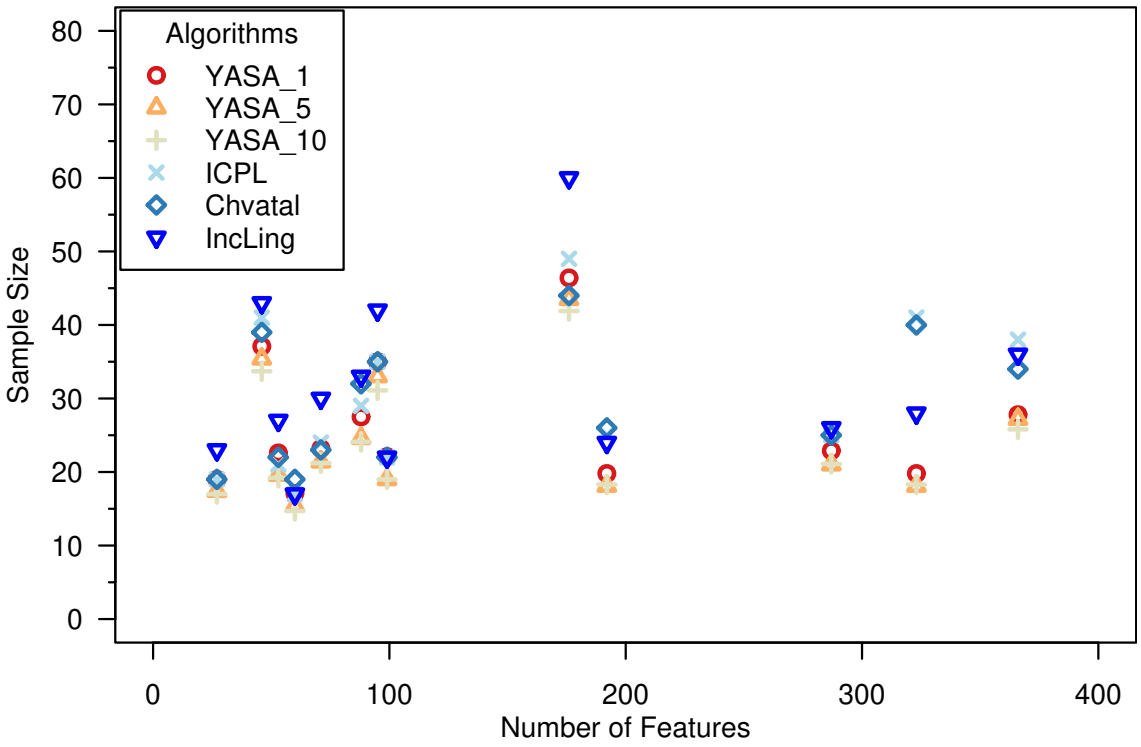

Figure 4.5: Sample size for systems with a number of features less than 1,000

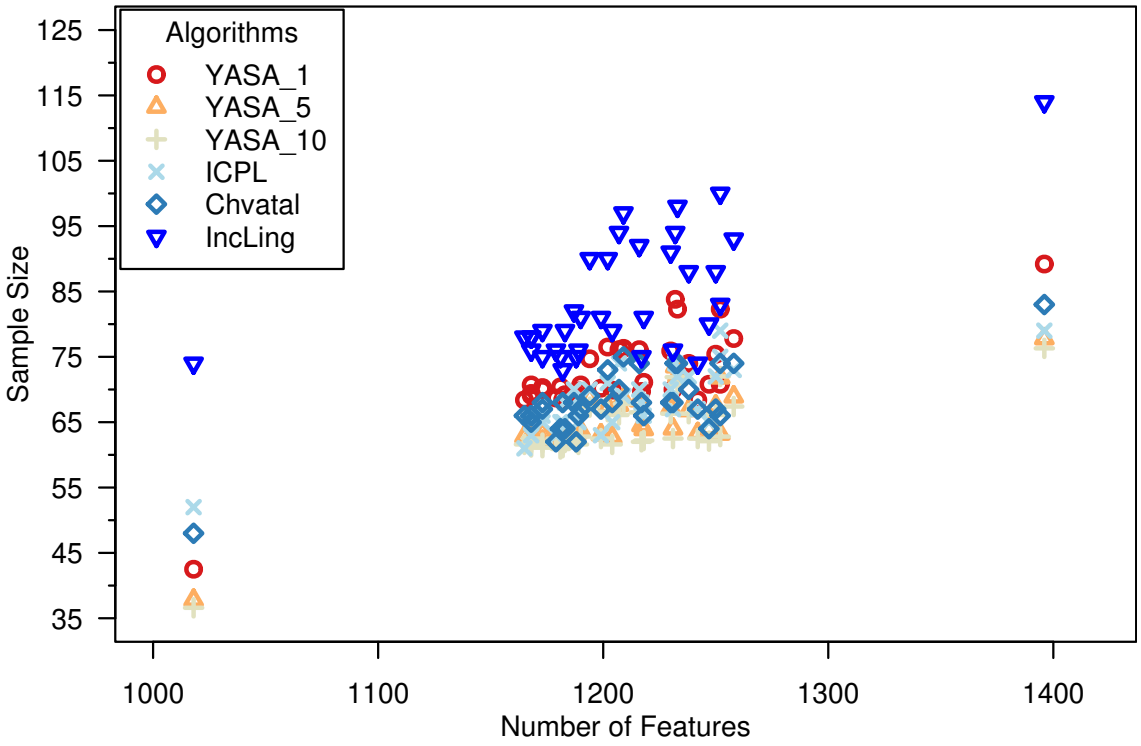

Figure 4.6: Sample size for systems with a number of features larger than 1,000

correlation between sample size and the number of features throughout all algorithms. Surprisingly, YASA_10, and even YASA_5, produce mostly smaller samples than

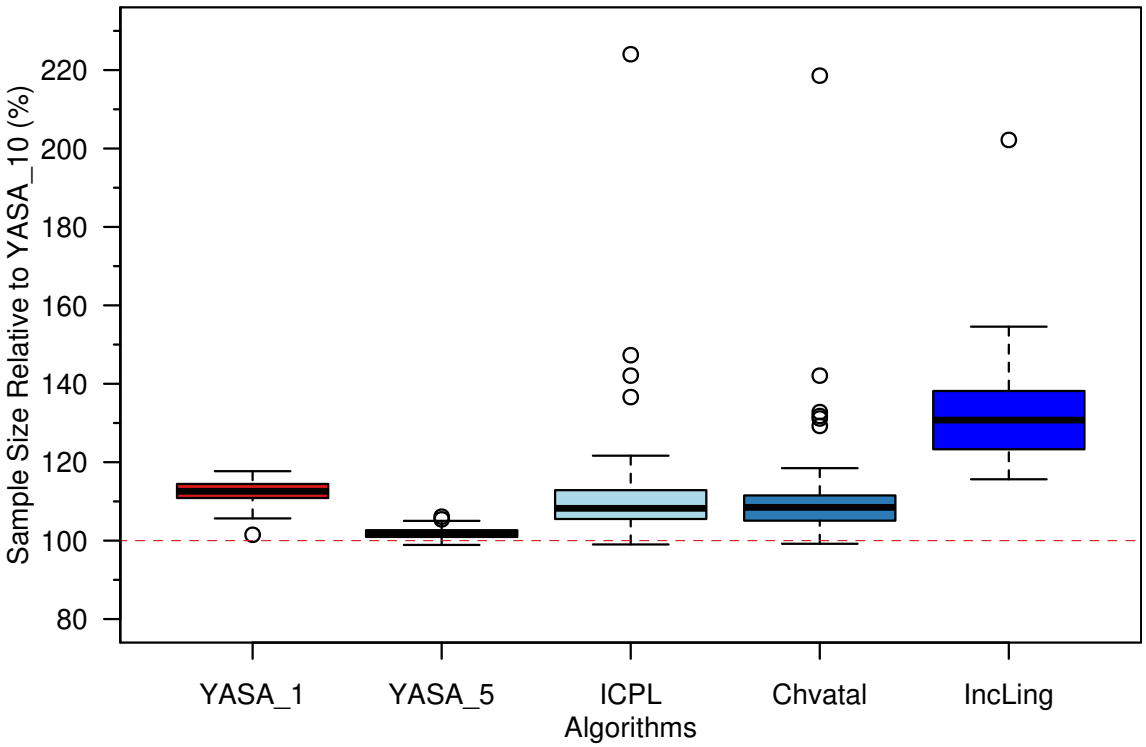

Figure 4.7: Aggregated sample size relative to YASA_10 ($m = 10$)

all other algorithms. When applying the Mann-Whitney test, we find that this is significant with p-values all $< 10^{-8}$ (i.e., comparing YASA_5 and YASA_10 with all other algorithms).

We visualize the sample size of all algorithms relative to YASA_10 in Figure 4.7. In the diagram, we can see by how much the samples of YASA_10 are smaller compared to all other algorithms. Both, ICPL and Chvátal generate samples that are on average about 10% larger than the samples from YASA_10. We can see a larger difference for IncLing, which generates samples that are on average about 30% larger. As expected, an increase of the value for m decreases the sample size of YASA. However, the average sample size of YASA_5 is close to the average sample size of YASA_10, confirming that a further increase of m yields only little benefit.

RQ_2 Regarding RQ_2, we found that we are able to produce smaller samples compared to other tested algorithms. Again, this depends on the value of parameter m. Having a low value for m increases the size of the sample, but increasing m up to a certain point can decrease the sample size by at least 10% compared to $m = 1$.

Experiment 2 (Scaling to Highly Configurable Systems)

In Table 4.3, we show our results for creating pair-wise samples for the system *FinancialServices01*. For this system, we encountered no scalability issues during the sampling process and were able to create samples for all versions of this system. Both, sampling time and memory consumption seem feasible for the practical usage of YASA. On the other hand, the sample sizes seem to be quite high and might be

Table 4.3: Results for *FinancialServices01* with $t = 2$

Version	Sample Size	Time (s)	Memory (MB)
2017-05-22	402	12	125
2017-09-28	3,193	45	209
2017-10-20	3,190	46	197
2017-11-20	3,184	46	281
2017-12-22	3,188	48	281
2018-01-23	3,184	47	281
2018-02-20	3,647	64	528
2018-03-26	3,559	72	528
2018-04-23	4,422	81	528
2018-05-09	4,405	79	528

too large for testing. However, it is difficult to reason about the feasibility of the sample size as this is highly depended on the actual test suite and the testing process for an SPL.

In addition to the feature models of *FinancialServices01*, we also attempt to sample the systems *Automotive02_V1* and *Linux_2013-11-06*. Unfortunately, the sampling time for both systems was too high, which is why we aborted the sampling process for both. However, we can still would report some insights on the challenges during sampling process of these two systems and reason about potential approaches for generating samples faster.

The main problem with both feature models is their high number of features (i.e., 14,010 and 49,247, respectively). As we already discussed, this leads to a high number of interactions and more computationally expensive SAT instances. For example, creating a pair-wise sample for *Automotive02_V1* requires to consider 392,532,180 interactions. In addition, due to the large SAT instance for such a large-scale feature model, we were only able to handle about 400 interactions per second. In comparison, in the latest version of *FinancialServices01*, we were able to handle about 20,000 interactions per second. After five days, we aborted the generation of a sample for *Automotive02_V1*, which was about 50% done (i.e., 50% of interactions were processed). At this point, the sample contained 2 complete and 23,015 partial configurations. The algorithm would have eventually terminated, but probably would have run for at least 5 more days.

For Linux, this problem is even worse, since the number of features in the feature model is artificially increased. This is due to the usage of the Tseytin transformation [Tseytin, 1983] to transform the given feature model into a conjunctive normal form (CNF). A CNF is necessary for most SAT solvers. However, computing a CNF can be an expensive operation in terms of both, memory consumption and transformation time, because the number of CNF clauses that are created from applying the law of distribution can grow exponentially for some formulas. To this end, techniques such as the Tseytin transformation allow to create an equisatisfiable CNF by introducing new auxiliary variables, which prevents an exponential growth

of clauses. We tried to tackle this problem for Linux by using a feature subset that omits all auxiliary variables for the construction of the interactions, as these are not present in an actual product, and thus cannot interact. Auxiliary variables for the Linux feature model can be easily identified in the corresponding Dimcas file, as they have no associated name in the comments section. With this feature subset, we are able to reduce the number of features from 49,248 down to 14,781. However, although we are able to reduce the number of interactions, we still need to take the auxiliary variables into account for every SAT instance. Thus, we were only able to handle about 200 interactions per second for this particular model, which again makes the sampling time infeasible.

RQ$_3$ To answer RQ$_3$, currently, we see a limit for the scalability of YASA regarding its sampling time for feature models with more than 10,000 features. At the moment, we think that a promising solution could be to further reduce the size of the feature set in the feature model by applying domain knowledge to reason about the interactions between certain features. It may also help to facilitate SAT instances by reducing the size of a CNF by applying feature slicing [Krieter et al., 2016].

4.5.3 Threats to Validity

The results of our evaluation and their generalizability may be influenced by some threats to validity. In the following, we list possible internal or external threats to validity and explain how we attempt to mitigate any potential biases.

Internal

Due to the long execution times of some algorithms, we were only able to conduct our experiments with a single repetition. Thus, our results for the sampling time from these experiments may vary, if we would repeat the them. Given the absolute sampling times, we measured in our experiments, especially for the larger feature models, we argue that any small difference in the sampling time does not invalidate our overall findings.

As we implemented YASA and our evolution environment by ourselves, there is a chance that we might have faults in the implementation that may bias our results. To mitigate the risk of a faulty implementation, we use automated unit tests to check the samples computed by any algorithm in our evolution. In particular, we ensure that all computed samples reach a t-wise coverage of 100% and contain only valid configurations.

External

In practice, there exist many different feature models of all complexities and size. In our evaluation, we are only using a comparably small number of different feature models. Thus, the results could be different for other systems, which may hamper the generalizability of our results. To mitigate this potential bias, we reused multiple real-world feature models from different sources. In addition, these feature models come from different domains and highly differ in their number of features and constraints, which makes them quite diverse. By using this diverse set of feature model, we expect to see varying results, which are sufficient to infer general trends.

A similar issues, concerns the number of evaluated sampling algorithms. In total, we included four different algorithms in our evaluation. However, there exist many more algorithms for performing t-wise interaction sampling. We have chosen these algorithms, because of their good availability and reputation in the research field. Thus, we are confident that the algorithms we run in our experiment reflect the current state-of-the-art and are well suited for a comparison.

4.6 Summary

In this chapter, we presented our approach YASA for efficiently building a flexible t-wise sample using an automated configuration process. We explained the details of the sampling process and how a sample is created for a given feature model. We explained all parameters of YASA that can affect the sample properties and the sampling time. Further, we discussed multiple optimizations and possibilities to adapt YASA in order to decreasing sampling time or sample size. Within YASA, we use our data structure MIG, which we introduced in Chapter 3, to increase the performance of finding invalid interactions during the sampling process. In our evaluation, we made a comparison with other sampling algorithms and demonstrated the current limits of our approach. We find that YASA is able to outperform all state-of-the-art algorithms in most cases, in terms of sampling time and sample size, depending on its parameter settings. We showed that we can use the parameters YASA to either reduce the sample size or its sampling time, whichever is more suitable for a given scenario. However, although we find YASA to be efficient for most feature models, we also demonstrated that it still has limitations regarding its scalability, as for large system with more than 10,00 feature variables its sampling time can be infeasible. In conclusion, we developed an adaptable and efficient t-wise sampling algorithm that improves upon the state-of-the-art algorithms. We further extend and use YASA in the following Chapter 5.

5. Presence-Condition Sampling

This chapter introduces the concept of presence conditions coverage and extends the sampling algorithm from the previous chapter. It shares material with the following publication: T-Wise Presence Condition Coverage and Sampling for Configurable Systems (arXiv 2022) *[Krieter et al., 2022].*

In this chapter, we propose a new coverage criterion *t-wise presence condition coverage* with which we aim to facilitate the creation of effective samples for product-based testing in an automatic configuration process. To this end, we combine t-wise interaction coverage with presence condition coverage of implementation artifacts to create a coverage criterion that increases the fault-detection rate of a sample. For this new coverage criterion, we present an extension of our sampling algorithm YASA from Chapter 4 that enables *t-wise presence condition sampling*, which generates samples with a 100% *t-wise presence condition coverage* for a given configurable system and a value of t. This sampling algorithm works independently from the employed variability mechanisms, such as preprocessors, build systems, or plug-in systems.

We structure this chapter as follows. First, we give a motivation on t-wise presence condition coverage by providing a problem statement on the current limitations of t-wise interaction coverage (see Section 5.1). Second, we describe the details of t-wise presence condition coverage (see Section 5.2). Third, we present a sampling algorithm for achieving t-wise presence condition coverage for a given system (see Section 5.3). Fourth, we investigate the testing effectiveness, testing efficiency, and sampling efficiency of our sampling algorithm and coverage criterion compared to traditional t-wise interaction sampling algorithms (see Section 5.4).

5.1 Motivation

Testing is an important task in software engineering to detect faults and to check intended behavior [Ammann and Offutt, 2016; McGregor, 2010]. However, exhaustive testing may be impossible and binds resources that could be used in other phases of development. This is especially an issue when testing highly configurable systems.

Using t-wise interaction sampling, developers can ensure that all interactions of at most t features (e.g., all selected, none selected, only one selected, etc.) are contained in at least one configuration in the generated sample and can thus be tested. A property of t-wise interaction sampling is that this automatic configuration process works purely on the problem space of a configurable system. Thus, it is a black-box approach that does not take into account the mapping between features and actual implementation artifacts, such as source code, models, and test cases. This can lead to some issues. First, a traditional t-wise interaction sampling algorithm may create samples containing configurations that result in similar products, when build. Having many similar products can lower the overall testing efficiency of a sample. Second, t-wise interaction sampling may consider some irrelevant feature combinations, which do not interact in the implementation at all. This can yield an unnecessarily large sample, which again leads to a lower testing efficiency. Third, t-wise interaction sampling may not be sufficient to reliably detect a fault resulting from an interaction of a degree larger than the provided value for t. Typically, t-wise interaction sampling does not scale well for higher values of t, as the number of possible feature interactions grows exponentially with t. Thus, developers are incentivized to choose a low value for t, such as $t = 2$ or $t = 3$, in order to keep the testing effort feasible. However, some faults may require a feature interaction of a high degree to be included in a configuration, and thus may not be reliably included in a sample, which potentially decreases testing effectiveness. In summary, due to the black-box nature of t-wise interaction sampling, it may not reveal certain faults resulting from feature interaction beyond t or requires too many configurations to even reach a certain code coverage or fault-detection rate.

There exist some approaches that attempt to address the mentioned issues. Tartler et al. [2014] propose *statement coverage*, a white-box approach that considers *presence conditions* (i.e., the selection of features for which an artifact is included in a product) to ensure that every artifact is present in at least one configuration in the sample and gets a chance of being tested. However, simply including an implementation artifact in a product does not guarantee that is indeed being tested. Therefore, Ruland et al. [2018] argue that, in addition to including each artifact, at least one test case must also cover each artifact in order to properly test it. While the approach of Tartler et al. [2014] can generate a relatively small sample, the approach of Ruland et al. [2018] produces a relatively large sample, but guarantees that each artifact will indeed be tested. We aim to find a reasonable trade-off between both approaches by building on the idea of Tartler et al. [2014] and try to increase testing effectiveness by combining it with t-wise interaction sampling. In particular, we propose the coverage criterion *t-wise presence condition coverage*, which combines t-wise interaction coverage with presence condition coverage of implementation artifacts. Rather than counting only interactions between features, our new criterion considers interactions of presence conditions of implementation artifacts.

```
 ⋮
053  #include "libbb.h"
054  #include <syslog.h>
055
056  #if GET || PUT // G ∨ P
 ⋮
621  # if TFTP // T
622  int tftp_main(int argc, char **argv) {
 ⋮
625  #   if BLOCKSIZE // B
626    const char *blksize_str=TFTP_BLKSIZE_DEFAULT;
 ⋮
649    int blksize = tftp_blksize_check(blksize_str, 65564);
650    if (blksize < 0)
651      return EXIT_FAILURE;
652  #   endif
 ⋮
670  #   if DEBUG // D
671    printf("blksize = %d\n", blksize); // changed
672  #   endif
 ⋮
690  }
691  # endif
 ⋮
827  #endif
```

Listing 5.1 Excerpt of the file `tftp.c` adopted from `BusyBox`

In the following, we use an example to describe the potential problems with current approaches for t-wise interaction sampling and motivate our solution. In Listing 5.1, we show a slightly edited code snippet from the system `BusyBox`, which uses the C preprocessor [Stallman and Weinberg, 1987] to implement its variability [Liebig et al., 2010]. The example is taken from the file `tftp.c`, which handles client-server communication via tftp. We show an excerpt of the corresponding `BusyBox` feature model in Figure 4.1. The code snippet contains five features from the feature model, `TFTP` (T), `GET` (G), `PUT` (P), `BLOCKSIZE` (B), and `DEBUG` (D), which each can be set to either *true* or *false*. The other features of the feature model do not appear in the code snippet. In comparison to the original code, we changed a statement in Line 671 such that a compilation error occurs for certain products. The variable `blksize` is declared in Line 626, which is dependent on the feature B. Then, `blksize` is used in Line 671, which is dependent on the feature D. Thus, if D is selected in a configuration, but B is not, the generated product will be syntactically incorrect.

Efficiency of Sample-Based Testing In product-based testing, we run the all test cases of a system once per sampled configuration. To this end, the total testing effort is dependent on the number of tested configuration of a systems. This is because the

variant for each configuration must be generated, complied, and run against all of its test cases. Although the execution time and number of the test cases may differ from configuration to configuration, in general, there exists a linear correlation between the number of configurations in a sample (i.e., the sample size) and the total testing effort. If the sample size increases, the overall testing effort increases as well. Thus, analogous to other research [Ruland et al., 2018; Varshosaz et al., 2018], we consider the testing of smaller samples to be more efficient (i.e., *testing efficiency*). Applying pair-wise interaction sampling to our running example considers every interaction between two features. For instance, in Listing 5.1, for the features P and G, all four possible interactions are considered. However, the three interactions $(P, \neg G)$, $(\neg P, G)$, (P, G) all lead to the same product, as all of them satisfy the expression in Line 056. Thus, it is sufficient to consider only two interactions (e.g., $(\neg P, \neg G)$ and (P, G)) for this code snippet. In the table below, we show the sample generated from the t-wise interaction sampling algorithm ICPL [Johansen et al., 2011, 2012a], which consists of six configurations:

Feature	**Configurations**					
	c_1	c_2	c_3	c_4	c_5	c_6
`T`	×	✓	×	✓	✓	×
`G`	×	✓	✓	×	✓	×
`P`	×	✓	×	×	✓	✓
`D`	×	×	✓	×	✓	✓
`B`	×	✓	✓	×	×	✓

All four interactions of P and G are included. Including such unnecessary interactions can lead to a larger sample, and thus to a lower testing efficiency.

Effectiveness of Sample-Based Testing We consider a sample to be more effective, the more faults could be detected using its derived products. Analogous to other research [Ruland et al., 2018; Varshosaz et al., 2018], we refer to this as *testing effectiveness*. Although the fault in Listing 5.1 apparently involves only two features, it is in fact a feature interaction of degree four. To actually generate a product that contains the error, the corresponding configuration must have the feature B deselected and the features T, D, and G or P selected. Below, we show the sample generated by the pair-wise sampling algorithm IncLing [Al-Hajjaji et al., 2016a]:

Feature	**Configurations**						
	c_1	c_2	c_3	c_4	c_5	c_6	c_7
T	✓	×	✓	×	×	×	✓
G	✓	×	×	✓	×	×	✓
P	✓	×	×	✓	×	✓	×
D	✓	×	×	×	✓	✓	×
B	✓	×	✓	×	✓	×	×

Although the interaction $(D, \neg B)$ is covered in configuration c_6, the actual fault will not be included in the product, as the feature T is not selected and the preprocessor will remove the entire code block. As a result this sample has a lower testing effectiveness than the previous sample. In particular, this sample is not sufficient to find the fault in Listing 5.1.

5.2 Presence Condition Coverage

In the following, we define our coverage criterion for *t-wise presence condition coverage*, based on the presence conditions and feature model of a configurable system and a value for the parameter t. To this end, we describe interactions between presence conditions and how our coverage criterion is calculated for a given sample and t value.

5.2.1 Presence Condition Interactions

An interaction between two presence conditions (i.e., between their implementation artifacts) is similar to interactions between features. However, the key difference is that a presence condition can be an arbitrary propositional formula. In a complete configuration, a feature can either be selected, if its corresponding positive literal is included in the configuration, or deselected, if its negative literal is included. In contrast, a presence condition may be active or inactive for several different literal combinations (cf. Section 2.2.2). Thus, an interaction between t presence conditions must include all possible union sets of these literal combinations.

Inactive Presence Conditions

In order to account for the interactions between *present* and *absent* implementation artifacts, we must consider the interactions between *active* and *inactive* presence conditions. For this reason, we construct the complement of each presence condition, which is itself a presence condition and represents all literal combinations, for which the original presence condition is inactive. To this end, we negate the formula of a presence condition and convert it back into a DNF. As an example, consider the presence condition $\phi(\mathcal{P}^+_{626}) = (G \wedge T \wedge B) \vee (P \wedge T \wedge B)$ for Line 626 of Listing 5.1. We process this formula as follows:

$$
\begin{aligned}
\phi(\mathcal{P}^-_{626}) &= \neg\, \phi(\mathcal{P}^+_{626}) \\
&= \neg((G \wedge T \wedge B) \vee (P \wedge T \wedge B)) \\
_{De\ Morgan's\ Law} &\equiv (\neg G \vee \neg T \vee \neg B) \wedge (\neg P \vee \neg T \vee \neg B) \\
_{Distributive\ Law} &\equiv (\neg G \wedge \neg P) \vee (\neg G \wedge \neg T) \vee (\neg G \wedge \neg B) \\
&\quad \vee (\neg T \wedge \neg P) \vee (\neg T \wedge \neg T) \vee (\neg T \wedge \neg B) \\
&\quad \vee (\neg B \wedge \neg P) \vee (\neg B \wedge \neg T) \vee (\neg B \wedge \neg B) \\
_{simplify} &\equiv (\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B) \\
\Rightarrow \qquad \mathcal{P}^-_{626} &= \{\{\neg G, \neg P\}, \{\neg T\}, \{\neg B\}\}
\end{aligned}
$$

After negating the formula, we apply De Morgan's law to get a CNF and then apply the distributive law to convert it back into a DNF. From the new presence condition

$\mathcal{P}^-_{626}$, we see that if a configuration either contains the literal $\neg T$, $\neg B$, or both, $\neg G$ and $\neg P$, the resulting product will not include Line 626.

For a given system, we construct a single set $\mathcal{PC}$ that contains all presence conditions from all lines of the system and their corresponding complements. To construct all t-wise interactions for a system, we can then generate all t-wise combinations of $\mathcal{PC}$ by constructing the Cartesian product for the given t (i.e., $\mathcal{PC}^t$).

Combined Presence Condition

For each interaction of t presence conditions, we can build a new combined presence condition $\mathcal{P}^*$ that is satisfied if and only if all individual presence conditions are active or inactive, respectively. To this end, we conjoin the DNFs of all presence conditions involved in a given interaction and convert the resulting expression back to a DNF. For each presence condition that is active in the given interaction we use its original DNF and for each presence condition that is inactive we use its complementary DNF. For instance, consider the DNFs for *excluding* Line 626 ($\mathcal{P}^-_{626}$) and *including* Line 671 ($\mathcal{P}^+_{671}$) of Listing 5.1:

$$\begin{aligned} \text{Line 626:} \quad & \phi(\mathcal{P}^-_{626}) = (\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B) \\ \text{Line 671:} \quad & \phi(\mathcal{P}^+_{671}) = (G \wedge T \wedge D) \vee (P \wedge T \wedge D) \end{aligned}$$

We build the combined presence condition for the interaction (i.e., $\mathcal{P}^*$) by conjoining the literals of each pair-wise clause combination:

$$\begin{aligned} \phi(\mathcal{P}^*) &= \phi(\mathcal{P}^-_{626}) \wedge \phi(\mathcal{P}^+_{671}) \\ \equiv_{\textit{Distributive Law}} &((\neg G \wedge \neg P) \wedge (G \wedge T \wedge D)) \vee ((\neg G \wedge \neg P) \wedge (P \wedge T \wedge D)) \vee \\ &((\neg T) \wedge (G \wedge T \wedge D)) \vee ((\neg T) \wedge (P \wedge T \wedge D)) \\ &((\neg B) \wedge (G \wedge T \wedge D)) \vee ((\neg B) \wedge (P \wedge T \wedge D)) \\ \equiv_{simplify} &(\neg B \wedge G \wedge T \wedge D) \vee (\neg B \wedge P \wedge T \wedge D) \\ \Rightarrow \quad & \mathcal{P}^* = \{\{\neg B, G, T, D\}, \{\neg B, P, T, D\}\} \end{aligned}$$

After merging, we further simplify the combined DNF by removing contradictions and redundant clauses. In case of our example, this results in the simplified DNF $(\neg B \wedge G \wedge T \wedge D) \vee (\neg B \wedge P \wedge T \wedge D)$. Thus, if a configuration either contains the literals $\neg B$, G, T, and D or $\neg B$, G, T, and P, Line 671 will appear in the resulting product, but not Line 626.

5.2.2 Presence Condition Coverage Criterion

We introduce the coverage criterion *t-wise presence condition coverage* based on the coverage of presence condition interactions. Given a set of presence conditions $\mathcal{PC}$, a value for t, the feature model $\mathcal{M}$ of a configurable system, and a configuration sample $\mathcal{S}$ for the same system, we can determine a value for our coverage criterion t-wise presence condition coverage. To this end, we define the function $\mathcal{I}$ that returns the set of all presence condition interactions covered by a given set of configurations:

$$\mathcal{I}(t, \mathcal{PC}, \mathcal{M}, \mathcal{S}) = \{I \in \mathcal{PC}^t \mid \exists c \in \mathcal{S},\ \forall \mathcal{P} \in I : \mathit{satisfiable}(c, \mathcal{M}) \wedge \mathit{active}(\mathcal{P}, c)\}$$

We define t-wise presence condition coverage as the ratio between the number of t-wise presence condition interactions that are covered by $\mathcal{S}$ and the number of valid t-wise presence condition interactions for $\mathcal{M}$.

$$coverage(t, \mathcal{PC}, \mathcal{M}, \mathcal{S}) = \frac{|\mathcal{I}(t, \mathcal{PC}, \mathcal{M}, \mathcal{S})|}{|\mathcal{I}(t, \mathcal{PC}, \mathcal{M}, \mathcal{C}(\mathcal{M}))|}$$

We can apply the t-wise presence condition coverage criterion to evaluate a sample computed by any sampling algorithm. The higher the degree of t-wise presence condition coverage of a sample, the more likely is the sample to contain a faulty interaction, which can be detected by a test case. As this coverage criterion takes into account the solution space by considering interactions between concrete implementation artifacts, we expect that it is more suited as indicator for the testing effectiveness of a sample than regular interaction coverage. Thus, given a value for t, a sample that achieves 100% t-wise presence condition coverage can potentially detect more faults than a sample that achieves 100% t-wise interaction coverage, but not a 100% t-wise presence condition coverage. Naturally, with *t-wise presence condition sampling* we aim to compute a sample that achieves a coverage of 100%.

In our running example, there are five different presence conditions and their five complements.

$true$,	$false$,
$(G) \vee (P)$,	$(\neg G \wedge \neg P)$,
$(G \wedge T) \vee (P \wedge T)$,	$(\neg G \wedge \neg P) \vee (\neg T)$,
$(G \wedge T \wedge B) \vee (P \wedge T \wedge B)$,	$(\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B)$,
$(G \wedge T \wedge D) \vee (P \wedge T \wedge D)$,	$(\neg G \wedge \neg P) \vee (\neg T) \vee (\neg D)$

In total, there are 11 valid combinations. Considering the sample from IncLing in Section 5.1, there are two interactions that are not covered by any configuration in the sample: $(G \wedge T \wedge B \wedge \neg D) \vee (P \wedge T \wedge B \wedge \neg D)$ and $(G \wedge T \wedge \neg B \wedge D) \vee (P \wedge T \wedge \neg B \wedge D)$. Thus, the number of interactions covered by the sample is 9, which results in a pair-wise presence condition coverage of approximately 82%.

5.3 T-Wise Presence Condition Sampling

Based on our new coverage criterion t-wise presence condition coverage, we introduce a sampling algorithm that enables the generation of samples that achieve 100% t-wise presence condition coverage. To this end, we extend our t-wise sampling algorithm YASA from Chapter 4. In total, we make three major changes to YASA in order to enable it to cope with presence conditions and t-wise presence condition coverage. First, we add two additional steps before running the actual sampling algorithm. In these steps, we *extract a list of presence conditions* from the source code of a given configurable system and *preprocess this list* to prepare it as input for our sampling algorithm. Second, we adapt the creation of the list of interactions $\vec{\mathcal{I}}$, as we need to *build the combined presence condition* for each interaction. Third, we adapt the advanced covering strategy to enable *covering presence conditions* (i.e., arbitrary propositional expressions) in a sample. Note that, running our extended sampling

algorithm with $t = 1$ will yield a sample that achieves 100% of the coverage criterion of Tartler et al. [2014]. Every variable code block will be included by at least one configuration in the sample. In addition, every variable code block will also be excluded by at least one configuration in the sample, which is not a requirement of the original coverage criterion. Thus, in terms of testing effectiveness, one-wise presence conditions sampling will perform at least as good or even better than the approach of Tartler et al. [2014].

5.3.1 Extracting Presence Conditions

As a first step, we have to extract a set of presence conditions from the target system. This step is highly dependent on the specifics of the given system, such as the programming language, the variability mechanism, and the configuration mechanism. In our thesis, we focus on the handling of extracted presence conditions, and therefore do not discuss a general approach for the extraction process. In fact, both processes, extracting presence conditions and testing products, are independent from our sampling approach and can be adapted to other variability mechanism, programming language, and testing framework. We did however implement an algorithm for extracting presence conditions from systems that use the languages C/C++ and the C preprocessor as a variability mechanism. The algorithm is based on the tool PCLocator [Kuiter et al., 2018], which is able to parse C files and analyze preprocessor annotations. The basic procedure from this algorithm would also work for other preprocessor implementations, such as Antenna for Java, but would of course require a different parser implementation. In the following, we describe the general logic behind the algorithm. We give some more details for our particular implementation in our prototype YASA in Section 5.4.1.

In general, the algorithm parses a C file and determines a presence condition for each line. As a result, the algorithm returns a list of presence conditions for each file. The C preprocessor annotations that are used to implement a variability mechanism always form a block around variable code artifacts. Each block begins with an annotation, such as `#if` or `#ifdef` and ends with an annotation, such as `#else` or `#endif`. The first annotation specifies the condition for which the code block is included in the final product. Annotated blocks can be nested, as we illustrate in our example in Listing 5.1. Consequently, for each line in a C file, the algorithm determines all C preprocessor blocks that surround the line. This is done by pushing each annotation that begins a block on a stack and removing the latest pushed annotation whenever an annotation is encountered that ends a block. Thus, for each line the stack contains all beginning annotations from its surrounding blocks. The presence condition of the line is then constructed by the conjunction of the conditionals of the surrounding blocks.

Regarding our example in Listing 5.1, we extract the following list of presence conditions (for brevity, we only show presence conditions from the lines visible in the code snippet):

Line	Formula
053	*true*
054	*true*
055	*true*
056	$G \vee P$
621	$(G \vee P) \wedge T$
622	$(G \vee P) \wedge T$
625	$(G \vee P) \wedge T \wedge B$
626	$(G \vee P) \wedge T \wedge B$
649	$(G \vee P) \wedge T \wedge B$

Line	Formula
650	$(G \vee P) \wedge T \wedge B$
651	$(G \vee P) \wedge T \wedge B$
652	$(G \vee P) \wedge T \wedge B$
670	$(G \vee P) \wedge T \wedge D$
671	$(G \vee P) \wedge T \wedge D$
672	$(G \vee P) \wedge T \wedge D$
690	$(G \vee P) \wedge T$
691	$(G \vee P) \wedge T$
827	$G \vee P$

5.3.2 Preprocessing Presence Conditions

The result from our extraction algorithm is a list of arbitrary propositional formulas. To prepare the set of presence conditions $\mathcal{PC}$ that we use as input for our sampling algorithm, we need to further refine this list. In particular, we need to remove all duplicates and convert each propositional formula to DNF. We perform the preprocessing of the list in three steps. First, we make a syntactical comparisons of the original formulas to remove all identical formulas. This step is of low computational effort and filters out all syntactic duplicates in the list. Naturally, there may be formulas in the list that are syntactically different but are semantically equivalent. Second, in order to filter out more possibly redundant presence conditions and for a simpler sampling process later on, we convert all remaining formulas to DNF. For the conversion to DNF, we use an algorithm based on standard boolean algebra. Note that we do not need to convert a presence condition from CNF to DNF, but directly convert the propositional formula extracted from the source code, which typically contains only a small set of features and no complicated formulas. Third, we again make a syntactical comparisons of all DNFs to remove identical formulas. The syntactical comparison we use in this step is indifferent to different orderings of literals and clauses within a DNF. For instance, the formula $(G \wedge T) \vee (P \wedge T)$ is equal to the formula $(T \wedge P) \vee (T \wedge G)$. Thus, we are likely to find and remove most redundant formulas in the set. Nevertheless, it is still possible that two semantically equivalent formulas have different DNFs. For instance, $(G \wedge T) \vee (P \wedge T) \vee (T)$ is equal to (T) when simplified. However, a complete semantical comparison requires much more computational effort and is unlikely to reduce the current set much further. As a result of the preprocessing, we get a set of DNFs $\mathcal{PC}$ representing all presence conditions of the target system.

The preprocessed list of presence conditions for our example in Listing 5.1 is the following

First Line	Original Formula	DNF Formula
053	*true*	*true*
056	$G \vee P$	$(G) \vee (P)$
621	$(G \vee P) \wedge T$	$(G \wedge T) \vee (P \wedge T)$
625	$(G \vee P) \wedge T \wedge B$	$(G \wedge T \wedge B) \vee (P \wedge T \wedge B)$
670	$(G \vee P) \wedge T \wedge D$	$(G \wedge T \wedge B) \vee (P \wedge T \wedge D)$

Algorithm 5.1 Main sampling algorithm of YASA adapted for presence conditions sampling

Require:

$\mathcal{M}$ – Feature Model
t – Interaction Size
$\mathcal{S}_{init}$ – Initial Sample (*Default*: $\mathcal{S}_{init} = \emptyset$)
$\mathcal{PC}$ – Presence Conditions (*Default*: $\mathcal{PC} = \{\{f\} \mid f \in \mathcal{F}(\mathcal{M})\}$)
m – Resampling Limit (*Default*: $m = 1$)

Return:

$\mathcal{S}$ – Configuration Sample

1: **function** CREATESAMPLE($\mathcal{M}$, t, $\mathcal{S}_{init}$, $\mathcal{F}^*$, m)
2: $\mathcal{S} \leftarrow \mathcal{S}_{init}$
3: $\mathcal{G} \leftarrow$ CREATEMIG($\mathcal{M}$)
4: $\mathcal{PC}^{+/-} \leftarrow$ GETEXPRESSIONS($\mathcal{PC}$, $\mathcal{G}$)
5: $\mathcal{C}_{History} \leftarrow \emptyset$
6: $\mathcal{S} \leftarrow$ SAMPLE($\mathcal{M}$, $\mathcal{G}$ t, $\mathcal{S}$, $\mathcal{PC}^{+/-}$, $\mathcal{C}_{History}$)
7: **for** 1 **to** $m - 1$ **do**
8: $\mathcal{S} \leftarrow$ TRIM($\mathcal{S}$)
9: $\mathcal{S} \leftarrow$ SAMPLE($\mathcal{M}$, $\mathcal{G}$ t, $\mathcal{S}$, $\mathcal{PC}^{+/-}$, $\mathcal{C}_{History}$)
10: $\mathcal{S} \leftarrow$ AUTOCOMPLETE($\mathcal{S}$)
11: **return** $\mathcal{S}$

12: **function** SAMPLE($\mathcal{M}$, $\mathcal{G}$ t, $\mathcal{S}$, $\mathcal{PC}^{+/-}$, $\mathcal{C}_{History}$)
13: $\vec{\mathcal{P}}^* \leftarrow$ COMBINE($\mathcal{PC}^{+/-}$, t)
14: **for** $\mathcal{P}^* \in \vec{\mathcal{P}}^*$ **do**
15: $\mathcal{S} \leftarrow$ COVER($\mathcal{P}^*, \mathcal{S}, \mathcal{M}, \mathcal{G}, \mathcal{C}_{History}$)
16: **return** $\mathcal{S}$

Splitting our algorithm in a preprocessing and a sampling part allows us to flexibly determine the input for the actual sampling algorithm. Similar to the the feature subsets used in the regular YASA (cf. Section 4.2.4), we can also build subsets of $\mathcal{PC}$ and only consider interactions between these. To this end, we run the sampling algorithm once for each subset and iteratively add configurations to $\mathcal{S}$. Using such subsets, we can, for example, group presence conditions for single files or folders of a system, and thereby limit the number of possible interactions and the resulting sample size.

5.3.3 Building the List of Combined Presence Conditions

After preparing the set of presence conditions, we start with constructing the sample. We show the general algorithm for t-wise presence condition sampling in Algorithm 5.1, which is an extension of our sampling algorithm YASA from Chapter 4. In this chapter, we only go into detail on the modifications we did to the original algorithm (cf. Section 4.3).

The basic algorithm differs from the regular YASA mainly in the parameter $\mathcal{PC}$, which replaces the feature subset $\mathcal{F}*$. $\mathcal{PC}$ contains the set of presence conditions in DNF.

Similar to the set of literals $\mathcal{L}_I$ in regular YASA, we create a list of expressions $\mathcal{PC}^{+/-}$ that represents all active and inactive presence conditions (cf. Section 5.2.1). For this, we call the function `getExpressions` (Line 4), which simplifies the expressions in $\mathcal{PC}$, removes obvious tautologies and contradictions, and computes the complement of each presence condition. With the help of the MIG $\mathcal{G}$, we remove all core and dead features variables from the clauses in DNF. A DNF clause that contains the positive literal of a dead feature variable or the negative literal of a core feature variable always evaluates to *false*, and thus, we can remove the entire clause from the DNF. If the resulting DNF is empty, then the entire DNF is a contradiction and the corresponding presence condition is always inactive. Thus, we do not need to consider an interaction with other presences conditions, analogous to core and dead feature variables (cf. Section 4.3.2). Similarly, if a DNF clause contains the negative literal of a dead feature variable or the positive literal of a core feature variable this literal always evaluates to *true*, and thus, we can remove the literal from the clause. If the resulting clause is empty, then the entire DNF is a tautology and the corresponding presence condition is always active. Again, in this case, we do not need to consider an interaction with other presences conditions. Based on this reasoning, we simplify all DNFs and remove any tautologies or contradictions, we find. Afterwards, we compute the complements of all presence conditions in $\mathcal{PC}$ to account for interactions with inactive presence conditions (cf. Section 5.2.1).

Regarding our example in Listing 5.1, the resulting set $\mathcal{PC}^{+/-}$ contains the following formulas:

DNF	**Complement**
$(G) \vee (P)$	$(\neg G \wedge \neg P)$
$(G \wedge T) \vee (P \wedge T)$	$(\neg G \wedge \neg P) \vee (\neg T)$
$(G \wedge T \wedge B) \vee (P \wedge T \wedge B)$	$(\neg G \wedge \neg P) \vee (\neg T) \vee (\neg B)$
$(G \wedge T \wedge B) \vee (P \wedge T \wedge D)$	$(\neg G \wedge \neg P) \vee (\neg T) \vee (\neg D)$

The remaining algorithm is again similar to YASA. We start with the initial sample $\mathcal{S}_{init}$ (Line 2) and then iterate over all t-wise interactions one at a time (Lines 14–15). For each, we either add a new partial configuration to the sample or the literals of one clause of the interaction's presence condition to an existing configuration. To this end, we compute the combined presence condition $\mathcal{P}^*$ of an interaction as outlined in Section 5.2.1 (Line 13) and use it in our covering strategy. Due to the nature of this algorithm, we are guaranteed to check every possible combination of the given presence conditions exactly once. For each combination of presence conditions, we either include it in at least one configuration in the sample or determine that it cannot be covered by any valid configuration. Since there are finitely many possible combinations (i.e., $2^t \cdot \binom{n}{t}$ with n being the number of presence conditions), the algorithm will eventually terminate and guarantees that a presence condition coverage of 100% is achieved by the computed sample. This, of course, assumes that the system executing the algorithm has enough memory and can run for as long as necessary.

Algorithm 5.2 Advanced presence condition covering strategy for YASA

Require:
$\mathcal{P}^*$ – Combined Presence Condition
$\mathcal{S}$ – Configuration Sample
$\mathcal{M}$ – Feature Model
$\mathcal{G}$ – MIG for $\mathcal{M}$
$\mathcal{C}_{History}$ – Configuration History
Return:
$\mathcal{S}$ – Configuration Sample

```
1: function COVER(P*, S, M, G, C_History)
2:     if ∄c ∈ S : ∃d ∈ P* : d ⊆ c then
3:         for d ∈ P* do
4:             S' ← {c ∈ S | ∄l ∈ d : ¬l ∈ c)}
5:             for c ∈ S' do
6:                 if isValid_NoSAT(c ∪ d, C_History) then
7:                     S ← (S \ {c}) ∪ {c ∪ d}
8:                     return S
9:         P' ← ∅
10:        for d ∈ P* do
11:            if isValid_NoSAT(i, C_History) then
12:                P' ← P' ∪ {d}
13:            else if ¬ isInvalid_MIG(i, G) then
14:                if isValid_SAT(i, M, C_History) then
15:                    P' ← P' ∪ {d}
16:        if P' = ∅ then
17:            return S
18:        for d ∈ P' do
19:            for c ∈ S' do
20:                c_i ← c ∪ d
21:                if ¬ isInvalid_MIG(c_i, G) then
22:                    if isValid_SAT(c_i, M, C_History) then
23:                        S ← (S \ {c}) ∪ {c_i}
24:                        return S
25:        c_i ← min(P')
26:        c_i ← decisionPropagation(c_i, M, G)
27:        S ← S ∪ {c_i}
28:    return S
```

5.3.4 Covering Presence Conditions

In Algorithm 5.2, we present our modified version of the advanced covering strategy (cf. Section 4.4.1) for covering a combined presence condition $\mathcal{P}^*$. This modified covering strategy uses the same optimizations, we described before in Section 4.4.1. However, we need to adapt it to enable the handling of presence conditions in DNF. In contrast to an interaction with a single literal set, a presence condition is active if any of its clauses is satisfied. Thus, instead of considering the entire combined presence condition $\mathcal{P}^*$, we iterate over each of its clauses d (Lines 2, 3, 10, 18) .

First, we check whether any clause is already covered by at least one configuration in the sample (Line 2). If not, we check whether any configuration in $\mathcal{C}_{History}$ is able to cover the current interaction (Lines 3–8). In that case, we add the respective clause of the presence condition to the corresponding configuration in the sample and return. Otherwise, we check the validity of all clauses in $\mathcal{P}^*$ and create a new filtered set $\mathcal{P}'$ that only contains valid clauses (Lines 10–15). We use the filtered set to iterate over all configurations in our current sample and check, whether we can add the literals of any $d \in \mathcal{P}'$ to it without causing a contradiction (Lines 10–15). If we find any such clause d and configuration c, we add all literals of d to c (i.e., covering the interaction) and continue with the next interaction. Otherwise, if we cannot find any suitable configuration for any clause in $\mathcal{P}'$, we use one clause from $\mathcal{P}'$ to build a new configuration that we add to our sample (Lines 25–28). For this, we run a stable sorting algorithm to sort all clauses in $\mathcal{P}'$ by their number of literals in ascending order and then use the first clause (i.e., with the least amount of literals) for the new configuration.

As an example, we describe some iterations of the algorithm for Listing 5.1. In the first iteration we consider the interaction of $(G \vee P)$ and $(\neg G \wedge \neg P)$, which is converted into the combined presence condition $\mathcal{P}^* = \{\{G, \neg G\}, \{P, \neg P\}\}$. As there is no configuration yet in $\mathcal{S}$, we continue with checking the validity of $\mathcal{P}^*$. Both clauses in $\mathcal{P}^*$, $d_1 = \{G, \neg G\}$ and $d_2 = \{P, \neg P\}$ are invalid and are therefore not considered for inclusion into a configuration. Thus, we continue with the next iteration. In the second iteration, we get the interaction of $(G \vee P)$ and $((G \wedge T) \vee (P \wedge T))$, which results in $\mathcal{P}^* = \{\{G, T\}, \{P, T\}\}$. There is still no configuration in $\mathcal{S}$, which may cover this interaction. Both clauses $d_1 = \{G, T\}$ and $d_2 = \{P, T\}$ are valid, but there is no configuration yet to which they could be added. Thus, both of them are added to $\mathcal{P}'$. Next, the smallest clause in $\mathcal{P}'$ is added to a new configuration c_i in $\mathcal{S}$. As both clauses have the same size, we use the first one, resulting in $\mathcal{S} = \{\{G, T\}\}$. In the third iteration, we get the combined presence condition $\mathcal{P}^* = \{\{G, T, D\}, \{P, T, D\}\}$. Both clauses are valid and can be added to the existing configuration in $\mathcal{S}$. We use the first clause, which results in $\mathcal{S} = \{\{G, T, D\}\}$. In the fourth iteration, we get the combined presence condition $\mathcal{P}^* = \{\{G, T, \neg D\}, \{P, T, \neg D\}\}$. Both clauses are valid, but conflict with the existing configuration in $\mathcal{S}$. Thus, a new configuration is added, resulting in $\mathcal{S} = \{\{G, T, D\}, \{G, T, \neg D\}\}$.

After the algorithm terminated, the complete sample $\mathcal{S}$ consists of five configurations:

Feature	**Configurations**				
	c_1	c_2	c_3	c_4	c_5
T	✓	✓	✓	×	×
G	✓	✓	✓	✓	×
P	✓	✓	✓	✓	×
D	✓	×	✓	×	✓
B	✓	✓	×	×	✓

When comparing this sample to the sample generated by IncLing in Section 5.1, we see that, in contrast, it contains a configuration that covers the fault in the example (i.e., c_3). This corresponds to an increase in testing effectiveness. Compared to the sample produced by ICPL in Section 5.1, we can see that it contains only the two real interactions of P and G, and thus requires only five configurations instead of seven, which is an increase in testing efficiency.

5.4 Evaluation

With t-wise presence condition coverage we aim to generate samples for a novel coverage criterion, which we expect to increase the chance of detecting faults in product-based testing. We are interested in the degree of testing effectiveness and testing efficiency of the t-wise presence condition coverage criterion and our algorithm for t-wise presence condition sampling. Therefore, we evaluate whether samples generated with t-wise presence condition sampling can detect more faults than samples generated with t-wise interaction sampling. We also evaluate what degree of t-wise presence condition coverage can be achieved by existing algorithms for t-wise interaction sampling. Further, we evaluate the sample size (i.e., testing efficiency) and sampling time (i.e., sampling efficiency) of our extended algorithm YASA. In summary, we aim to answer the following research questions:

- RQ_1 Does t-wise presence condition coverage indicate the fault detection potential better than t-wise interaction coverage for the same value of t?
- RQ_2 Can algorithms for t-wise interaction sampling achieve complete t-presence condition coverage?
- RQ_3 Does the usage of t-wise presence condition sampling affect the sample size (i.e., testing efficiency) compared to t-wise interaction sampling?
- RQ_4 Does the usage of t-wise presence condition sampling affect the sampling time (i.e., sampling efficiency) compared to t-wise interaction sampling?

Within our experiments, we compute several samples for different systems using our algorithm YASA and a selection of different state-of-the-art t-wise interaction sampling algorithms and compare the samples with respect to our evaluation criteria. In the following, we describe the setup for our experiments and our evaluation results. First, we introduce the algorithms that we compare against each other. Second, we present the subject systems, for which we generate samples. Third, we describe our measuring methods for our four evaluation criteria, fault detection, coverage, sample size, and sampling time. Fourth, we analyze and discuss our results. Finally, we discuss potential threats to the validity of our evaluation.

5.4.1 Setup of Experiments

Algorithms

We use several state-of-the-art algorithms for t-wise interaction sampling as comparison for testing efficiency and effectiveness, which were also used in previous evaluations [Al-Hajjaji et al., 2016a, 2019; Johansen et al., 2012a]. First, we employ Chvátal [Chvatal, 1979], ICPL [Johansen et al., 2011, 2012a], and IncLing [Al-Hajjaji et al., 2016a] as pure t-wise interaction sampling algorithms. Second, we use YASA as a pure t-wise interaction sampling algorithm, which only uses the feature model as input (YASA-FM). All of these algorithms compute complete t-wise samples for certain values of t using different methods. Third, we use a random sampling algorithm [Al-Hajjaji et al., 2016b]. Instead of aiming for a certain coverage criteria it generates a fixed number of valid random configurations. Fourth, we include the algorithm PLEDGE [Henard et al., 2014b], which does not try to achieve a certain t-wise interaction coverage, but is based on an evolutionary algorithm to optimize a sample of fixed size such that its contained configurations are as dissimilar as possible. By increasing dissimilarity, the sample's t-wise interaction coverage should also increase. Although this approach does not guarantee a complete t-wise interaction coverage, it aims to increase sampling and testing efficiency while maintaining a reasonably good testing effectiveness. Fifth, we use YASA to compute samples based on presence conditions (YASA-PC). Finally, we run YASA-PC with $t = 1$ (YASA-PC-1) to compute a sample that is comparable to the approach of Tartler et al. [2014] (cf. Section 5.3). The sample produced by this algorithm will most likely be different from a sample produced the original algorithm of Tartler et al. [2014]. However, it is guaranteed to achieve 100% the coverage criterion of Tartler et al. [2014].

Implementation Details The implementation of these algorithm is provided by multiple open-source Java libraries, which we employ in our evaluation. Chvátal and ICPL are implemented in the SPLCATool [Johansen et al., 2012a]. IncLing and Random are implemented in FeatureIDE [Al-Hajjaji et al., 2016b; Meinicke et al., 2017]. PLEDGE is implemented in a library of the same name [Henard et al., 2014b]. For all other sampling algorithms we use YASA, for which we employ our own implementation.

We extended our existing implementation of YASA for t-wise presence condition sampling. It includes an algorithm for extracting presence conditions from systems that use the *C preprocessor* and the *kbuild* build tool. Both, the extraction algorithm and the extension of YASA are written in Java and employ several other Java libraries to implement its functionality, including *FeatureIDE* [Al-Hajjaji et al., 2016b], *Sat4J* [Le Berre and Parrain, 2010], *PCLocator* [Kuiter et al., 2018], and *KClause* [Oh et al., 2019].

For parsing C files and identifying preprocessor statements, we use the tool *PCLocator* [Kuiter et al., 2018], which combines several C parsers, such as SuperC, TypeChef, and FeatureCoPP to achieve more accurate results. This tool computes a presence condition for each line in a source file. To this end, we analyze every C file (i.e., files with the file extensions `.c`, `.h`, `.cxx`, and `.hxx`) in the source directories of the target

system. For this, we exclude special directories that do not contribute to the actual implementation of the system, but contain examples, configuration logic, or header files of system libraries. As result, we get a list containing propositional formulas for each line within a C project. We use this list as input for the t-wise presence condition sampling. Thus, in our evaluation, we focus on C projects that use the C preprocessor and kbuild as build system to enable variability. During the extraction process, we warn the user, if we find presence conditions that contain features that are not on the feature model and vice versa. For our evaluation, we only consider features that we can find in both, the feature model and the source code.

Within our sampling algorithm, we use the satisfiability solver *Sat4J* [Le Berre and Parrain, 2010] to check for validity of configurations and presence conditions. Furthermore, we use *KClause* [Oh et al., 2019] to extract a feature model for C projects that use Kconfig as configuration tool.

Regarding the random sampling, we use the default random sampling algorithm of *FeatureIDE* [Al-Hajjaji et al., 2016b]. Their implementation is based on *Sat4J* as well and generates configurations by asking the satisfiability solver for a valid configuration using a randomized feature order. While this algorithm does not generate uniformly distributed random samples, as it is biased by the internal structure of the solver, it is an efficient way to generate a high number of valid configurations. Note that it is possible for this algorithm to generate a sample that contains duplicate configurations. However, considering the enormous configuration space of larger system, such duplicates are unlikely to occur.

Parameter Details As we employ a variety of sampling algorithms in our evaluation, the required parameters differ for most of them. The only common parameter for every algorithm is the feature model, which specifies the feature dependencies. Naturally, all algorithms always use the same feature model as input.

Regarding the parameter t, not all algorithms support the same values. IncLing is designed as a strict pair-wise interaction coverage algorithm, and thus only works for $t = 2$. ICPL supports values for t up to 3 and Chvátal up to 4. We can run our algorithm for t-wise presence condition sampling with any value for t. However, YASA currently has a technical limitation that allows to process only up to 2^{31} interactions. To enable a fair comparison, we set the value of t to $t = 2$ for all algorithms.

As Random and PLEDGE do not try to achieve a certain t-wise coverage, but just generate a set of valid configurations, it is not possible to set a value for t. Instead, they require to set the size of the sample in advance. In order to ensure a fair comparison, for PLEDGE, we set the sample size equal to the size of the largest sample computed by any variant of YASA (i.e., either YASA-FM, YASA-PC, or YASA-Concrete, which ever returned the largest sample). For Random, we set several sample sizes for each system, ranging from the smallest to the largest sample size produced for every system by any algorithm.

PLEDGE also requires to set a time limit for the evolutionary algorithm. We decided to compare two different limits, the maximum and minimum time that any variant

of YASA needs to compute a sample for a particular model (i.e., either YASA-FM, YASA-PC, or YASA-Concrete, which ever requires the least and most amount of time).

For YASA, we also have to specify additional parameters beside t. We are able to specify which expressions should be considered for interaction (cf. Section 5.3). Thus, we test the following settings: *YASA-FM* considers all t-wise interactions within a feature model, and thus behaves like other pure t-wise interaction sampling algorithms. *YASA-PC* considers all t-wise interactions between all presence conditions of a system. Finally, *YASA-Concrete* considers t-wise interactions between features, but only includes features that appear in at least one presence condition (i.e., concrete features).

Summary We compare results from the following algorithms:

Short Name	Name
Chvatal	Chvátal [Chvatal, 1979]
ICPL	ICPL [Johansen et al., 2011, 2012a]
IncLing	IncLing [Al-Hajjaji et al., 2016a]
PLEDGE-Min	*PLEDGE* using minimum run time [Henard et al., 2014b]
PLEDGE-Max	*PLEDGE* using maximum run time [Henard et al., 2014b]
YASA-FM	*YASA* with all features of a model (cf. Section 4.3)
YASA-PC-1	*YASA* with all presence conditions of a system for $t = 1$ (cf. Section 5.3)
YASA-PC	*YASA* with all presence conditions of a system (cf. Section 5.3)
YASA-Concrete	*YASA* with all concrete features of a system (cf. Section 4.3)
Random	*Random* [Al-Hajjaji et al., 2016b]

Subject Systems

Currently, YASA can extract presence conditions from C preprocessor statements. Thus, we selected real-world open-source systems that use the C preprocessor as a variability mechanism. In particular, we reused 21 systems from the study of Medeiros et al. [2016], which also compared different sampling algorithms in terms of testing effectiveness. However, most of these systems do not have a separate feature model, which prevents us from taking their feature dependencies into account. For this reason, we include six real-world open-source systems that use the C preprocessor and the Kconfig tool, namely, `fiasco` (latest), `axtls` (latest), `uclibc-ng` (latest), `toybox` (latest), `BusyBox` (version 1.29.2), and `Linux` (version 2.6.28). For `Linux`, we use a feature model for version 2.6.28 provided by She et al. [She et al., 2011]. For all other systems, we extracted the feature models from their Kconfig files using the tool *KClause* [Oh et al., 2019].

In Table 5.1, we provide an overview of the all systems. At the top we show the 6 systems for which we have a feature model and at the bottom the 21 systems from the study of Medeiros et al. [2016]. For each respective feature model, we show its number of features ($\#F$), concrete features ($\#CF$) (i.e., features that appear in at least one presence condition), and dependencies ($\#D$). Regarding the extracted

Table 5.1: Subject systems for the evaluation of t-wise presence condition sampling — features ($\#F$), concrete features ($\#CF$), dependencies ($\#D$), presence conditions ($\#PC$), and the number of clauses ($\#C$) and literals ($\#L$) over all presence conditions

System (Version)	Feature Model			Presence Conditions		
	$\#F$	$\#CF$	$\#D$	$\#PC$	$\#C$	$\#L$
fiasco (latest)	71	7	120	9	12	14
axtls (latest)	95	32	190	90	126	162
uclibc-ng (latest)	270	104	1,561	225	315	406
toybox (latest)	323	8	90	14	14	14
BusyBox (1.29.2)	1,018	507	997	1,020	1,475	1,975
Linux (2.6.28.6)	6,888	1,696	80,715	3,512	5,494	8,767
totem (2.17.5)	–	–	–	223	278	332
gnome-vfs (2.0.4)	–	–	–	253	313	373
irssi (0.8.15)	–	–	–	318	369	428
lua (5.2.1)	–	–	–	324	496	714
xfig (3.2.4)	–	–	–	378	802	1,969
libssh (0.5.3)	–	–	–	393	663	962
gnome-keyring (3.14.0)	–	–	–	453	539	631
lighttpd (1.4.30)	–	–	–	567	875	1,219
dia (0.97.2)	–	–	–	606	708	810
bison (2.0)	–	–	–	695	1,161	1,871
fvwm (2.4.15)	–	–	–	777	1,482	4,075
xterm (224)	–	–	–	796	1,302	1,859
cherokee (1.2.101)	–	–	–	1,128	1,589	2,077
cvs (1.11.17)	–	–	–	1,495	2,491	3,785
gnuplot (4.6.1)	–	–	–	1,546	2,720	4,145
libpng (1.5.14)	–	–	–	1,752	3,937	7,421
apache (2.4.3)	–	–	–	1,814	2,915	4,360
libxml2 (2.9.0)	–	–	–	2,420	4,423	6,757
busybox (1.23.1)	–	–	–	3,278	5,046	7,281
bash (4.2)	–	–	–	3,659	6,577	10,262
vim (6.0)	–	–	–	3,888	8,714	16,613

presence conditions, we show the total number of conditions ($\#PC$), and the number of literals ($\#L$) and clauses ($\#C$) over all presence conditions. These numbers refer to the set of presence conditions after the preprocessing step (cf. Section 5.3.2).

Measuring Fault Detection

To answer our first research question, we reuse some artifacts from the study of Medeiros et al. [2016]. In the study the authors report known faults in multiple systems and the respective condition for which a fault is present in a variant (i.e., *fault condition*)[1]. If the fault condition of a fault is *true* under a given configuration, it means that the fault will be present in the corresponding variant. In

[1] http://www.dsc.ufcg.edu.br/~spg/sampling/

Table 5.2: Overview of fault conditions of faults used from Medeiros et al. [2016]

Degree	Count	Example
1	34	`!ENABLE_FEATURE_SYSLOG`
2	11	`ENABLE_FEATURE_EDITING && !ENABLE_HUSH_INTERACTIVE`
3	4	`ENABLE_FEATURE_GETOPT_LONG && !ENABLE_FEATURE_SEAMLESS_LZMA && !ENABLE_FEATURE_TAR_LONG_OPTIONS`
4	2	`!FEAT_GUI_W32 && !PROTO && !FEAT_GUI_MOTIF && !FEAT_GUI_GTK`
5	1	`!FEAT_GUI_W32 && !FEAT_GUI_GTK && !FEAT_GUI_MOTIF && !FEAT_GUI_ATHENA && !FEAT_GUI_MAC && FEAT_GUI`
$\sum$	52	

total, the study presents a list of 75 unique fault conditions. However, 23 of these conditions contained features that do not occur in the actual source code. This can be due to abstract features, features that are only used during the build process (e.g., in Makefiles), or due to features that have a different name in the configuration tool than in the source code. Therefore, we only used the remaining 52 fault conditions in our evaluation. Most of these fault conditions represent interaction of degree one, which means that the selection or deselection of a single feature is enough to make them active. However, the list also contains fault conditions that represent interactions of degree two, three, four, and five. We show the distribution of fault conditions in Table 5.2 together with an example for each degree of interaction. Note that, five of the 52 fault conditions also have multiple clauses in their DNF. For such a case, we consider the number of literals in the smallest clause as degree of interaction for that fault condition. This is because the smallest clause will be always be covered by any t-wise sampling with a value for t that is greater or equal to the number of its literals. For instance, the following fault condition represent an interaction of degree three, because it can be satisfied by the (de)selection of three features:

```
(SHUTDOWN_SERVER && NO_SOCKET_TO_FD && START_RSH_WITH_POPEN_RW) ||
(NO_SOCKET_TO_FD && !SHUTDOWN_SERVER && START_RSH_WITH_POPEN_RW)
```

This fault condition can be satisfied by any configuration the includes either the literals *SHUTDOWN_SERVER*, *NO_SOCKET_TO_FD*, and *START_RSH_WITH_POPEN_RW* or the literals *NO_SOCKET_TO_FD*, ¬*SHUTDOWN_SERVER*, and *START_RSH_WITH_POPEN_RW*. Thus a complete three-wise interaction coverage would guarantee to find this fault. All in all, the study includes a wide variety of interaction faults with varying degrees of complexity.

We use the list of fault conditions to check whether samples generated for theses systems do cover each fault in at least one configuration. To this end, we generate samples for each of these systems with YASA-PC (i.e., presence condition coverage) and YASA-Concrete (i.e., interaction coverage) for $t = 1$ and $t = 2$. We then count how many reported faults are covered by each sample. To determine whether a fault

is covered, we check if there exists at least one configuration in the sample that satisfies the corresponding presence condition of the fault.

Both algorithms are susceptible to the order of features and order of presence conditions that are provided as input, meaning that they will produce different results for different feature orders. Thus, we evaluate both algorithms using multiple iterations with a randomized feature order. In detail, we execute all algorithms 100 times, each time shuffling the feature order. To enable a fair comparison we use the same 100 randomized feature orders for each algorithm. A number of 100 iterations is an empirical value for our evaluation that provides a good trade-off between effort and accuracy.

Note, that we do not use any of the other algorithms in this experiment, as we do not have feature models for these systems. The lack of a feature model for a system also means that the configurations within a sample may be invalid according to the feature dependencies of the system. However, without a feature model we are not able to verify whether a configuration is invalid.

Measuring Coverage

We compute the coverage achieved by every sample with regard to two different coverage criteria, pair-wise interaction coverage (FM) and pair-wise presence condition coverage (PC). We consider a sample and, consequently, its sampling algorithm to be more effective the higher its coverage, as it potentially exposes more faults in the code.

Similar to the previous experiment, all used sampling algorithms are susceptible to the feature order in a feature model. Thus, again, we execute all algorithms 100 times, each time shuffling the feature order. In addition, we execute Random 10 times for each feature order, which results in 1,000 iterations for each system. For `Linux`, we only use 5 iterations of the experiment, as most algorithms take several hours to compute just one sample.

Measuring Sample Size

Regarding testing efficiency, we count the number of configurations in each sample computed by each algorithm. We do not consider the time required to run any actual test cases of a particular system. We do not consider the time required to run any test cases of a particular system, as this time is depended on the actual test cases for each product and the general testing approach. Nevertheless, we can assume that the testing time increases with the number of configuration in a sample, and thus, in general, a smaller sample will lead to a smaller testing time. Analogous to measuring coverage, we execute each algorithm, except Random, 100 times and randomize the feature order. Random is again run 1,000 times for each sample size.

Measuring Sampling Time

For measuring sampling efficiency, we take the time that is needed for generating a sample with each algorithm. Each experiment runs on an own JVM, in order to mitigate any side effects (e.g., just-in-time compilation). As our algorithm requires additional information from the source code (i.e., the presence conditions), we differentiate between the time needed to extract the presence conditions from the source code and the time to actually generate the sample. This is relevant, as the

Table 5.3: Faults covered across all 21 systems from Medeiros et al. [2016], including aggregated sample size and sampling time over all systems

Algorithm	t	Size			Time (s)			Faults Covered	
		∅	*Min*	*Max*	∅	*Min*	*Max*	*Yes*	*No*
YASA-PC	1	7.5	4	14	0.3	0.2	0.6	41	11
YASA-Concrete	1	2.0	2	2	0.3	0.2	0.4	36	16
YASA-PC	2	65.7	22	167	5.6	0.3	38.8	51	1
YASA-Concrete	2	16.7	12	24	0.9	0.2	3.7	47	5

extraction process only needs to be run once for each system. Though it takes some time to analyze the source code, the resulting presence conditions for each file can be saved for later reuse. For instance, if we compute samples for different values of t, we only need to run the extraction process once.

Evaluation System

We run all algorithms on the same evaluation system, with the following specifications:

- *CPU*: Intel(R) Core(TM) i5-8350U
- *Physical Memory*: 16 GB
- *JVM Max Memory*: Xmx: 14 GB
- *OS*: Manjaro (Arch Linux)
- *JVM*: OpenJDK 15.0.2

5.4.2 Results of Experiments

For brevity, we primarily present figures showing aggregated data over our measurement results. All data and a tabular overview can be found online.[2] We structure our findings according to our four research questions, that is fault detection, coverage, testing efficiency, and sampling efficiency. Afterwards, we analyze and discuss our results.

Faults Covered

We present the results of our first experiment in Table 5.3. For each algorithm and value for t, we show the number of faults that are covered or not covered by the produced samples across all systems. The number of covered faults is the *minimum* number over all 100 iterations, meaning that if none of the 100 samples for a system was able to cover a particular fault it is *not* counted as covered. Analogous, the number of not covered faults is the *maximum* number over all 100 iterations. In addition, we report the aggregated sample size and sampling time over all systems. For both values, we report its minimum, maximum and average over all 21 systems and 100 iterations.

[2] https://github.com/skrieter/evaluation-pc-sampling/tree/master/results

Table 5.4: Relative mean sample size, mean sampling time, and mean coverage aggregated over all six systems with a feature model

Algorithm	**∅Time (%)**	**∅Size (%)**	**∅Coverage (%)**	
			FM	*PC*
YASA-FM	100.0	100.0	100.0	98.6
YASA-PC	59.3	73.9	79.1	100.0
YASA-PC-1	38.8	12.6	57.1	69.7
YASA-Concrete	36.1	16.6	61.7	62.9
ICPL	319.5	132.7	100.0	97.9
Chvatal	1,046.7	131.3	100.0	98.0
IncLing	53.6	153.7	100.0	99.3
PLEDGE-Min	51.5	122.0	98.8	97.1
PLEDGE-Max	118.2	122.0	98.8	97.1

Of the 52 faults, which we investigated, we see that for both values of t YASA-PC is able to detect more faults than YASA-Concrete (i.e., 36 vs. 31 for $t = 1$ and 51 vs. 47 for $t = 2$). Only a single fault could not always be covered by YASA-PC with $t = 2$. The corresponding fault condition is the following:

```
ENABLE_HUSH_CASE && ENABLE_FEATURE_EDITING_SAVE_ON_EXIT
  && ENABLE_HUSH_INTERACTIVE && !ENABLE_FEATURE_EDITING
```

This fault condition is of degree four, and thus, when using t-wise interaction sampling, is only guaranteed to be found with $t \geq 4$.

On the other hand, we can also see that on average YASA-PC produced larger samples than YASA-Concrete (i.e., 7.5 vs. 2.0 for $t = 1$ and 65.7 vs. 16.7 for $t = 2$). Thus, the higher fault detection may also be a result of the larger sample sizes. However, as we pointed out before, we do not use a feature model for this experiment. Therefore, there are no restrictions on the configuration space, which can lead to a lower sample size. Furthermore, as we see in later experiments, a feature model, which may also include abstract features, leads to a larger sample size than considering only concrete features.

Achieved Coverage

In Table 5.4, we show a comparison of the coverage for different criteria for all algorithms. These values are aggregated over all systems and all experiments using the arithmetic mean. We show a more detailed plot of the coverage criterion *PC* over all systems and experiments in Figure 5.1 using boxplots. In addition, we performed paired t-tests to test whether the difference in achieved coverage by the different algorithms is significant. We present the results of the statistical tests in Table 5.5. In this table, we compare the coverage of all three variants of YASA with the coverage of all other algorithms for the two coverage criteria *FM* and *PC* with $t = 2$. The symbol $=$ indicates that there is no significant difference (i.e., $p > 0.05$) in the achieved coverage between both algorithms. The symbol $-$ indicates that the coverage achieved by the variant of YASA is significantly lower than the coverage of the other algorithm (i.e., $p < 0.001$). Analogous, the symbol $+$ indicates the coverage of YASA is significantly higher (i.e., $p < 0.001$).

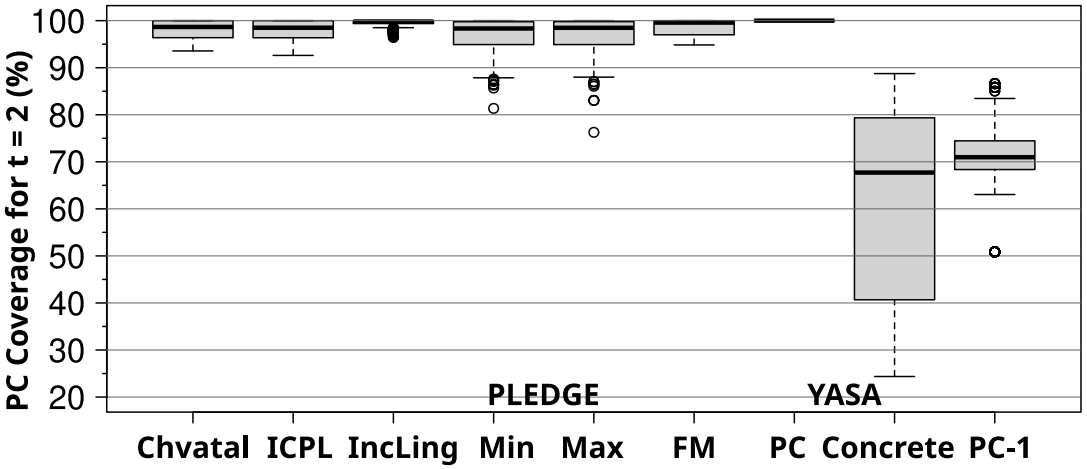

Figure 5.1: Pair-wise presence condition coverage aggregated over all systems

Table 5.5: Results of the paired t-tests for difference in *FM* and PC-coverage between YASA and other algorithms. Symbols: = indicates *no significant difference*; – indicates a *significantly lower coverage* of YASA compared to the other algorithm; + indicates a *significantly higher coverage* of YASA

Algorithm	**ICPL**	**Chvatal**	**IncLing**	**PLEDGE** *Min*	**PLEDGE** *Max*	**YASA-PC-1**
FM-Coverage						
YASA-FM	=	=	=	+	+	+
YASA-PC	–	–	–	–	–	+
YASA-Concrete	–	–	–	–	–	+
PC-Coverage						
YASA-FM	+	+	–	+	+	+
YASA-PC	+	+	+	+	+	+
YASA-Concrete	–	–	–	–	–	–

Regarding the coverage criterion *FM*, we can see that only the t-wise interaction sampling algorithms (Chvátal, ICPL, IncLing, YASA-FM) are able to achieve a 100% coverage. Both, YASA-PC and YASA-Concrete achieve a significant lower *FM* coverage. On the other hand, only YASA-PC is able to achieve a 100% *PC* coverage. All other algorithms produce samples with a significant lower *PC* coverage on average. Still, many algorithms (Chvátal, ICPL, IncLing, PLEDGE, YASA-FM) achieve a rather high average *PC* coverage of over 97%.

Sample Size

In Table 5.4, we show a comparison of the sample sizes for all algorithms. Again, the values are aggregated over all systems and all experiments using the arithmetic mean. As the actual sample size is dependent on the subject system, we normalized the sample size for every experiment using the sample size of YASA-FM as 100%. In Figure 5.2, we depict the sample size for all algorithms in more detail using boxplots. Additionally, we show the absolute values of the mean sample size per system for

Table 5.6: Results of the paired t-tests for sample size difference between YASA and other algorithms. Symbols: − indicates a *significantly lower coverage* of YASA compared to the other algorithm; + indicates a *significantly higher coverage* of YASA

Algorithm	**ICPL**	**Chvatal**	**IncLing**	**PLEDGE**		**YASA-PC-1**
				Min	*Max*	
YASA-FM	−	−	−	−	−	+
YASA-PC	−	−	−	−	−	+
YASA-Concrete	−	−	−	−	−	+

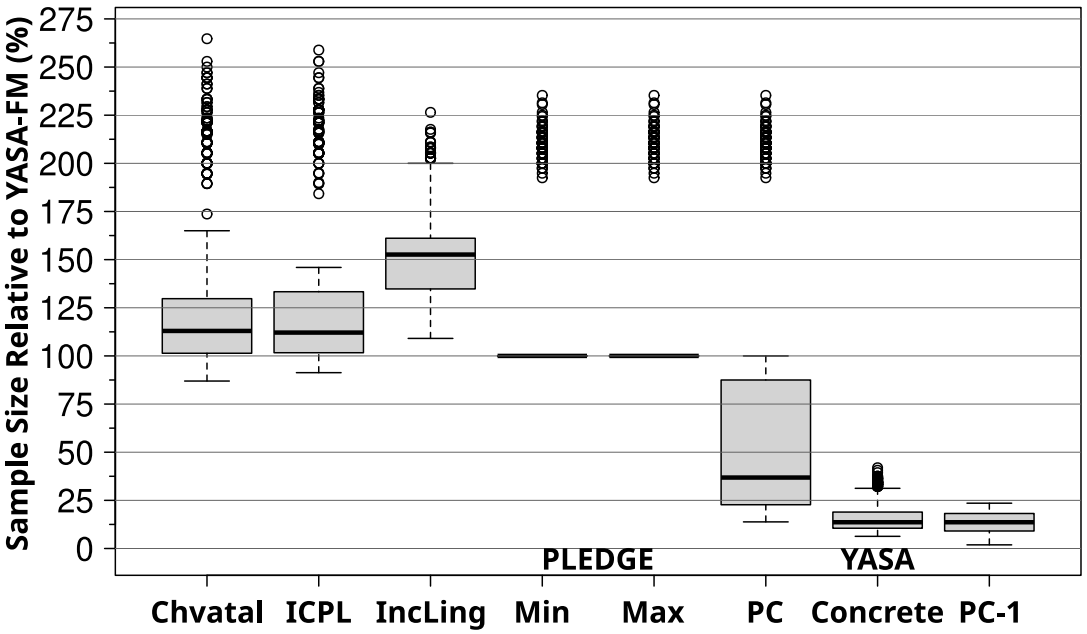

Figure 5.2: Sample size relative to YASA-FM for all six systems with a feature model

YASA-FM and YASA-PC in Table 5.7. Furthermore, we performed paired t-tests to test whether the difference in sample size computed by the different algorithms is significant and show the results in Table 5.6. In this table, we compare the sample size of all three variants of YASA with the sample size of all other algorithms. The symbol − indicates that the sample size computed by the variant of YASA is significantly smaller than the sample size of the other algorithm (i.e., $p < 0.001$). Analogous, the symbol + indicates the sample size of YASA is significantly greater (i.e., $p < 0.001$).

We can observe that on average all algorithms that consider the solution space (i.e., YASA-PC, YASA-Concrete, and YASA-PC-1) produce significantly smaller samples than algorithms that only use the feature model. An exception is the system `BusyBox`, for which YASA-PC produces samples that are about two times larger than the sample of YASA-FM.

Correlation Between Coverage and Sample Size

To illustrate the correlation between sample size and testing effectiveness, we show, in Figure 5.3, a comparison of the coverage criterion PC with $t = 2$ for all algorithms

Table 5.7: Absolute mean sample size and mean sampling time for YASA-FM and YASA-PC for all 6 systems with a feature model

System	∅Size		∅Time (s)		∅Extract (s)
	FM	PC	FM	PC	
fiasco	21.5	5.4	1.0	0.6	0.7
axtls	32.3	27.3	1.4	1.3	1.4
uclibc-ng	362.4	54.5	6.8	3.3	0.8
toybox	18.4	6.5	3.6	0.7	4.5
BusyBox	37.6	79.1	28.6	20.7	2.1
Linux	493.4	189.4	8,938.4	1,248.6	64.5

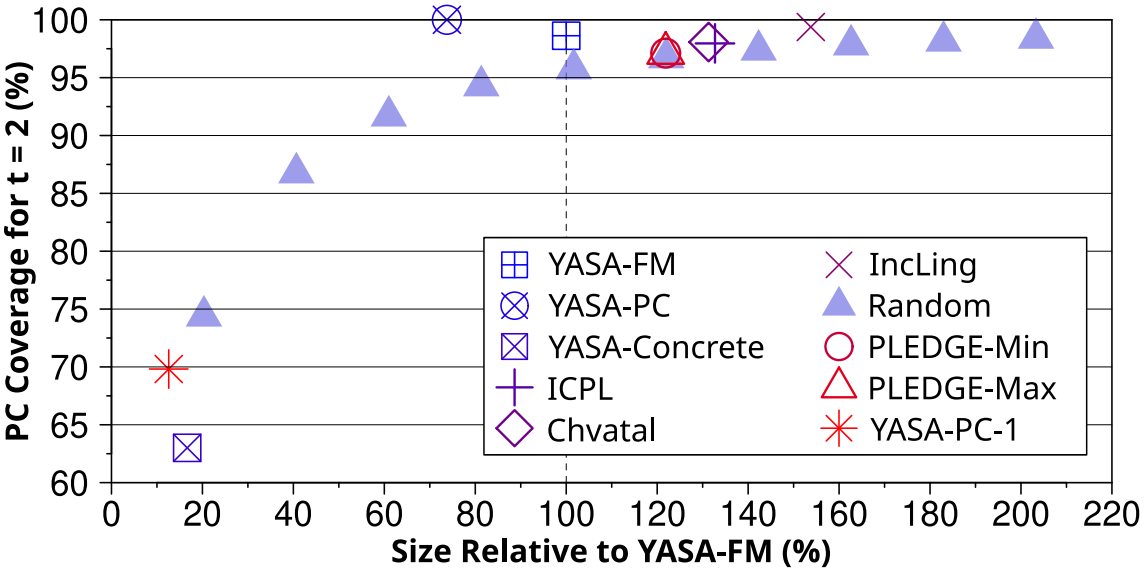

Figure 5.3: Presence condition coverage compared to sample size for all algorithms aggregated over all six systems with a feature model

for different configuration sizes. On the x-axis, we show the sample size relative to the sample size of YASA-FM (i.e., being 100%). On the y-axis, we show coverage in % for PC with $t = 2$. Each data point represents the average for all samples per algorithm and system. Random acts as a base line in this diagram, as it does not aim for a certain coverage criterion. We can see a clear correlation between sample size and the coverage criterion PC (i.e., increasing the sample size leads to higher coverage on average). Further, we can see that YASA can reach a 100% coverage with a substantially smaller sample than all other tested algorithms for most cases.

In addition, we calculate the Spearman's rank correlation coefficient between the degree of coverage and the sample size for all algorithms. For the coverage criterion PC, we get a significant positive correlation of ≈ 0.157 with a p-value of $p < 0.001$. Similarly, for the coverage criterion FM, we also get a significant positive correlation of ≈ 0.2 with a p-value of $p < 0.001$.

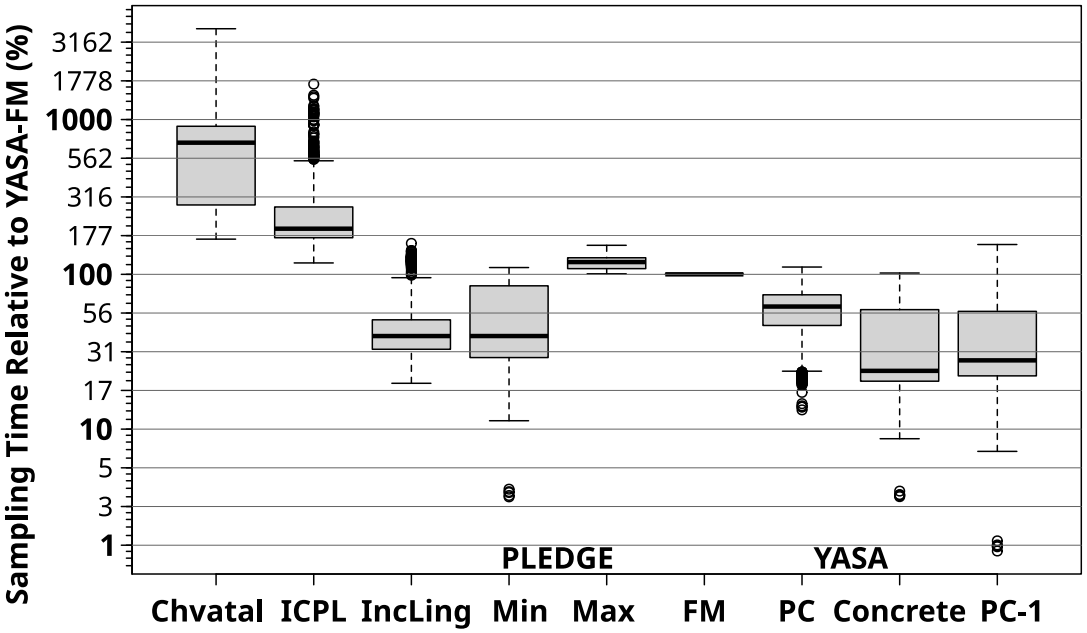

Figure 5.4: Sampling times of all algorithms relative to *YASA-FM* on the same system

Sampling Time

In Table 5.4, we show the average sampling time for all algorithms, aggregated over all systems using the arithmetic mean and relative to the sampling time of YASA-FM. In addition, we depict the sampling time for all algorithms in more detail in Figure 5.4 relative to the sampling time of YASA-FM. Here, we only show the pure sampling time, excluding the time needed for extracting presence conditions. As we described in Section 5.3.1, the extraction process is dependent on the employed variability mechanism, but only needs to be executed once, if the implementation artifacts do not change. In Table 5.7, we show for each system the absolute time in seconds required for YASA to extract the presence conditions. Additionally, in Table 5.7, we show the absolute values for the mean sampling time of YASA-FM and YASA-PC. We performed paired t-tests to test whether the difference in sampling time required by the different algorithms is significant. In Table 5.8, we show the results of these statistical tests by comparing the sampling time of all three variants of YASA with the sample size of all other algorithms. The symbol − indicates that the sampling time computed by the variant of YASA is significantly smaller than the sampling time of the other algorithm (i.e., $p < 0.001$). Analogous, the symbol + indicates the sampling time of YASA is significantly larger (i.e., $p < 0.001$).

We can see that YASA-Concrete is significantly faster than all other algorithms, except for YASA-PC-1. Notably, the sampling time of YASA-PC-1 is significantly smaller than all other three variants of YASA, which we did expect, because of its lower value of t. YASA-FM and YASA-PC are able to significantly outperform ICPL, Chvátal, and PLEDGE-Max, but not IncLing and PLEDGE-Min. Chvátal can be up to 30 times and ICPL up to 17 times slower than YASA-FM. However, YASA-PC, which uses presence conditions, is faster than YASA-FM by a factor of 0.6 on average. Regarding presence condition extraction time, we can see that in most cases it is

Table 5.8: Results of the paired t-tests for difference in sampling time between YASA and other algorithms

Algorithm	ICPL	Chvatal	IncLing	PLEDGE		YASA-PC-1
				Min	*Max*	
YASA-FM	−	−	+	+	−	+
YASA-PC	−	−	+	+	−	+
YASA-Concrete	−	−	−	−	−	+

similar to the sampling time. In case of Linux and BusyBox the extraction time is even substantially lower than their sampling time.

5.4.3 Discussion

RQ$_1$ Regarding RQ_1, we found in our experiments that samples with a 100% t-wise presence condition coverage were able to cover more faults than samples with 100% t-wise interaction coverage. These results indicate that t-wise presence condition coverage is indeed able to detect more faults than t-wise interaction coverage for the same value of t. Furthermore, from our results we can see that t-wise presence condition coverage for $t = 1$, which achieves at least the same and possible even higher PC-coverage than Tartler et al. [2014]'s approach, finds less faults than t-wise presence condition coverage for $t = 2$. This confirms our expectation that covering combinations of presence conditions is more effective than covering single presence conditions.

RQ$_2$ For RQ_2, we can see that only YASA-PC is able to guarantee a 100% t-wise presence condition coverage. Nevertheless, most of the other sampling algorithms, such as Chvátal, ICPL, IncLing, and PLEDGE achieve a high pair-wise presence conditions coverage (i.e., over 97% on average). YASA-PC-1 achieves the lowest coverage, which is expected, as it aims for a presence condition coverage for $t = 1$, which is not concerned with presence condition interactions of higher degrees. While these results indicate an already good t-wise presence condition coverage with traditional sampling algorithms, it also shows that samples from these algorithms most likely miss some interactions between code blocks, which then cannot be tested. Furthermore, the high coverage of these algorithms might be due to the relatively large sample size compared to YASA-PC. Notably, random sampling with similar sample sizes also yields a similar coverage as the traditional sampling algorithms. In summary, it is possible for other algorithms to achieve a high t-wise presence condition coverage, though it cannot be guaranteed. However, the high t-wise presence condition coverage of the traditional sampling algorithms seems to be caused by their larger sample sizes.

RQ$_3$ Concerning RQ_3, we observe that for all systems, except `BusyBox`, YASA-PC generates smaller samples, than Chvátal, ICPL, IncLing, and YASA-FM. This may be caused by the relatively low number of concrete features for some systems, which facilitates the coverage of presence conditions within a configuration. The larger

sample size of `BusyBox` may be caused due to a high number of mutually exclusive presence conditions. When considering presence conditions, we see that even for systems with more presence conditions than features (e.g., `axtls`, `uclibc-ng`, and `Linux`) YASA-PC is able to generate smaller samples. Moreover, the results of YASA-Concrete show that it is crucial to not just consider concrete features, which yields smaller samples using t-wise interaction algorithms, but only reaches a PC coverage of about 67% on average. In summary, for most cases t-wise presence condition sampling produce relatively small samples, which may increase its testing efficiency.

RQ$_4$ Regarding RQ_4, we see that the sampling time of YASA-PC is relatively small and even outperforms some t-wise interaction sampling algorithms. The additional time for extracting presence conditions is similar to the sampling time for the smaller systems and even neglectable for larger systems such as `Linux`. In summary, the initial generation of samples with YASA-PC is only slightly less efficient than with traditional sampling algorithms due to the necessary extraction of presence conditions.

To conclude, with YASA we are able to efficiently generate relatively small samples for t-wise presence condition coverage. In addition, our results indicate that t-wise presence condition coverage is able to increase the fault-detection rate for product-based testing. Thus, we may be able to increase the testing efficiency and testing effectiveness by using t-wise presence condition sampling.

5.4.4 Threats to Validity

We are aware that our evaluation might suffer from some biases that may threaten its validity. In the following, we look into potential internal and external threats to validity regarding our evaluation.

Internal

We aim to evaluate the concept of t-wise presence condition sampling. However, we only use one particular sampling algorithm to cover interactions between presence conditions. It may be the case that the concept performs better or worse when employing other sampling algorithms or different heuristics. Still, we evaluated our employed sampling algorithm YASA separately in Section 4.5 and found that it performs on par or even better than other state-of-the-art algorithms for t-wise interaction sampling.

For extraction presence conditions for configurable systems, we use an extraction algorithm, based on the tool *PCLocator*. Due to the high expressiveness of the C preprocessor, the extraction algorithm has some limitations, which can lead to incorrect results in some cases. Most notably, the algorithm does not expand any preprocessor macro statements, which can lead parsing problems or missed annotations. In case of a parsing problem for a line, the resulting presence conditions for this line is assumed to be *true* (i.e., the line is always present). Further, the algorithm does only consider Boolean features. All features with numerical or string values are ignored. Due to these limitations, it is possible that some returned presence

conditions might be incorrect or incomplete. There is unfortunately nothing we can currently do to mitigate this bias. Nevertheless, the tool on which our algorithm is based employs three established tools for analyzing C preprocessor annotations in order to achieve more reliable results.

Another potential threat is the chance of faults in our own implementation that may affect the results. To this end, we use automated tests to ensure that all samples we compute with YASA and any other algorithm are valid and achieve the appropriate degree of interaction or presence condition coverage.

External

We are using a rather small set of configurable systems with feature models in our evaluation. Thus, the results might not scale to other systems with more complex dependencies and presence conditions. To mitigate this potential bias, we used configurable systems from different domains and with varying numbers of features, constraints, and presence conditions.

We do not evaluate the actual testing effectiveness of our approach, but only compare it to other algorithms with respect to our own coverage criterion. As we proposed the coverage criterion of t-wise presence condition coverage, this may entail an unfair comparison with other algorithms that aim for different coverage criteria. We do evaluate whether samples with 100% t-wise presence condition coverage are able to detect some known faults in several systems. However, as this is only a small set of faults, it may hamper the generalizability of our results. Nevertheless, we tried to include faults with varying degrees of interaction and complex presence conditions to mitigate this bias.

Finally, as we are using randomized features orders and randomness within our sampling algorithms, our results may be affected by a random bias. We tried to mitigate this bias by performing multiple repetitions for all experiments.

5.5 Summary

In this chapter, we presented our new coverage criterion t-wise presence condition coverage, which we can use to evaluate a sample from any sampling algorithm. Further, we introduced an extension to our sampling algorithm YASA to enable the generation of samples with a t-wise presence condition coverage of 100% for a given value of t. Our evaluation confirms that this new coverage criterion is more suitable to indicate the effectiveness of a sample for product-based testing than regular interaction coverage, as it takes the solution space into account. Furthermore, samples with t-wise presence condition sampling also tend to be smaller than samples from t-wise interaction coverage with the same value of t. To develop our algorithm for t-wise presence condition sampling, we use our contributions from Chapter 3 and Chapter 4. Due to this combination, we developed a sampling algorithm that uses a our new coverage criterion and has the efficiency and adaptability of YASA. In conclusion, we find that with t-wise presence condition sampling, we are able to achieve better or similar results for product-based testing than existing sampling algorithms, while maintaining a smaller sample size.

6. Related Work

In this chapter, we present other work that is related to our contributions in this thesis. We compare our contributions to others by pointing out commonalities and differences and identify whether existing concepts can be treated as an alternative or as a complement to our contributions.

We structure this chapter as follows: First, we present related work on the *fully-automated configuration process* and especially sampling (see Section 6.1). Second, we look into work related to the *semi-automated configuration process* (see Section 6.2). Finally, we discuss other *techniques for variability analysis* similar to MIGs (see Section 6.3).

6.1 Fully-Automated Configuration Process

Besides manual configuration by a user, configurations can also be created automatically by certain algorithms. An application scenario for an automated configuration process is sampling, which requires the generation of a representative set of configurations [Carmo Machado et al., 2014; Lopez-Herrejon et al., 2014; Perrouin et al., 2010]. There exist many strategies for sampling using different technical approaches and having different purposes in mind [Carmo Machado et al., 2014; Lopez-Herrejon et al., 2014; Perrouin et al., 2010]. Most sampling approaches are designed for testing configurable systems [Al-Hajjaji et al., 2016a; Cohen et al., 2006, 2008; Fischer et al., 2016; Henard et al., 2013, 2014a; Lochau et al., 2012; Oster et al., 2010, 2011; Perrouin et al., 2012; Tartler et al., 2014]. However, there are also other application scenarios, such as the optimization of non-functional properties [Benavides et al., 2005; Ochoa et al., 2015; Siegmund et al., 2012; White et al., 2007, 2009]. While we also develop YASA and especially t-wise presence condition sampling for the purpose of testing configurable systems, like many other approaches they are not limited to this use case, but can be utilized for other purposes as well.

Some sampling approaches are focused on prioritizing the configurations within a sample, for instance for increasing testing efficiency [Al-Hajjaji et al., 2017, 2019;

Ensan et al., 2011; Henard et al., 2014b]. In our thesis, we did not focus on any prioritization for the configurations in sample generated by YASA. However, we discussed the possibility of using a different covering strategy that can affect the order of configurations in the final sample.

YASA uses a greedy strategy to attempt to compute minimal samples, which is deterministic and guarantees a certain degree of coverage. Greedy strategies are applied by many other sampling algorithms as well [Abal et al., 2018; Al-Hajjaji et al., 2016a; Ensan et al., 2011; Haslinger et al., 2013; Johansen et al., 2011, 2012a,b; Kim et al., 2010, 2011; Kowal et al., 2013; Liebig et al., 2013; Oster et al., 2010; Reuling et al., 2015; Shi et al., 2012; Tartler et al., 2014]. As the underlying problem of finding an optimal sample is NP-hard, a greedy strategy often provides a good trade-off between sampling time and sample effectiveness. Thus, all of these algorithms, including YASA, have in common that they are not guaranteed to find an optimal sample. In fact, YASA is similar to the algorithm IPOG [Lei et al., 2007], which also starts with an empty sample and iteratively adds and refines partial configurations. However, YASA is more flexible due to its configurability and integrates many concepts to increase its efficiency. Besides greedy sampling algorithms, there also exist many sampling algorithms that employ meta-heuristic strategies [Cmyrev and Reissing, 2014; Devroey et al., 2015; Ensan et al., 2012; Ferreira et al., 2016, 2017; Filho et al., 2017; Henard et al., 2013, 2014a; Lopez-Herrejon et al., 2014; Marijan et al., 2013; Matnei Filho and Vergilio, 2016]. While these algorithms also cannot guaranteed to find an optimal sample, they are often faster than greedy strategies. These approaches can be seen as alternative to our concept. Moreover, it would be possible to use some of these algorithm in combination with YASA. YASA is able to use a sample produced by another sampling algorithm as input (cf. Section 4.2.3), which could speed up the sampling process or even increase the sampling effectiveness. A sample created by YASA might also serve as input for a meta-heuristic algorithm to further refine the sample.

YASA uses a deterministic approach for t-wise interaction sampling. Another popular sampling strategy is random sampling [Varshosaz et al., 2018]. The difficulty for random sampling of feature model lies in the feature dependencies, which make it hard generate configurations that are uniformly distributed over the valid problem space [Plazar et al., 2019]. Nevertheless, there are approaches that implement uniform or approximate uniform random sampling [Heradio et al., 2020; Munoz et al., 2019; Oh et al., 2017, 2019]. Compared to YASA and other deterministic t-wise sampling algorithms, random sampling has some advantages. It allows users to control the number of configurations in a sample, which is useful, for instance, if only a certain number of configurations can be tested in a given time. Furthermore, in general, random sampling uses less memory and can be substantially faster than deterministic t-wise sampling algorithms. Another disadvantage of random sampling is that it cannot guarantee a 100% t-wise coverage. Users are able to cover a certain percentage of interactions with only a few configurations, however to cover all interactions often requires more and also a specific combination of configurations.

Most sampling algorithms are based on only using the feature model as input, and thus solely consider the problem space of a configurable system [Varshosaz et al., 2018]. With t-wise presence condition sampling, we take variability from the problem

and solution space into account for creating samples. However, there already exist sampling strategies that consider other inputs in addition to a feature model [Liebig et al., 2013; Varshosaz et al., 2018]. Similar to t-wise presence condition sampling, there are sampling algorithms that consider implementation artifacts [Liebig et al., 2013; Shi et al., 2012; Tartler et al., 2012, 2014] and also testing artifacts [Kim et al., 2010, 2011] to compute a sample. While these algorithms also consider the solution space, we are the first to combine presence conditions for implementation artifacts with regular t-wise interaction sampling to achieve higher effectiveness.

6.2 Semi-Automated Configuration Process

One of the most important tasks in the semi-automated configuration process is to ensure the validity of the resulting configuration. For this, many tools already provide decision propagation as part of an interactive configuration process. Among those tools are FeatureIDE [Pereira et al., 2016a; Thüm et al., 2014b], GEARS [Krueger, 2007; Krueger and Clements, 2013], GUIDSL [Batory, 2005], S2T2 Configurator [Botterweck et al., 2009], S.P.L.O.T. [Mendonça et al., 2009a], and VariaMos [Mazo et al., 2012]. Our basic algorithm for decision propagation is based on the SAT-based decision propagation as proposed by Janota [Janota, 2008]. Others proposed to propagate decisions using BDDs [Hadzic et al., 2004; Mendonça et al., 2008] and algorithms based on solvers of the constraint satisfaction problem (CSP) [Amilhastre et al., 2002; Benavides et al., 2005]. As we mentioned earlier, BDDs are similar to MIGs, as both are a type of knowledge compilation technique that avoid some effort during decision propagation (i.e., online phase) by demanding more effort for their creation (i.e., offline phase). A problem with BDDs is that they typically do not scale for feature models larger than 1,000 features. For instance, there is no BDD for Linux so far [Thüm, 2020]. While CSP solvers can handle constraints beyond boolean formulas as needed for extended feature models [Benavides et al., 2005], they often reduce inputs to satisfiability problems internally. Hence, we do not expect that they could improve performance compared to our MIG-assisted algorithms. Similar to CSP, satisfiability modulo theories (SMT) extend the boolean SAT problem to first-order logic [De Moura and Bjørner, 2011; Sprey et al., 2020]. As far as we know, SMT solvers, such as Z3 [De Moura and Bjørner, 2008], have not been applied to decision propagation yet. However, as our approach is independent from the actual solver, but instead tries to reduce the number of required SAT instances, we assume that approaches that employ SMT solvers can also benefit from MIGs.

Another technique to avoid contradictions within a configuration is *error resolution*, which automatically detects and tries to resolve conflicts [Benavides et al., 2007; Ochoa et al., 2015; White et al., 2008]. Configuration tools that support this kind of technique are, for example, pure::variants [Beuche, 2012; pure::systems, 2017] and FaMa [Benavides et al., 2007]. Contrary to decision propagation, error resolution can be applied at any point during or after the configuration process. To support error resolution, MIGs could be traversed to determine whether an updated graph for a given configuration contains a contradiction, which we discussed in Section 3.3.2.

Besides ensuring the creation of valid configurations, the semi-automated configuration process comprises a multitude of different approaches for supporting a user in

creating desired configurations. For instance, users can be supported by recommender systems [Pereira et al., 2016b] or visual feedback [Martinez et al., 2014; Nestor et al., 2008]. All techniques that consider feature dependencies can potentially benefit from MIGs, as they provide a fast and complete access to binary feature relations.

6.3 Variability Analysis Techniques

MIGs are a type of knowledge compilation [Cadoli and Donini, 1997] that represent configuration knowledge. There are other data structures and methods with a similar purpose, such as *Binary Decision Diagrams (BDDs)* [Bryant, 1986; Darwiche and Marquis, 2002] and *enumeration of all configurations* [Galindo et al., 2016; Halin et al., 2019]. These and similar approaches for knowledge compilation could be used instead of MIGs in the task of decision propagation and also other analyses [Jensen, 2004; Perez-Morago et al., 2015]. While these approaches may be more effective in facilitating certain analysis tasks, they are mostly harder to build for large and complex feature models [Thüm, 2020]. Therefore, MIGs can be seen as an alternative to these approaches that are less effective but faster to create, and thus might be more suitable in certain situations. For instance, in application scenarios where the feature model of a configurable system evolves frequently, an incrementally built MIG might outperform other knowledge compilation techniques.

Besides the feature model analyses, we mention in Chapter 2 and Chapter 3, there exist many other analyses for configurable systems that reason about feature model dependencies [Benavides et al., 2010; Liebig et al., 2013; Thüm et al., 2014a]. In our thesis, we are mainly focused on feature model analyses that consider only the problem space, such as decision propagation [Batory, 2005], finding core and dead features [Benavides et al., 2010; Trinidad and Ruiz-Cortés, 2009], and detecting atomic sets [Segura, 2008; Zhang et al., 2004]. In contrast, to previous contributions, we are the first to employ MIGs within these analyses. Similar to decision propagation, most variability analyses can be implemented with a SAT-based algorithm [Benavides et al., 2010; Thüm et al., 2014a]. In Section 3.3.2, we reasoned that many other SAT-based analyses can benefit from employing a MIG, because it can decrease the amount of SAT instances to solve.

With t-wise presence condition coverage, we use artifacts from the solution space to create better sample for testing. There exist approaches that further analyzing the variability of the solution space more directly [Liebig et al., 2013; Meinicke et al., 2014]. For instance, with type checking, a system can be analyzed for syntactical soundness [Aversano et al., 2002; Czarnecki and Pietroszek, 2006; Kenner et al., 2010; Metzger et al., 2007; Thaker et al., 2007]. Static analyses of implementation artifacts, such as control- and data-flow analysis, can provide further insights to the semantic behavior of a system and its dependencies [Midtgaard et al., 2015; Nadi et al., 2015]. Model checking allows to analyze the correctness of a system by verifying it specified behavior [Ben-David et al., 2015; Bessling and Huhn, 2014; Classen et al., 2014; Millo et al., 2013; ter Beek et al., 2019; Thüm et al., 2014].

With the incremental build process for MIGs, we attempt to reuse previous analysis results to speed up the construction of a MIG (cf. Section 3.5). A related concept is incremental SAT solving [Mouhoub and Sadaoui, 2007]. Incremental SAT solving

improves the performance for solving consecutive SAT queries, in which only small parts of each query are changed by reusing information from previous solutions [Eén and Biere, 2005; Eén and Sörensson, 2003; Fazekas et al., 2019; Nadel et al., 2014]. This could be used to complement our approach in two ways. First, as we are solving multiple similar SAT queries within the MIG build process, an incremental SAT solver may be able to speed up some operations within the original and incremental build process. Second, incremental SAT solvers could be used in conjunction with the analysis result available from previous MIGs to facilitate the build process even more. If the feature model change is small all SAT analyses within the build process could be sped up by providing them with SAT solutions from a previous feature model version.

In our evaluation of MIGs, we use configurable systems such as Linux and eCos that provide their own feature modeling language and corresponding configuration tools (i.e., KConfig [Community, 2018] and CDL [Veer and Dallaway, 2017]). Although, these languages allow for multi-valued logic, they can be translated into Boolean feature models [Berger et al., 2010; Knüppel et al., 2017; Sincero et al., 2007]. The KConfig language differentiates between *select* and *depends* constraints. In terms of modal implication graphs, select can be considered a strong edge, as it directly implies other features, while depends can be seen as a set of weak edges. In contrast, MIGs do not rely on manual specification of select and depends constraints, but can compute the respective relationships, which avoids mistakes by users and represents feature dependencies more efficiently.

7. Conclusion and Future Work

In the following, we summarize the contributions and insights within our thesis and draw a conclusion on our main findings by answering the five research questions we state at the beginning. Further, we present potential future research, which we think is of relevance and should be investigated in more detail.

7.1 Summary

In this thesis, we made three main contributions for facilitating the configuration process and testing of configurable systems, the introduction of *modal implication graphs (MIGs)*, the development of the *sampling algorithm YASA*, and the introduction of the coverage criterion *t-wise presence condition coverage*. We evaluated our contributions in consideration of our main research questions. In the following, we summarize our main contributions and key findings.

We introduced MIGs to support the semi- and fully-automated configuration process. MIGs are an extension of regular implication graphs and let us easily identify pairwise relationships between features, which decreases the computational effort for reasoning about the validity of a configuration (cf. Section 3.3). We presented an algorithm for decision propagation that employs MIGs and increases performance compared to state-of-the-art algorithms. In addition, we employ MIGs in our sampling algorithm YASA. We developed a basic and advanced build process to construct a MIG for any non-void feature model and an incremental build process to update an existing MIG after a feature model evolution. As a result of our evaluation of MIGs and their build process, we gathered some key insights in this concept. First, using a MIG for decision propagation can significantly lower its execution time. Compared to regular SAT-based decision propagation, we can see a high reduction when employing a MIG. This is even true for incomplete and non-minimal MIGs. Second, constructing a MIG requires some additional time that depends on the particular build process. Using our basic build process, an incomplete and non-minimal MIG can be constructed fast. Even for a single complete configuration process using decision propagation the time required to build an incomplete and non-minimal MIG can be amortized.

The advanced build process to construct a complete and minimal MIG requires significantly more time. The incremental build process is substantially faster than the advanced build process and on par with the basic build process. However, the degree of minimality and completeness of an incrementally built MIG can be higher than from a MIG constructed by the basic build process. Third, the difference in effectiveness regarding the execution time of decision propagation of a minimal and complete MIG compared to a non-minimal and incomplete MIG is relatively small. Completeness of a MIG can increase the performance of some analysis substantially, such as finding atomic sets, however for decision propagation even an incomplete MIG is sufficient. Fourth, we conclude that for most scenarios, using an incomplete MIG resulting from our basic or incremental build process is most suitable. Using a pure SAT-based approach without a MIG is only suitable if decision propagation is performed only a few times for a feature model. In contrast, using a complete and minimal MIG is only suitable if decision propagation is used many times or if the initial build time is irrelevant. Thus, for most scenarios, an incomplete MIG seems to provide the best trade-off between the time required for its construction and the time saved during analysis.

We introduced YASA, a new sampling algorithm for t-wise interaction coverage. We designed YASA to efficiently produce small samples and to be adaptable to particular user requirements. To this end, YASA provides a number of parameters, such as feature subsets, resampling limit, and initial sample, that give users more control over sampling time, sample size, and other properties of the sample. Furthermore, YASA consists of a modular architecture that allows to replace certain strategies within the algorithm in order to further affect the properties of the computed samples. From our evaluation of YASA, we collected some key insights. First, YASA is more or at least as efficient as other state-of-the-art algorithms. The extend of the decrease in sampling time compared to another algorithm is dependent on the parameter settings of YASA. The parameter for setting a resampling limit has an inverse linear correlation with the sampling time. Setting a low value speeds up the creation of a sample, but may also increase the resulting sample size. Furthermore, providing a small feature subset to YASA substantially lowers the amount of interactions considered within a sample, and thus also decreases sampling time. However, this also decreases the interaction coverage degree, which may only be acceptable in some use cases. Second, the samples generated by YASA are smaller compared to other state-of-the-art algorithms for t-wise interaction coverage. The sample size is also dependent on the parameter settings of YASA. Choosing a high value for the resampling limit can decrease the sample size, while simultaneously increasing the sampling time. Third, summarizing our first two key findings, we conclude that users of YASA have more control over sampling time, sample size, and coverage degree than with other algorithms. By choosing specific parameter settings, user can adapt the behavior of YASA to their concrete use case scenario. Fourth, currently, there is a limit for the scalability of YASA regarding its sampling time. For feature models with more than 10,000 features and a t value of ≥ 2 the sampling time of YASA can be infeasible for many applications. However, this can be mitigated by using YASA parameter to set a feature subset and allowing for samples with a lower interaction coverage degree. If choosing feature subsets based on domain knowledge

of a configurable system, this could be a viable solution to create samples even for large-scale feature models.

We introduced the coverage criterion t-wise presence condition coverage to improve the testing effectiveness and efficiency for configurable systems. In contrast to interaction coverage, presence conditions coverage considers the solution space of a configurable system by taking presence conditions of implementation artifacts into account. Our coverage criterion can be utilized to determine to which degree a given sample covers the combinations of actual implementation artifacts and by this to indicate its fault-detection potential. In addition, we extended our algorithm YASA to produce samples with a 100% degree of t-wise presence condition coverage. After evaluating t-wise presence condition coverage and our extension to YASA for t-wise presence condition sampling, we can report some key insights. First, samples with a 100% t-wise presence condition coverage are able to detect more faults than samples with a 100% t-wise interaction coverage for the same value of t. Thus, t-wise presence condition coverage seems to be a better indicator for the effectiveness of a sample in detecting faults during testing. Second, YASA is able to generate a sample with a 100% t-wise presence condition coverage compared to other algorithms that only reach 97% on average. Most notably, random sampling yields the same degree of t-wise presence condition coverage than the traditional sampling algorithms with a similar sample size. Thus, samples from these algorithms do not cover some interactions between implementation artifacts, which then cannot be tested. Third, the samples generated by YASA for t-wise presence condition sampling are on average smaller, than the samples from t-wise interaction coverage algorithms. Thus, samples generated by t-wise presence condition sampling lead to a increased testing efficiency. Fourth, the sampling time of YASA for t-wise presence condition sampling is similar or faster than t-wise interaction sampling algorithms. Thus, the sampling efficiency for t-wise presence condition sampling is about equal to regular t-wise interaction sampling.

7.2 Conclusion

With the key insights described above, we are able to answer our main research questions (cf. Section 1.1). *Can we increase the efficiency of configuration generation in an interactive and automated configuration process using MIGs?* MIGs are well suited to speed up decision propagation in the semi-automated configuration process. By increasing the performance of propagating each decision, they facilitate the entire interactive configuration process of a user. Furthermore, MIGs are a valuable part of our sampling algorithms, which perform better than current state-of-the-art algorithms. Thus, we are able to utilize MIGs to increase the efficiency of the fully-automated configuration process, as well.

Can we control the initial build costs and effectiveness of a MIG by modifying its build process? We presented different variants of a constructing a MIG with varying degrees of completeness and minimality. While creating a working, but incomplete and non-minimal MIG requires only neglectable computational effort, creating a complete and minimal MIG is much more computationally expensive. On the other hand, using a complete MIG improves its effectiveness compared to an incomplete

version of it only by a small degree. Thus, we can control the effectiveness and build time of a MIG by choosing an appropriate build process. Nevertheless, the trade-off between additional build time and effectiveness is biased heavily towards creating incomplete MIGs, which are slightly less effective. For most scenarios a MIG can benefit the configuration process, while also being built and updated quickly. Only in certain scenarios constructing a complete MIG is worth the required computational effort.

Can we increase the efficiency of configuration generation in an automated configuration process using YASA? With YASA, we can produce configurations more efficiently compared to other-state-of the art sampling algorithms. This is true for computing samples with an equal degree of coverage as other sampling algorithms. However, by lowering the coverage degree using appropriate parameter settings, we are able to compute samples even faster. Thus, we are able to successfully facilitate the fully-automated configuration process using YASA.

Can we control the sample size and sampling time of YASA with its parameter settings? YASA is capable of producing smaller samples and also being faster than other state-of-the-art sampling algorithms, however not always simultaneously. The behavior of YASA can be controlled by its parameter settings, which provide a trade-off between the sample size and sampling time. Especially the parameter for setting a resampling limit has a high impact on both properties. A high value for this parameter creates smaller samples, but also increases the sampling time, and vice-versa. Thus, we enable users to control the sampling time, sample size, and also coverage degree of a sample by modifying the parameter settings of YASA.

Can we increase the testing efficiency or testing effectiveness by employing t-wise presence condition coverage? With t-wise presence condition sampling, we are able to create small samples that are more effective regarding their fault detection potential than with state-of-the-art algorithms for t-wise interaction coverage. The sampling time of t-wise presence condition sampling is on par or even faster than current state-of-the-art sampling algorithms. Thus, we are able to increase both, the testing efficiency and testing effectiveness by using t-wise presence condition sampling, which can facilitate the testing effectiveness and testing efficiency for configurable systems.

7.3 Future Work

In the following, we present some potential future work that could lead to interesting results, in our opinion.

We use MIGs throughout our thesis for facilitating decision propagation and increasing the performance of our sampling algorithms. In Section 3.3.2, we also mentioned the application of MIGs for other feature model analyses. Another concept, we did not discuss is feature model transformation, such as the decomposition of feature models into smaller sub feature models [Schroeter et al., 2012; Schröter et al., 2016] by means of feature model slicing [Acher et al., 2011; Krieter et al., 2016]. Feature model decomposition can be useful for many application scenarios, especially sampling and the analysis of feature models. Decreasing the number of features in a feature model is a straight-forward way to decrease the complexity of the variability and by

this the computational effort of most analyses, including sampling and interactive configuration. We expect that MIGs can help with feature model decomposition by identifying suitable sub models of a feature model. If there exists two sets of vertices where there is no path between any vertex from one set to another one, these two sets form disconnected sub graphs, which means that their corresponding feature variables do not affect each other. By partitioning a MIG into disconnected or less connected sub graphs, we can find sets of features that are mostly independent from each other, which makes the process of feature model slicing more efficient. Thus, we want to further research the decomposing of feature models using MIGs.

Our sampling algorithm YASA can be customized via three parameters, in addition to the feature model and a value for t. We already showed how these parameters affect the sampling time, sample size, and other properties of a sample. However, in this thesis we did not investigate how we can optimize these parameter given a particular configurable system. Regarding, the parameter feature subsets, we assume that MIGs can be helpful to determine a suitable partition of the entire feature set. Alternatively, domain knowledge about the system can be used to reason about useful feature subset. Therefore, we plan more research on the investigation of suitable feature subsets to increase the performance of YASA either further. To this end, we aim to enable the identification of suitable subsets using MIGs. In addition, we want to investigate whether there is other domain knowledge that can be used to determine useful feature subsets.

Another parameter of YASA is the specification of an initial sample. We already discussed that an initial sample could be provided by a different sampling algorithm. By this YASA may be easily combined with another sampling algorithm, for instance with an evolutionary algorithm, to enhance the quality of a sample. Thus, we plan to use this mechanism to combine YASA with other sampling algorithms that provide an initial (partial) sample. To this end, we aim to evaluate the impact of such a combination on sampling time, sample size, and effectiveness of a sample

With the parameter resampling limit we can affect sampling time and sample size, substantially. In our evolution of YASA, we experimented with different setting for the resampling limit. However, we do not determine an optimal value for a given feature model. Thus, we plan to develop an approach to determine an optimal value for the resampling limit given a feature model. Alternatively, we want to look into a dynamic termination criterion, instead of a fixed value for the resampling limit.

Another property of YASA is that we can easily replace the used covering and auto completion strategy in order to impact the properties of a sample. In the thesis, we focus on one particular covering strategy designed for reducing the sampling time. However, we are interested in researching other useful covering strategies that may optimize different aspects of the sampling process. The same is true for different auto completion strategies. Although, we presented several alternative completion strategies in this thesis, we did not compare them directly in our experiments. Thus, we plan to develop additional covering strategies specific to other use case scenarios, such as regression testing, and evaluated them in terms of sampling time, sample size, and other properties of a sample. Moreover, we plan to look into other auto completion strategies and their effect on the properties of a sample.

For t-wise presence condition sampling, we rely on the extraction of presence conditions from the source code of a configurable system. For the prototype we used in our evaluation, we implement an extraction process for presence conditions of systems that use kconfig and the C preprocessor as variability mechanism. However, this extraction process cannot be used for any other system that implements its variability differently. Thus, we want to look into the feasibility of tool support to partly automate the extraction of presence conditions on arbitrary configurable systems.

Furthermore, similar to regular t-wise interaction sampling, not all implementation artifacts actually interact with each other. Including unnecessary combination of presence conditions in sample can increase the sample size and by this decrease testing efficiency. As we use YASA, we are able to provide a subset of presence conditions considered for coverage, which can mitigate this issues. Thus, we aim to develop static analysis that can be executed before starting a sampling process and identifies groups of presence conditions that probably interact.

We aim to perform the proposed future research in order to further improve the configuration process and testing of configurable systems.

List of Figures

List of Tables

List of Algorithms

List of Code Listings

List of Symbols

Name	Symbol	Domain
Universes		
Features	$\mathbb{F}$	–
Literals	$\mathbb{L}$	$= \{\phi^+(f) \mid f \in \mathbb{F}\} \cup \{\phi^-(f) \mid f \in \mathbb{F}\}$
Propositional Formulas	$\mathbb{P}$	$= \{\phi(\mathcal{M}) \mid \mathcal{M} \in \mathbb{M}\}$
Modal Implication Graphs	$\mathbb{G}$	$= 2^{\mathbb{L}} \times 2^{\mathbb{L}} \times 2^{(\mathbb{L}^2)} \times 2^{2^{\mathbb{L}}}$
Variables		
Feature Model	$\mathcal{M}$	$\in \mathbb{M}$
Feature	f	$\in \mathbb{F}$
Feature Set	$\mathcal{F}$	$\subseteq \mathbb{F}$
Literal	l	$\in \mathbb{L}$
Constraint	d	$\subseteq \mathbb{L}$
Constraint Set	$\mathcal{D}$	$\subseteq 2^{\mathbb{L}}$
Configuration	c	$\subseteq \mathbb{L}$
Configuration Set	$\mathcal{C}$	$\subseteq 2^{\mathbb{L}}$
Configuration List	$\vec{\mathcal{C}}$	$\in 2^{(\mathbb{L}^{<\mathbb{N}})}$
Modal Implication Graph (MIG)	$\mathcal{G}$	$\in \mathbb{G}$
MIG Vertex	v	$\in \mathbb{L}$
MIG Vertices	$\mathcal{V}$	$\subseteq \mathbb{L}$
MIG Vertex Sequence	$\vec{\mathcal{V}}$	$\in \mathbb{L}^{<\mathbb{N}}$

MIG Edge	e	$\in \mathbb{L}^2$
MIG Path	p	$\in (\mathbb{L}^2)^{<\mathbb{N}}$
Interaction	i	$\subseteq \mathbb{L}$
Interaction List	$\vec{\mathcal{I}}$	$\in 2^{(\mathbb{L}^{<\mathbb{N}})}$
Presence Condition	$\mathcal{P}$	$\subseteq 2^{\mathbb{L}}$
Presence Condition Clause	d	$\subseteq \mathbb{L}$

Functions

Positive Literal	$pos(f)$	$\mathbb{F} \to \mathbb{L}$
Negative Literal	$neg(f)$	$\mathbb{F} \to \mathbb{L}$
Feature Model Features	$\mathcal{F}(\mathcal{M})$	$\mathbb{M} \to 2^{\mathbb{F}}$
Feature Model Dependencies	$\mathcal{D}(\mathcal{M})$	$\mathbb{M} \to 2^{\mathbb{L}}$
Literals of a Feature Set	$\mathcal{L}(\mathcal{F})$	$2^{\mathbb{F}} \to 2^{\mathbb{L}}$
Literals of a Feature Model	$\mathcal{L}(\mathcal{M})$	$\mathbb{M} \to 2^{\mathbb{L}}$
Valid Configurations	$\mathcal{C}(\mathcal{M})$	$\mathbb{M} \to 2^{\mathbb{L}}$
Feature Model Formula	$\phi(\mathcal{M})$	$\mathbb{M} \to \mathbb{P}$
Feature Model Change	$\Delta(\mathcal{M}, \mathcal{M}')$	$\mathbb{M}^2 \to 2^{\mathbb{L}} \times 2^{\mathbb{L}} \times 2^{2^{\mathbb{L}}} \times 2^{2^{\mathbb{L}}}$
CNF Clause	$\bigvee(d)$	$2^{\mathbb{L}} \to \mathbb{P}$
DNF Clause	$\bigwedge(d)$	$2^{\mathbb{L}} \to \mathbb{P}$
MIG Vertices	$\mathcal{V}(\mathcal{G})$	$\mathbb{G} \to \mathbb{L}$
MIG Set of Strong Edges	$\mathcal{E}^S(\mathcal{G})$	$\mathbb{G} \to 2^{\mathbb{L}^2}$
MIG Set of Weak Edges	$\mathcal{E}^W(\mathcal{G})$	$\mathbb{G} \to 2^{\mathbb{L}^2}$
MIG Strong Paths	$\mathcal{P}^S(\mathcal{G})$	$\mathbb{G} \to 2^{(\mathbb{L}^2)^{<\mathbb{N}}}$
MIG Weak Paths	$\mathcal{P}^W(\mathcal{G})$	$\mathbb{G} \to 2^{(\mathbb{L}^2)^{<\mathbb{N}}}$

Bibliography

Iago Abal, Jean Melo, Stefan Stănciulescu, Claus Brabrand, Márcio Ribeiro, and Andrzej Wąsowski. Variability Bugs in Highly Configurable Systems: A Qualitative Analysis. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 26(3):10:1–10:34, 2018. (cited on Page 3, 21, 64, and 120)

Mathieu Acher, Philippe Collet, Philippe Lahire, and Robert B. France. Slicing Feature Models. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 424–427. IEEE Computer Science, 2011. (cited on Page 128)

Mathieu Acher, Anthony Cleve, Philippe Collet, Philippe Merle, Laurence Duchien, and Philippe Lahire. Extraction and Evolution of Architectural Variability Models in Plugin-Based Systems. *International Journal on Software and Systems Modeling (SoSyM)*, 13(4):1367–1394, 2014. (cited on Page 16)

Mustafa Al-Hajjaji, Sebastian Krieter, Thomas Thüm, Malte Lochau, and Gunter Saake. IncLing: Efficient Product-line Testing Using Incremental Pairwise Sampling. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 144–155. ACM, 2016a. (cited on Page 3, 18, 51, 78, 79, 92, 103, 105, 119, and 120)

Mustafa Al-Hajjaji, Jens Meinicke, Sebastian Krieter, Reimar Schröter, Thomas Thüm, Thomas Leich, and Gunter Saake. Tool Demo: Testing Configurable Systems with FeatureIDE. In *Proceedings of the International Conference on Generative Programming: Concepts & Experiences (GPCE)*, pages 173–177. ACM, 2016b. (cited on Page 78, 103, 104, and 105)

Mustafa Al-Hajjaji, Sascha Lity, Remo Lachmann, Thomas Thüm, Ina Schaefer, and Gunter Saake. Delta-Oriented Product Prioritization for Similarity-Based Product-Line Testing. In *Proceedings of the International Workshop on Variability and Complexity in Software Design (VACE)*, pages 34–40. IEEE Computer Science, 2017. (cited on Page 119)

Mustafa Al-Hajjaji, Thomas Thüm, Malte Lochau, Jens Meinicke, and Gunter Saake. Effective Product-Line Testing Using Similarity-Based Product Prioritization. *International Journal on Software and Systems Modeling (SoSyM)*, 18(1):499–521, 2019. (cited on Page 78, 103, and 119)

Jérôme Amilhastre, Hélene Fargier, and Pierre Marquis. Consistency Restoration and Explanations in Dynamic CSPs-Application to Configuration. *Artificial Intelligence*, 135(1-2):199–234, 2002. (cited on Page 121)

Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016. (cited on Page 3, 64, and 90)

Sven Apel and Christian Kästner. An Overview of Feature-Oriented Software Development. *Journal of Object Technology (JOT)*, 8(5):49–84, 2009. (cited on Page 8)

Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2013. (cited on Page 1, 2, 7, 8, 11, and 15)

Lerina Aversano, Massimiliano Di Penta, and Ira D. Baxter. Handling Preprocessor-Conditioned Declarations. In *Proceedings of the International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 83–92. IEEE Computer Science, 2002. (cited on Page 122)

Don Batory. Feature Models, Grammars, and Propositional Formulas. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 7–20. Springer, 2005. (cited on Page 2, 9, 20, 121, and 122)

Don Batory, David Benavides, and Antonio Ruiz-Cortés. Automated Analyses of Feature Models: Challenges Ahead. *Communications of the ACM*, 49(12):45–47, 2006. (cited on Page 1)

Shoham Ben-David, Baruch Sterin, Joanne M. Atlee, and Sandy Beidu. Symbolic Model Checking of Product-Line Requirements Using SAT-Based Methods. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 189–199. IEEE Computer Science, 2015. (cited on Page 122)

David Benavides. *On the Automated Analysis of Software Product Lines Using Feature Models - A Framework for Developing Automated Tool Support*. PhD thesis, University of Seville, 2007. (cited on Page 13)

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 119 and 121)

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. Technical Report 2007-01, Lero, 2007. (cited on Page 19 and 121)

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 and 122)

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. Variability Modeling in the Real: A Perspective from the Operating Systems Domain. In *Proceedings of the International Conference on Automated Software Engineering (ASE)*, pages 73–82. ACM, 2010. (cited on Page 1 and 123)

Thorsten Berger, Ralf Rublack, Divya Nair, Joanne M. Atlee, Martin Becker, Krzysztof Czarnecki, and Andrzej Wąsowski. A Survey of Variability Modeling in Industrial Practice. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 7:1–7:8. ACM, 2013a. (cited on Page 1)

Thorsten Berger, Steven She, Rafael Lotufo, Andrzej Wąsowski, and Krzysztof Czarnecki. A Study of Variability Models and Languages in the Systems Software Domain. *IEEE Transactions on Software Engineering (TSE)*, 39(12):1611–1640, 2013b. (cited on Page 48 and 62)

Sara Bessling and Michaela Huhn. Towards Formal Safety Analysis in Feature-Oriented Product Line Development. In *Proceedings of the International Symposium on Foundations of Health Information Engineering and Systems (FHIES)*, pages 217–235. Springer, 2014. (cited on Page 122)

Danilo Beuche. Modeling and Building Software Product Lines with Pure::Variants. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 255–255, Salvador, Brazil, 2012. (cited on Page 121)

Goetz Botterweck, Mikolás Janota, and Denny Schneeweiss. A Design of a Configurable Feature Model Configurator. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, ICB Research Report, pages 165–168. Universität Duisburg-Essen, 2009. (cited on Page 121)

Randal E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986. (cited on Page 122)

Marco Cadoli and Francesco M. Donini. A Survey on Knowledge Compilation. *AI Communications*, 10(3-4):137–150, 1997. (cited on Page 122)

Ivan Do Carmo Machado, John D. McGregor, Yguaratã Cerqueira Cavalcanti, and Eduardo Santana De Almeida. On Strategies for Testing Software Product Lines: A Systematic Literature Review. *Journal of Information and Software Technology (IST)*, 56(10):1183–1199, 2014. (cited on Page 18 and 119)

Vasek Chvatal. A Greedy Heuristic for the Set-Covering Problem. *Mathematics of Operations Research (MOR)*, 4(3):233–235, 1979. (cited on Page 78, 79, 103, and 105)

Andreas Classen, Maxime Cordy, Patrick Heymans, Axel Legay, and Pierre-Yves Schobbens. Formal Semantics, Modular Specification, and Symbolic Verification of Product-Line Behaviour. *Science of Computer Programming (SCP)*, 80(PB): 416–439, 2014. (cited on Page 122)

Paul Clements and Linda Northrop. *Software Product Lines: Practices and Patterns*. Addison-Wesley, 2001. (cited on Page 7 and 11)

Anastasia Cmyrev and Ralf Reissing. Efficient and Effective Testing of Automotive Software Product Lines. *International Journal of Applied Science and Technology (IJAST)*, 7(2), 2014. (cited on Page 120)

Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Coverage and Adequacy in Software Product Line Testing. In *Proceedings of the International Workshop on the Role of Software Architecture for Testing and Analysis (ROSATEA)*, pages 53–63. ACM, 2006. (cited on Page 119)

Myra B. Cohen, Matthew B. Dwyer, and Jiangfan Shi. Constructing Interaction Test Suites for Highly-Configurable Systems in the Presence of Constraints: A Greedy Approach. *IEEE Transactions on Software Engineering (TSE)*, 34(5):633–650, 2008. (cited on Page 3, 21, 64, and 119)

The Kernel Development Community. KConfig Language. Website `https://www.kernel.org/doc/html/latest/kbuild/kconfig-language.html`, 2018. Visited March 13th, 2020. (cited on Page 8 and 123)

Stephen A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. ACM, 1971. (cited on Page 3)

Krzysztof Czarnecki and Ulrich Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000. (cited on Page 1, 2, 7, 8, and 9)

Krzysztof Czarnecki and Krzysztof Pietroszek. Verifying Feature-Based Model Templates Against Well-Formedness OCL Constraints. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 211–220. ACM, 2006. (cited on Page 122)

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. IEEE Computer Science, 2007. (cited on Page 9)

Adnan Darwiche and Pierre Marquis. A Knowledge Compilation Map. *Journal of Artificial Intelligence Research (JAIR)*, 17(1):229–264, 2002. (cited on Page 122)

Leonardo De Moura and Nikolaj Bjørner. Z3: An Efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 337–340. Springer, 2008. (cited on Page 121)

Leonardo De Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011. (cited on Page 121)

Xavier Devroey, Gilles Perrouin, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. Covering SPL Behaviour with Sampled Configurations: An Initial Assessment. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 59:59–59:66. ACM, 2015. (cited on Page 120)

Deepak Dhungana, Rick Rabiser, Paul Grünbacher, Klaus Lehner, and Christian Federspiel. DOPLER: An Adaptable Tool Suite for Product Line Engineering. In

Proceedings of the International Software Product Line Conference (SPLC), pages 151–152. Kindai Kagaku Sha Co. Ltd., Tokyo, Japan, 2007. (cited on Page 8)

Niklas Eén and Armin Biere. Effective Preprocessing in SAT Through Variable and Clause Elimination. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75. Springer, 2005. (cited on Page 123)

Niklas Eén and Niklas Sörensson. Temporal Induction by Incremental SAT Solving. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 89(4):543–560, 2003. (cited on Page 123)

Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 502–518. Springer, 2004. (cited on Page 13)

Emelie Engström and Per Runeson. Software Product Line Testing - A Systematic Mapping Study. *Journal of Information and Software Technology (IST)*, 53(1): 2–13, 2011. (cited on Page 3, 22, and 64)

Alireza Ensan, Ebrahim Bagheri, Mohsen Asadi, Dragan Gasevic, and Yevgen Biletskiy. Goal-Oriented Test Case Selection and Prioritization for Product Line Feature Models. In *Proceedings of the International Conference on Information Technology: New Generations (ITNG)*, pages 291–298. IEEE Computer Science, 2011. (cited on Page 120)

Faezeh Ensan, Ebrahim Bagheri, and Dragan Gasevic. Evolutionary Search-Based Test Generation for Software Product Line Feature Models. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, volume 7328 of *Lecture Notes in Computer Science*, pages 613–628. Springer, 2012. (cited on Page 120)

Katalin Fazekas, Armin Biere, and Christoph Scholl. Incremental Inprocessing in SAT Solving. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 136–154. Springer, 2019. (cited on Page 123)

Kevin Feichtinger, Johann Stöbich, Dario Romano, and Rick Rabiser. TRAVART: An Approach for Transforming Variability Models. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 8:1–8:10. ACM, 2021. (cited on Page 8 and 10)

Thiago N. Ferreira, Josiel Neumann Kuk, Aurora Pozo, and Silvia Regina Vergilio. Product Selection Based on Upper Confidence Bound MOEA/D-DRA for Testing Software Product Lines. In *Proceedings of the Congress Evolutionary Computation (CEC)*, pages 4135–4142. IEEE Computer Science, 2016. (cited on Page 120)

Thiago N. Ferreira, Jackson A. Prado Lima, Andrei Strickler, Josiel N. Kuk, Silvia R. Vergilio, and Aurora Pozo. Hyper-Heuristic Based Product Selection for Software Product Line Testing. *IEEE Computational Intelligence Magazine (CIM)*, 12(2): 34–45, 2017. (cited on Page 120)

Helson L. Jakubovski Filho, Jackson A. Prado Lima, and Silvia R. Vergilio. Automatic Generation of Search-Based Algorithms Applied to the Feature Testing of Software Product Lines. In *Proceedings of the Brazilian Symposium on Software Engineering (SBES)*, pages 114–123. ACM, 2017. (cited on Page 120)

Stefan Fischer, Roberto E. Lopez-Herrejon, Rudolf Ramler, and Alexander Egyed. A Preliminary Empirical Assessment of Similarity for Combinatorial Interaction Testing of Software Product Lines. In *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, pages 15–18. ACM, 2016. (cited on Page 119)

José A. Galindo, Mathieu Acher, Juan Manuel Tirado, Cristian Vidal, Benoit Baudry, and David Benavides. Exploiting the Enumeration of All Feature Model Configurations: A New Perspective With Distributed Computing. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 74–78. ACM, 2016. (cited on Page 122)

Hassan Gomaa. Designing Software Product Lines with the Unified Modeling Language (UML). In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 3154 of *Lecture Notes in Computer Science*, page 317. Springer, 2004. (cited on Page 8)

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. In *Proceedings of the International Conference on Economic, Technical and Organisational Aspects of Product Configuration Systems*, pages 131–138. Gamez Publishing, 2004. (cited on Page 121)

Axel Halin, Alexandre Nuttinck, Mathieu Acher, Xavier Devroey, Gilles Perrouin, and Benoit Baudry. Test Them All, Is It Worth It? Assessing Configuration Sampling on the JHipster Web Development Stack. *Empirical Software Engineering (EMSE)*, 24(2):674–717, 2019. (cited on Page 22 and 122)

Svein Hallsteinsen, Mike Hinchey, Sooyong Park, and Klaus Schmid. Dynamic Software Product Lines. *IEEE Computer*, 41(4):93–95, 2008. (cited on Page 16)

Alan Hartman. *Graph Theory, Combinatorics and Algorithms: Interdisciplinary Applications*, chapter Software and Hardware Testing Using Combinatorial Covering Suites, pages 237–266. Springer, 2005. (cited on Page 23)

Evelyn Nicole Haslinger, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using Feature Model Knowledge to Speed Up the Generation of Covering Arrays. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 16:1–16:6. ACM, 2013. (cited on Page 120)

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, and Yves Le Traon. Multi-Objective Test Generation for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 62–71. ACM, 2013. (cited on Page 119 and 120)

Christopher Henard, Mike Papadakis, and Yves Le Traon. Mutation-Based Generation of Software Product Line Test Configurations. In *Search-Based Software Engineering*, volume 8636 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2014a. (cited on Page 119 and 120)

Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-Wise Test Configurations for Software Product Lines. *IEEE Transactions on Software Engineering (TSE)*, 40(7):650–670, 2014b. (cited on Page 103, 105, and 120)

Ruben Heradio, David Fernández-Amorós, José Antonio Galindo, and David Benavides. Uniform and Scalable SAT-Sampling for Configurable Systems. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 17:1–17:11. ACM, 2020. (cited on Page 120)

Arnaud Hubaux, Yingfei Xiong, and Krzysztof Czarnecki. A User Survey of Configuration Challenges in Linux and eCos. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 149–155. ACM, 2012. (cited on Page 1 and 2)

Mikolás Janota. Do SAT Solvers Make Good Configurators? In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 2, pages 191–195. Lero Int. Science Centre, University of Limerick, 2008. (cited on Page 2, 13, 20, and 121)

Rune M. Jensen. CLab: A C++ Library for Fast Backtrack-Free Interactive Product Configuration. In *Proceedings of the International Conference on Principles and Practice of Constraint Programming (CP)*, volume 3258 of *Lecture Notes in Computer Science*, page 816. Springer, 2004. (cited on Page 122)

Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. Properties of Realistic Feature Models Make Combinatorial Testing of Product Lines Feasible. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 6981 of *Lecture Notes in Computer Science*, pages 638–652. Springer, 2011. (cited on Page 78, 79, 92, 103, 105, and 120)

Martin Fagereng Johansen, Øystein Haugen, and Franck Fleurey. An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 46–55. ACM, 2012a. (cited on Page 3, 51, 78, 79, 92, 103, 105, and 120)

Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Anne Grete Eldegard, and Torbjørn Syversen. Generating Better Partial Covering Arrays by Modeling Weights on Sub-Product Lines. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, volume 7590 of *Lecture Notes in Computer Science*, pages 269–284. Springer, 2012b. (cited on Page 120)

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 8)

Christian Kästner, Sven Apel, Syed Saif ur Rahman, Marko Rosenmüller, Don Batory, and Gunter Saake. On the Impact of the Optional Feature Problem: Analysis and Case Studies. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 181–190. Software Engineering Institute, 2009. (cited on Page 3, 21, and 64)

Andy Kenner, Christian Kästner, Steffen Haase, and Thomas Leich. TypeChef: Toward Type Checking #Ifdef Variability in C. In *Proceedings of the International SPLC Workshop Feature-Oriented Software Development (FOSD)*, pages 25–32. ACM, 2010. (cited on Page 122)

Chang Hwan Peter Kim, Eric Bodden, Don Batory, and Sarfraz Khurshid. Reducing Configurations to Monitor in a Software Product Line. In *Proceedings of the International Conference on Runtime Verification (RV)*, pages 285–299. Springer, 2010. (cited on Page 120 and 121)

Chang Hwan Peter Kim, Don Batory, and Sarfraz Khurshid. Reducing Combinatorics in Testing Product Lines. In *Proceedings of the International Conference on Aspect-Oriented Software Development (AOSD)*, pages 57–68. ACM, 2011. (cited on Page 120 and 121)

Alexander Knüppel, Thomas Thüm, Stephan Mennicke, Jens Meinicke, and Ina Schaefer. Is There a Mismatch Between Real-World Feature Models and Product-Line Research? In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 291–302. ACM, 2017. (cited on Page 1, 48, 77, and 123)

Matthias Kowal, Sandro Schulze, and Ina Schaefer. Towards Efficient SPL Testing by Variant Reduction. In *Conference of the International workshop on Variability & composition (VariComp)*, pages 1–6. ACM, 2013. (cited on Page 120)

Sebastian Krieter. Efficient Configuration of Large-Scale Feature Models Using Extended Implication Graphs. Master's thesis, University of Magdeburg, 2015. (cited on Page 25)

Sebastian Krieter. Large-Scale T-Wise Interaction Sampling Using YASA. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 29:1–29:4. ACM, 2020. (cited on Page 63)

Sebastian Krieter, Reimar Schröter, Thomas Thüm, Wolfram Fenske, and Gunter Saake. Comparing Algorithms for Efficient Feature-Model Slicing. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 60–64. ACM, 2016. (cited on Page 86 and 128)

Sebastian Krieter, Marcus Pinnecke, Jacob Krüger, Joshua Sprey, Christopher Sontag, Thomas Thüm, Thomas Leich, and Gunter Saake. FeatureIDE: Empowering Third-Party Developers. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 42–45. ACM, 2017. (cited on Page 52)

Sebastian Krieter, Thomas Thüm, Sandro Schulze, Reimar Schröter, and Gunter Saake. Propagating Configuration Decisions with Modal Implication Graphs. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 898–909. ACM, 2018. (cited on Page 25, 26, and 60)

Sebastian Krieter, Thomas Thüm, Sandro Schulze, Gunter Saake, and Thomas Leich. YASA: Yet Another Sampling Algorithm. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 4:1–4:10. ACM, 2020. (cited on Page 63)

Sebastian Krieter, Rahel Arens, Michael Nieke, Chico Sundermann, Tobias Heß, Thomas Thüm, and Christoph Seidl. Incremental Construction of Modal Implication Graphs for Evolving Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 64–74. ACM, 2021. (cited on Page 25 and 26)

Sebastian Krieter, Thomas Thüm, Sandro Schulze, Sebastian Ruland, Malte Lochau, Gunter Saake, and Thomas Leich. T-Wise Presence Condition Coverage and Sampling for Configurable Systems. arXiv:2205.15180 [cs.SE] `https://arxiv.org/abs/2205.15180`, 2022. (cited on Page 89)

Charles Krueger. BigLever Software Gears and the 3-tiered SPL Methodology. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 844–845, Montreal, Quebec, Canada, 2007. (cited on Page 121)

Charles Krueger and Paul Clements. Systems and Software Product Line Engineering with BigLever Software Gears. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 136–140, Tokyo, Japan, 2013. (cited on Page 121)

Elias Kuiter, Sebastian Krieter, Jacob Krüger, Kai Ludwig, Thomas Leich, and Gunter Saake. PClocator: A Tool Suite to Automatically Identify Configurations for Code Locations. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 284–288, 2018. (cited on Page 96 and 103)

Daniel Le Berre and Anne Parrain. The Sat4j Library, Release 2.2. *Journal of Satisfiability, Boolean Modeling and Computation*, 7(2-3):59–64, 2010. (cited on Page 13, 52, 78, 103, and 104)

Jihyun Lee, Sungwon Kang, and Danhyung Lee. A Survey on Software Product Line Testing. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 31–40. ACM, 2012. (cited on Page 3, 22, and 64)

Yu Lei, Raghu N. Kacker, D. Richard Kuhn, Vadim Okun, and James Lawrence. IPOG: A General Strategy for T-Way Software Testing. In *Proceedings of the International Conference on Engineering of Computer-Based Systems (ECBS)*, pages 549–556. IEEE Computer Science, 2007. (cited on Page 71 and 120)

Jörg Liebig, Sven Apel, Christian Lengauer, Christian Kästner, and Michael Schulze. An Analysis of the Variability in Forty Preprocessor-Based Software Product Lines.

In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 105–114. IEEE Computer Science, 2010. (cited on Page 16 and 91)

Jörg Liebig, Alexander von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the European Software Engineering Conference/Foundations of Software Engineering (ESECFSE)*, pages 81–91. ACM, 2013. (cited on Page 120, 121, and 122)

Malte Lochau, Ina Schaefer, Jochen Kamischke, and Sascha Lity. Incremental Model-Based Testing of Delta-oriented Software Product Lines. In *Proceedings of the International Conference on Tests and Proofs (TAP)*, pages 67–82. Springer, 2012. (cited on Page 119)

Roberto E. Lopez-Herrejon, Stefan Fischer, Rudolf Ramler, and Aalexander Egyed. A First Systematic Mapping Study on Combinatorial Interaction Testing for Software Product Lines. In *Proceedings of the International Workshop on Combinatorial Testing (IWCT)*, pages 1–10. IEEE Computer Science, 2015. (cited on Page 23)

Roberto Erick Lopez-Herrejon, Javier Ferrer, Francisco Chicano, Alexander Egyed, and Enrique Alba. Comparative Analysis of Classical Multi-Objective Evolutionary Algorithms and Seeding Strategies for Pairwise Testing of Software Product Lines. In *Proceedings of the Congress Evolutionary Computation (CEC)*, pages 387–396. IEEE Computer Science, 2014. (cited on Page 18, 119, and 120)

Mike Mannion. Using First-Order Logic for Product Line Model Validation. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 176–187. Springer, 2002. (cited on Page 9)

Dusica Marijan, Arnaud Gotlieb, Sagar Sen, and Aymeric Hervieu. Practical Pairwise Testing for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 227–235. ACM, 2013. (cited on Page 3, 21, 64, and 120)

Jose Luis Martinez, Tewfik Ziadi, Raul Mazo, Tegawendé F. Bissyandé, John Klein, and Yves Le Traon. Feature Relations Graphs: A Visualisation Paradigm for Feature Constraints in Software Product Lines. In *Proceedings of the Working Conference on Software Visualization (VISSOFT)*, pages 50–59. IEEE, 2014. (cited on Page 19 and 122)

Rui Angelo Matnei Filho and Silvia Regina Vergilio. A Multi-Objective Test Data Generation Approach for Mutation Testing of Feature Models. *Journal of Software Engineering Research and Development (JSERD)*, 4(1), 2016. (cited on Page 120)

Raúl Mazo, Camille Salinesi, and Daniel Diaz. VariaMos: A Tool for Product Line Driven Systems Engineering with a Constraint Based Approach. In *Proceedings of the International Conference on Advanced Information Systems Engineering (CAiSE)*, pages 147–154. CEUR-WS.org, 2012. (cited on Page 121)

John McGregor. Testing a Software Product Line. In *Testing Techniques in Software Engineering*, volume 6153 of *Lecture Notes in Computer Science*, pages 104–140. Springer, 2010. (cited on Page 3, 64, and 90)

Flávio Medeiros, Thiago Lima, Francisco Dalton, Márcio Ribeiro, Rohit Gheyi, and Baldoino Fonseca. Colligens: A Tool to Support the Development of Preprocessor-Based Software Product Lines in C. In *Proceedings of the Brazilian Conference Software: Theory and Practice (CBSoft)*. CBSOFT, 2013. (cited on Page 16)

Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. A Comparison of 10 Sampling Algorithms for Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 643–654. ACM, 2016. (cited on Page 21, 105, 106, 107, and 109)

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, and Gunter Saake. An Overview on Analysis Tools for Software Product Lines. In *Proceedings of the Workshop on Software Product Line Analysis Tools (SPLat)*, pages 94–101. ACM, 2014. (cited on Page 22 and 122)

Jens Meinicke, Thomas Thüm, Reimar Schröter, Fabian Benduhn, Thomas Leich, and Gunter Saake. *Mastering Software Variability with FeatureIDE*. Springer, 2017. (cited on Page 3, 52, 78, and 103)

Marcílio Mendonça. *Efficient Reasoning Techniques for Large Scale Feature Models*. PhD thesis, University of Waterloo, 2009. (cited on Page 2, 13, 20, 26, and 45)

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. ACM, 2008. (cited on Page 121)

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. ACM, 2009a. (cited on Page 20 and 121)

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. Software Engineering Institute, 2009b. (cited on Page 26 and 44)

Andreas Metzger, Klaus Pohl, Patrick Heymans, Pierre-Yves Schobbens, and Germain Saval. Disambiguating the Documentation of Variability in Software Product Lines: A Separation of Concerns, Formalization and Automated Analysis. In *Requirements Engineering*, pages 243–253. IEEE Computer Science, 2007. (cited on Page 122)

Jan Midtgaard, Aleksandar S. Dimovski, Claus Brabrand, and Andrzej Wąsowski. Systematic Derivation of Correct Variability-Aware Program Analyses. *Science of Computer Programming (SCP)*, 105(C):145–170, 2015. (cited on Page 122)

Jean-Vivien Millo, S. Ramesh, Shankara Narayanan Krishna, and Ganesh Khandu Narwane. Compositional Verification of Software Product Lines. In *Proceedings of the World Conference on Integrated Formal Methods (iFM)*, pages 109–123. Springer, 2013. (cited on Page 122)

Malek Mouhoub and Samira Sadaoui. Solving Incremental Satisfiability. *Artificial Intelligence Tools*, 16(1):139–147, 2007. (cited on Page 122)

Munge Development Team. Munge: A Purposely-Simple Java Preprocessor. Website `http://github.com/sonatype/munge-maven-plugin`, 2011. Visited January 11th, 2011. (cited on Page 16)

Daniel-Jesus Munoz, Jeho Oh, Mónica Pinto, Lidia Fuentes, and Don Batory. Uniform Random Sampling Product Configurations of Feature Models That Have Numerical Features. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 289–301. ACM, 2019. (cited on Page 21 and 120)

Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Ultimately Incremental SAT. In *Proceedings of the International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 206–218. Springer, 2014. (cited on Page 123)

Sarah Nadi, Thorsten Berger, Christian Kästner, and Krzysztof Czarnecki. Where Do Configuration Constraints Stem From? An Extraction Approach and an Empirical Study. *IEEE Transactions on Software Engineering (TSE)*, 41(8):820–841, 2015. (cited on Page 1 and 122)

Daren Nestor, Steffen Thiel, Goetz Botterweck, Ciarán Cawley, and Patrick Healy. Applying Visualisation Techniques in Software Product Lines. In *Proceedings of the Symposium on Software Visualization (SoftVis)*, pages 175–184. ACM, 2008. (cited on Page 19 and 122)

Michael Nieke, Jacopo Mauro, Christoph Seidl, Thomas Thüm, Ingrid Chieh Yu, and Felix Franzke. Anomaly Analyses for Feature-Model Evolution. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 188–201. ACM, 2018. (cited on Page 62)

Lina Ochoa, Oscar González-Rojas, and Thomas Thüm. Using Decision Rules for Solving Conflicts in Extended Feature Models. In *Proceedings of the International Conference on Software Language Engineering (SLE)*, pages 149–160. ACM, 2015. (cited on Page 19, 119, and 121)

Jeho Oh, Don Batory, Margaret Myers, and Norbert Siegmund. Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the International Symposium Foundations of Software Engineering (FSE)*, pages 61–71, 2017. (cited on Page 2, 21, and 120)

Jeho Oh, Paul Gazzillo, Don Batory, Marijn Heule, and Maggie Myers. Uniform Sampling from Kconfig Feature Models. Technical Report TR-19-02, The University of Texas at Austin, Department of Computer Science, 2019. (cited on Page 77, 103, 104, 105, and 120)

Sebastian Oster, Florian Markert, and Philipp Ritter. Automated Incremental Pairwise Testing of Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 196–210. Springer, 2010. (cited on Page 3, 64, 119, and 120)

Sebastian Oster, Marius Zink, Malte Lochau, and Mark Grechanik. Pairwise Feature-Interaction Testing for SPLs: Potentials and Limitations. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 6:1–6:8. ACM, 2011. (cited on Page 119)

Juliana Alves Pereira, Sebastian Krieter, Jens Meinicke, Reimar Schröter, Gunter Saake, and Thomas Leich. FeatureIDE: Scalable Product Configuration of Variable Systems. In *Proceedings of the International Conference on Software Reuse (ICSR)*, pages 397–401. Springer, 2016a. (cited on Page 20, 21, and 121)

Juliana Alves Pereira, Pawel Matuszyk, Sebastian Krieter, Myra Spiliopoulou, and Gunter Saake. A Feature-Based Personalized Recommender System for Product-Line Configuration. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 120–131. ACM, 2016b. (cited on Page 19 and 122)

Juliana Alves Pereira, Sandro Schulze, Sebastian Krieter, Márcio Ribeiro, and Gunter Saake. A Context-Aware Recommender System for Extended Software Product Line Configurations. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, pages 97–104. ACM, 2018. (cited on Page 2)

Hector Perez-Morago, Ruben Heradio, David Fernández-Amorós, Roberto Bean, and Carlos Cerrada. Efficient Identification of Core and Dead Features in Variability Models. *IEEE Access*, 3:2333–2340, 2015. (cited on Page 122)

Gilles Perrouin, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Automated and Scalable T-Wise Test Case Generation Strategies for Software Product Lines. In *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, pages 459–468. IEEE Computer Science, 2010. (cited on Page 3, 18, 64, and 119)

Gilles Perrouin, Sebastian Oster, Sagar Sen, Jacques Klein, Benoit Baudry, and Yves Le Traon. Pairwise Testing for Software Product Lines: Comparison of Two Approaches. *Software Quality Journal (SQJ)*, 20(3-4):605–643, 2012. (cited on Page 119)

Tobias Pett, Thomas Thüm, Tobias Runge, Sebastian Krieter, Malte Lochau, and Ina Schaefer. Product Sampling for Product Lines: The Scalability Challenge. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 78–83. ACM, 2019. (cited on Page 3, 62, 64, and 77)

Tobias Pett, Sebastian Krieter, Tobias Runge, Thomas Thüm, Malte Lochau, and Ina Schaefer. Stability of Product-Line Sampling in Continuous Integration. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 18:1–18:9. ACM, 2021. (cited on Page 62 and 67)

Quentin Plazar, Mathieu Acher, Gilles Perrouin, Xavier Devroey, and Maxime Cordy. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet? In *Proceedings of the International Conference on Software Testing, Verification*

and Validation (ICST), pages 240–251. IEEE Computer Science, 2019. (cited on Page 120)

Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005. (cited on Page 7, 8, and 11)

pure::systems. pure::variants. Website `http://www.pure-systems.com/products/pure-variants-9.html`, 2017. Visited May 10th, 2017. (cited on Page 121)

Dennis Reuling, Johannes Bürdek, Serge Rotärmel, Malte Lochau, and Udo Kelter. Fault-Based Product-Line Testing: Effective Sample Generation Based on Feature-Diagram Mutation. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 131–140. ACM, 2015. (cited on Page 120)

Sebastian Ruland, Lars Luthmann, Johannes Bürdek, Sascha Lity, Thomas Thüm, Malte Lochau, and Márcio Ribeiro. Measuring Effectiveness of Sample-Based Product-Line Testing. In *Proceedings of the International Conference on Generative Programming and Component Engineering (GPCE)*, pages 119–133. ACM, 2018. (cited on Page 90 and 92)

Pierre-Yves Schobbens, Patrick Heymans, and Jean-Christophe Trigaux. Feature Diagrams: A Survey and a Formal Semantics. In *Requirements Engineering*, pages 136–145. IEEE Computer Science, 2006. (cited on Page 9)

Julia Schroeter, Malte Lochau, and Tim Winkelmann. Multi-Perspectives on Feature Models. In *Proceedings of the International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 252–268. Springer, 2012. (cited on Page 128)

Reimar Schröter, Sebastian Krieter, Thomas Thüm, Fabian Benduhn, and Gunter Saake. Feature-Model Interfaces: The Highway to Compositional Analyses of Highly-Configurable Systems. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 667–678. ACM, 2016. (cited on Page 62 and 128)

Sergio Segura. Automated Analysis of Feature Models Using Atomic Sets. In *Proceedings of the International Software Product Line Conference (SPLC)*, volume 2, pages 201–207. IEEE Computer Science, 2008. (cited on Page 122)

Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wasowski, and Krzysztof Czarnecki. The Variability Model of the Linux Kernel. In *Proceedings of the Workshop on Variability Modelling of Software-intensive Systems (VaMoS)*, volume 37 of *ICB-Research Report*, pages 45–51. Universität Duisburg-Essen, 2010. (cited on Page 8)

Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse Engineering Feature Models. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 461–470. ACM, 2011. (cited on Page 77 and 105)

Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration Testing of Software Product Lines Using Compositional Symbolic Execution. In *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*, pages 270–284. Springer, 2012. (cited on Page 120 and 121)

Norbert Siegmund, Marko Rosenmüller, Martin Kuhlemann, Christian Kästner, Sven Apel, and Gunter Saake. SPL Conqueror: Toward Optimization of Non-functional Properties in Software Product Lines. *Software Quality Journal (SQJ)*, 20(3-4): 487–517, 2012. (cited on Page 119)

Julio Sincero, Horst Schirmeier, Wolfgang Schröder-Preikschat, and Olaf Spinczyk. Is the Linux Kernel a Software Product Line? In *Proceedings of the International Workshop on Open Source Software and Product Lines (OSSPL)*, pages 9–12. IEEE Computer Science, 2007. (cited on Page 123)

Joshua Sprey, Chico Sundermann, Sebastian Krieter, Michael Nieke, Jacopo Mauro, Thomas Thüm, and Ina Schaefer. SMT-Based Variability Analyses in FeatureIDE. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 6:1–6:9. ACM, 2020. (cited on Page 121)

Richard M. Stallman and Zachary Weinberg. The C Preprocessor. *Free Software Foundation*, 1987. (cited on Page 16 and 91)

Chico Sundermann, Thomas Thüm, and Ina Schaefer. Evaluating #SAT Solvers on Industrial Feature Models. In *Proceedings of the International Working Conference on Variability Modelling of Software-Intensive Systems (VaMoS)*, pages 3:1–3:9. ACM, 2020. (cited on Page 1 and 3)

Chico Sundermann, Kevin Feichtinger, Dominik Engelhardt, Rick Rabiser, and Thomas Thüm. Yet Another Textual Variability Language? A Community Effort Towards a Unified Language. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 136–147. ACM, 2021. (cited on Page 8 and 9)

Reinhard Tartler, Daniel Lohmann, Christian Dietrich, Christoph Egger, and Julio Sincero. Configuration Coverage in the Analysis of Large-Scale System Software. *ACM SIGOPS Operating Systems Review*, 45(3):10–14, 2012. (cited on Page 121)

Reinhard Tartler, Christian Dietrich, Julio Sincero, Wolfgang Schröder-Preikschat, and Daniel Lohmann. Static Analysis of Variability in System Software: The 90,000 #Ifdefs Issue. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 421–432. USENIX Association, 2014. (cited on Page 18, 90, 96, 103, 115, 119, 120, and 121)

Maurice H. ter Beek, Ferruccio Damiani, Stefania Gnesi, Franco Mazzanti, and Luca Paolini. On the Expressiveness of Modal Transition Systems with Variability Constraints. *Science of Computer Programming (SCP)*, 169:1–17, 2019. (cited on Page 122)

Sahil Thaker, Don Batory, David Kitchin, and William Cook. Safe Composition of Product Lines. In *Proceedings of the International Conference on Generative*

Programming and Component Engineering (GPCE), pages 95–104. ACM, 2007. (cited on Page 122)

Thomas Thüm. A BDD for Linux? The Knowledge Compilation Challenge for Variability. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 16:1–16:6. ACM, 2020. (cited on Page 121 and 122)

Thomas Thüm, Sven Apel, Christian Kästner, Ina Schaefer, and Gunter Saake. A Classification and Survey of Analysis Strategies for Software Product Lines. *ACM Computing Surveys (CSUR)*, 47(1):6:1–6:45, 2014a. (cited on Page 3, 21, 64, and 122)

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)*, 79:70–85, 2014b. (cited on Page 121)

Thomas Thüm, Jens Meinicke, Fabian Benduhn, Martin Hentschel, Alexander von Rhein, and Gunter Saake. Potential Synergies of Theorem Proving and Model Checking for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 177–186. ACM, 2014. (cited on Page 122)

Thomas Thüm, Sebastian Krieter, and Ina Schaefer. Product Configuration in the Wild: Strategies for Conflicting Decisions in Web Configurators. In *Proceedings of the Configuration Workshop (ConfWS)*, pages 1–8. RWTH Aachen University, 2018. (cited on Page 20)

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. Universität Duisburg-Essen, 2009. (cited on Page 122)

Grigori S. Tseytin. *Automation of Reasoning: 2: Classical Papers on Computational Logic 1967–1970*, chapter On the Complexity of Derivation in Propositional Calculus, pages 466–483. Springer, 1983. (cited on Page 85)

Mahsa Varshosaz, Mustafa Al-Hajjaji, Thomas Thüm, Tobias Runge, Mohammad Reza Mousavi, and Ina Schaefer. A Classification of Product Sampling for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 1–13. ACM, 2018. (cited on Page 3, 21, 22, 23, 92, 120, and 121)

Bart Veer and John Dallaway. *The eCos Component Writer's Guide*, 2017. Available online `http://ecos.sourceware.org/ecos/docs-3.0/pdf/ecos-3.0-cdl-guide-a4.pdf`; Visited May 10th, 2017. (cited on Page 123)

Jules White, Douglas C. Schmidt, Egon Wuchner, and Andrey Nechypurenko. Automating Product-Line Variant Selection for Mobile Devices. In *Proceedings of the International Software Product Line Conference (SPLC)*, pages 129–140, 2007. (cited on Page 119)

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 19 and 121)

Jules White, Brian Dougherty, and Douglas C. Schmidt. Selecting Highly Optimal Architectural Feature Sets with Filtered Cartesian Flattening. *Journal of Systems and Software (JSS)*, 82(8):1268–1284, 2009. (cited on Page 119)

Huilin Ye and Hanchang Liu. Approach to Modelling Feature Variability and Dependencies in Software Product Lines. *IEE Proceedings - Software*, 152(3): 101–109, 2005. (cited on Page 1)

Wei Zhang, Haiyan Zhao, and Hong Mei. A Propositional Logic-Based Method for Verification of Feature Models. In *Proceedings of the International Conference on Formal Engineering Methods (ICFEM)*, pages 115–130. Springer, 2004. (cited on Page 122)