University of Passau

Department of Informatics and Mathematics

Master's Thesis

A Comparison of Six Constraint Solvers for Variability Analysis

Author:

Martin Bauer

September 20, 2019

Examiner:

Prof. Dr.-Ing. Sven Apel

Chair of Software Engineering I

Prof. Dr. Gordon Fraser

Chair of Software Engineering II

Supervisor:

Christian Kaltenecker

Chair of Software Engineering I

Bauer, Martin:
A Comparison of Six Constraint Solvers for Variability Analysis
Master's Thesis, University of Passau, 2019.

Abstract

Variability models are widely used to specify configurable options of highly customizable software. In practice, variability models can become quite complex with thousands of configuration options and ten thousands of constraints among them. Reasoning over huge variability models is usually done by using sampling strategies which suggest a sample set, i.e., they select a small, representative subset of all valid configurations. Many sampling strategies utilize a constraint solver to identify valid configurations in the search space to create a representative sample set. Using a constraint solver which traverses the configuration space in a way that benefits the sampling strategy's logic can greatly improve this process. Likewise, a poorly chosen constraint solver can prevent the sampling strategy from computing a representative sample set. In general, both a good performance of the constraint solver and a representative sample set is an advantage for the sampling strategy.

In this work, we compare six constraint solvers (Z3, Microsoft Solver Foundation, Choco, JaCoP, Google's Operations Research Tools and OptiMathSAT) using several aspects, which are vital for practical use in variability analysis. We integrate those constraint solvers into SPL Conqueror (a software suite for variability analysis) and use them to draw sample sets from configuration spaces of different software product lines. Based on the performance of the constraint solvers and the representativity of those sample sets, we aim at providing recommendations which constraint solver should be used depending on the exact needs of the sampling strategies.

Contents

List of Figures

List of Tables

List of Acronyms

API	Application Programming Interface
ASIL	Automotive Safety Integrity Level
BDD	Binary Decision Diagram
CLR	Common Language Runtime
CSP	Constraint Satisfaction Problem
ILP	Integer Linear Programming
JVM	Java Virtual Machine
MSF	Microsoft Solver Foundation
OMT	Optimization Modulo Theories
RNG	Random Number Generator
SAT	Satisfiability
SMT	Satisfiability Modulo Theories

1. Introduction

In the context of software product lines, variability models are commonly used to model configurable software systems in terms of configuration options and relations among them. From a different point of view, those configuration options and relations can be seen as variables and constraints, respectively. A well-known tool to solve problems in such areas is the Satisfiability (SAT) solver. Unfortunately, the binary nature of the SAT solver also restricts the configuration options to binary configuration options, i.e., configuration options that can be selected or deselected. However, there can be numeric configuration options and constraints in variability models. While SAT usually is extremely efficient, huge effort is required for numeric problems to be expressed as a SAT instance. A higher level paradigm like Satisfiability Modulo Theories (SMT) or Constraint Programming can allow for a more natural expression of the problem because those paradigms are more general than SAT and support such kind of constraints out of the box. Benavides et al. go into this matter and further push into that area of research [BTRC05, BSTRC05, BSTRC06].

Additionally, variability models can become quite complex, i.e., the number of configuration options and constraints can be big. That is why reasoning on (huge) variability models usually involves a sampling step, i.e., selecting a small, representative subset of all valid configurations. Among others, Kaltenecker et al. [KGS+19] propose a sampling strategy, which relies on an off-the-shelf constraint solver. This means, that the configurations returned by a constraint solver can greatly influence the outcome of the sampling process (or at least its runtime). In most cases, the user of such sampling strategies is responsible for selecting an appropriate constraint solver. If a constraint solver is used that already traverses the configuration space in a way, that benefits the sampling strategy, the sampling process can be greatly improved. On the other hand, using a constraint solver with a non-beneficial search strategy can lead to non-representative sample sets while taking very long computation time.

In this work, we aim at investigating, how well constraint solvers qualify for the analysis of software product lines by analyzing the representativity of the sample sets and the performance of the constraint solvers.

In summary, our contributions are as follows:

- We determine the requirements for our scenario and select several constraint solvers which fulfill those requirements.
- We integrate four constraint solvers into the variability analysis tool *SPL Conqueror*: Choco[1], JaCoP[2], OR-Tools[3], and OptiMathSAT[4].
- We perform an empirical study to compare the aforementioned constraint solvers together with the already integrated ones — Z3[5] and Microsoft Solver Foundation[6] — based on the solutions and the time it took to find them.
- We provide several rankings, which illustrate the abilities of the constraint solvers in numerous aspects, such as the representativity of the sample sets and the computation performance.
- We find, that JaCoP and Choco perform well in all aspects and recommend to replace Z3 with JaCoP as the default constraint solver for SPL Conqueror.

[1] http://www.choco-solver.org
[2] https://osolpro.atlassian.net/wiki/spaces/JACOP/overview
[3] https://developers.google.com/optimization
[4] http://optimathsat.disi.unitn.it/index.html
[5] https://github.com/Z3Prover/z3
[6] https://www.nuget.org/packages/Microsoft.Solver.Foundation

2. Background

In this chapter, we describe all relevant background knowledge, which is necessary to understand and follow the concepts, that we use in this work.

2.1 Constraint Solver

Satisfiability is the basic and ubiquitous problem of determining if a (boolean) formula has a model, i.e., can the variables of the formula be successively replaced by some values (e.g., `TRUE` or `FALSE`) in such a way that the entire formula evaluates to `TRUE`. A supporting (arithmetical) theory captures the meaning of those formulas.

Satisfiability Modulo Theories (SMT) solvers check the satisfiability of formulas built from boolean variables and operations by relying on efficient satisfiability procedures for propositional logic — the core concept of Satisfiability (SAT) solvers [dMB11]. Modern SAT procedures can check formulas with hundreds of thousands of variables and similar progress has been observed for SMT solvers [MZ09]. They have a wide range of applications in hardware and software verification, static checking, constraint solving, planning, scheduling, test case generation, and computer security. de Moura et al. [dMDS07] give a brief overview of the theory behind SAT and SMT solving, and present different key algorithms.

A more general approach is used for Constraint Satisfaction Problem (CSP) solvers. A CSP consists of a set of variables with domains and a set of constraints restricting the values of the variables. CSP solvers use different techniques like backtracking, constraint propagation, and local search to find solutions on finite domains [FW74, VK86].

Throughout this work, we use the term *constraint solvers* to refer to both types of solvers, regardless of their background theory.

2.2 Variability Model

Modern software systems usually provide a large number of configuration options to tailor the product to the needs of the customers. These configuration options enable

the user to change the behavior of the system, tweak computations or modify parts of the program. For instance, a data compression tool can provide two algorithms, which mutually exclude each other. Today, there are hundreds of configuration options in most software systems, which can be combined in various ways. However, not all combinations of configuration options are valid.

Let $\mathcal{O}$ be the set of all configuration options and let $\mathcal{C}$ be the set of all valid configurations. A configuration $c \in \mathcal{C}$ can be presented as a function $c\colon \mathcal{O} \to \mathbb{R}$ which assigns a value to every configuration option. For binary configuration options, the range is restricted to $\{0, 1\}$:

$$c(o) = \begin{cases} 1, & \text{configuration option } o \text{ is selected,} \\ 0, & \text{otherwise.} \end{cases}$$

For numeric configuration options, c returns a number in the range of the corresponding configuration option [SGAK15]. In this work, however, we only consider binary configuration options. This is no restriction for the variability models because every numeric configuration can be converted to an alternative group of binary configuration options. A *variability model* refers to the textual or visual representation that defines the configuration options of the configurable system and the relationships thereof.

A *variability diagram* is a graphical representation of a variability model, which uses hierarchical tree structures. Each node of the tree represents a configuration option in the corresponding model. A parent-child relation indicates that the child configuration option is only selectable if the parent configuration option is selected, as well. Parent nodes are used to define more general concepts, whereas child nodes specialize those abstractions. Several graphical notations illustrate additional constraints, like for instance the information if an option is mandatory or optional [ABKS16]. In Figure 2.1, we give an example of a variability diagram for a hypothetical system. The alternative group for security forces the selection of either `Standard` or `High`. The optional configuration option `Logging` enables logging functionality. Furthermore, there is a memory limit in this system, which can have three (numeric) values.

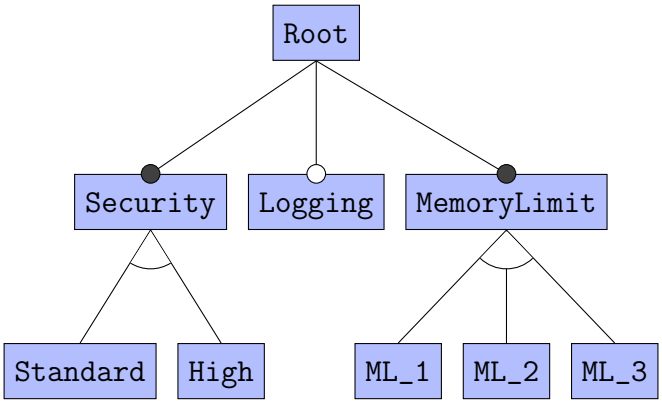


Figure 2.1: A visual representation of a hypothetical system. Mandatory and optional configuration options are represented by a filled and an empty circle, respectively. The edges between a parent and its children are connected with an empty arc if it is an alternative group, where only one option can be selected at a time.

As described above, we only focus on binary variability models, hence, the numeric configuration option is converted into a binary alternative group where each child represents the selection of a single numeric value.

2.3 SPL Conqueror

Given a variability model with a large number of configuration options and constraints, it can be difficult to find a configuration that performs as desired. Siegmund et al. [SGAK15] propose to build a so-called *performance-influence model*, which describes how configuration options and their interactions influence the performance of the system. This mathematical model does not only give insight into the (complex) interactions among configuration options but also enables performance predictions for every possible configuration and subspace of the configuration space. For instance, the inferred performance-influence model can be used to identify the best performing configuration under certain constraints (e.g., a specific configuration option needs to be enabled). It is also a valuable tool to check if the system behaves as expected by comparing it with the mental model of the developer.

Siegmund et al. infer the performance-influence model for a given configurable system in a black-box manner from a series of measurements of a set of sample configurations using machine learning. They benchmark a given system multiple times using different configurations and learn the influence of those configuration options and their interactions from the differences among the measurements. They built their approach on top of *SPL Conqueror*[1] which provides several sampling strategies covering both binary and numeric configuration options. Many of those sampling strategies make use of a constraint solver to find the best sample set.

For instance, *random sampling* is one of the most basic strategies. It randomly assigns a value to every configuration option and uses a constraint solver to check, if this assignment represents a valid configuration based on the variability model. However, configuration spaces are often highly constrained, which means that random sampling can become challenging because most random samples do not satisfy the constraints [LVRK+13]. For instance, Liebig et al. point out, that the Linux kernel can be configured with about 10 000 compile-time configuration options, which leads to possibly billions of variants that can be generated and compiled on demand. Liebig et al. compute 1 000 000 random configurations and do not find a single configuration that fulfills all variability model constraints of the Linux kernel.

For such cases, SPL Conqueror provides more sophisticated sampling algorithms, such as *distance-based sampling* by Kaltenecker et al. [KGS+19]. At the heart of most sampling strategies lies the constraint solver, which is utilized to find new, valid configurations, for example, when given a partial configuration with exactly n selected configuration options. Every constraint solver is used as a black box and thus, can return any configuration which fulfills the given constraints, irrespective of whether or not it benefits the sampling strategy. This means, that the choice of constraint solvers can greatly influence the quality and performance of the sampling strategies.

[1] https://github.com/se-passau/SPLConqueror

3. Experiment Setup

In Section 3.1, we present the obligatory requirements that have to be fulfilled by a constraint solver. Afterward, in Section 3.2, we present the constraint solvers which satisfy all requirements. In Section 3.3, we introduce our research questions regarding the comparison of the constraint solvers and describe how we attempt to answer the research questions in Section 3.4. Finally, in Section 3.5, we present the software systems that we use for the comparison.

3.1 Constraint Solver Requirements

A constraint solver has to fulfill several criteria to be in line for integration into SPL Conqueror. Some of them represent basic functionality that every constraint solver offers by definition, while others are included to be able to extend the abilities of SPL Conqueror in the future. For instance, support for numeric variability models is currently in an experimental state, i.e., constraint solvers should be able to work with numeric values, even if we do not use them in this work.

Boolean and Integer[†] Domains Constraints are typically specified over specific domains. Boolean domains (for boolean configuration options) can be handled more efficiently than integer domains (for numeric configuration options).

Satisfiability Checking As one of the most basic requirements, a constraint solver must be able to decide, if a given formula is satisfiable or not.

Model Generation If a formula is satisfiable, the constraint solver must be able to find a valid assignment for all variables in a formula.

Optimization[†] Optimizing a given formula is a key part of several sampling strategies in SPL Conqueror, e.g., given a partial configuration, it has to find a configuration with as few selected configuration options as possible.

[†]We do not use this functionality in this work but chose to require it anyway, because SPL Conqueror will take advantage of this requirement in the future or in scopes outside of this work.

Platform Independence Since SPL Conqueror is available for all major operating systems, every constraint solver has to support Linux, macOS, and Windows.

Binary API Since we aim at using the constraint solver in SPL Conqueror, every constraint solver must provide a binary API. We explicitly allow constraint solvers written in other programming languages than C#.

3.2 Constraint Solver Candidates

We use well-known competitions such as the SMT Competition [HNRW19] (international satisfiability modulo theories competition) and the MiniZinc Challenge [TS19] (world-wide competition of constraint programming solvers) to find current state-of-the-art constraint solvers. Both competitions are held annually and thus provide a good overview of the many different constraint programming solvers available to use. Due to our specific needs as described in Section 3.1 many award-winning constraint solvers were not applicable for our use case. See Chapter A for a list of all constraints solvers that we considered to use. Eventually, we selected six constraint solvers based on the 2018's results of these competitions, which are listed in Table 3.1.

Table 3.1: Overview of all constraint solvers in SPL Conqueror. The first two constraint solvers have already been part of SPL Conqueror before this thesis; the other four were integrated during the work for this thesis.

Constraint Solver	Version	Programming Language
Z3 Theorem Prover	4.8.1	C# bindings for C++
Microsoft Solver Foundation	3.1	C#
Choco	4.10	Java
JaCoP	4.6	Java
Google's Operations Research Tools	7.0	C++
OptiMathSAT	1.6.3	C

Z3 Theorem Prover

Z3 [Res19] is an open-source SMT solver from Microsoft Research. It is targeted at solving problems that arise in software verification and software analysis [dMB08]. Since its first external release in September 2007, Z3 has gone through three major release cycles and was open-sourced in October 2012. One notable change was the addition of optimization support in 2015 [BPF15]. The Z3 Theorem Prover is written in C++ but offers various bindings for different programming languages, including .NET, Java and Python. It has won many disciplines in the SMT Competition 2018.

Microsoft Solver Foundation

The Microsoft Solver Foundation (MSF) [Mic19] is a .NET library for mathematical programming, modeling, and optimization. It uses a declarative programming

model, consisting of simple compatible elements that are solved by built-in or third-party constraint solvers that employ operations research, metaheuristic, local search, and combinatorial optimization techniques. The first version was published in November 2008 and further extended in the following years. Since 2012, the Solver Foundation team has not been active on the dedicated forums and later announced that there will not be further standalone releases of the Solver Foundation[1].

Choco

Choco [PFL19] is an open-source constraint programming library written in Java. It originated from an early implementation (written in Claire) within the OCRE project[2] — a national initiative for an open constraint solver for both teaching and research. In 2003, it has been rewritten in Java for portability and to allow easier use for newcomers. For maintenance reasons, Choco has been completely rewritten in 2011, which has shown significant performance improvements. In the MiniZinc Challenge 2018, Choco has been awarded "Silver" and "Bronze" in several disciplines.

JaCoP

JaCoP [KS19] (**Ja**va **Co**nstraint **P**rogramming) is an open-source constraint programming solver written in Java. The development began in 2001 and is still continuously under development. Today, JaCoP provides a significant number of constraints to facilitate efficient modeling. It also provides a modular search design to help the user tailor the search to specific characteristics of the problem being addressed. JaCoP participated in several MiniZinc Challenges in the last years and has been awarded "Silver" many times.

Google's Operations Research Tools

Google's Operations Research Tools [Goo19] (a.k.a. OR-Tools) is an open-source software suite for solving combinatorial optimization problems. It is written in C++ but also provides bindings for Python, C#, and Java. The suite contains a constraint programming solver, a unified interface to several linear programming and mixed-integer programming solvers (e.g., GLOP, GLPK, and SCIP), several Knapsack algorithms, and various graph algorithms (e.g., shortest paths, min-cost flow, linear sum assignment). OR-Tools have been awarded "Gold" in many disciplines in the MiniZinc Challenge 2018.

OptiMathSAT

OptiMathSAT [ST19] is an Optimization Modulo Theories (OMT) solver (an umbrella term for SAT solvers which support optimizations). It builds on the basis of the SMT solver MathSAT 5 [GCR19] (written in C) and regularly synchronizes with its development progress. Compared to MathSAT 5, OptiMathSAT adds support for incremental multi-objective optimization over linear arithmetic objective functions.

[1] https://nathanbrixius.wordpress.com/2012/05/25/no-more-standalone-releases-of-microsoft-solver-foundation

[2] https://www.ocre-project.eu

3.3 Research Questions

Our goal is to compare the six constraint solver listed in Section 3.2 using several aspects. The main task for a constraint solver in SPL Conqueror is to find a sample set, i.e., to find valid configurations in the entire search space. To be used in practice, it is vital for every constraint solver to deliver representative results while at the same time be performant. Hence, we consider both the representativity of the sample set and the computation performance when comparing constraint solvers. To this end, we aim at answering the following research questions.

3.3.1 Representativity

The first part of the research questions deals with the quality of the sample set, which is defined by the representativity of the configurations (in the sampling set) with respect to the entire configuration space.

RQ 1.1

How representative are sample sets with respect to the whole population in terms of the cardinality distribution of the configurations?

Since a configuration is a set of selected configuration options, the cardinality of a configuration is defined by the cardinality of the option set, i.e., the number of selected configuration options. The distribution of the cardinalities then gives a view on the rate of interactions within the sampling set. An *interaction* among two or more selected configuration options describes the potential influence, they can exercise, which can impact the performance or correctness of the software product line. If the cardinality distribution has only small/big cardinalities, we can assume that the rate of interactions is small/big, as well.

However, this property alone is not enough to define representativity, since this does not cover the selection of different configuration options. By only creating a sample set with good coverage of the configuration cardinalities, the constraint solver may still miss configuration options. Hence, another criterion is to cover the configuration options themselves.

RQ 1.2

How representative are sample sets with respect to the whole population in terms of the configuration option frequency?

The distribution of the individual (selected) configuration options gives information, how often a configuration option is selected within the sampling set. This allows us to see if a constraint solver varies at using different configuration options within the sampling process. Variation is important here because this ensures better coverage of the entire configuration space and thus leads to a more representative sample set.

RQ 1.3

How robust is the representativity of the sample sets in terms of randomness?

As a cross-cutting concern, randomness is the third component in the research questions related to representativity. Many constraint solvers make use of a Random Number Generator (RNG), which is often used to select the value, which is assigned to the decision variables in the constraint propagation phase. A robust constraint solver can produce a representative sample set independently from the exact numbers returned by the RNG.

3.3.2 Performance

The second part of the research questions focuses on the performant computation of configurations because even the best results can become useless if it takes excessively long to compute them.

RQ 2.1

How fast can a constraint solver find all configurations?

There are many sampling strategies in SPL Conqueror, which all use the constraint solver differently. For instance, the *solver-based sampling strategy* uses the constraint solver to find n configurations. The *distance-based sampling strategy* asks for configurations, which have a specific set of configuration options selected while at the same time expects that no more than k configuration options are selected. Due to the different needs by the sampling strategies, we refrained from measuring the performance to find one or few configurations and instead use the traversal time of the entire configuration space for reference.

RQ 2.2

How robust is the performance of a constraint solver in terms of randomness?

Similar to RQ 1.3, we want to know if the seed for the RNG influences on the performance of the constraint solvers. Constraint solvers which are not affected in their performance by the RNG are more robust and produce more reliable results. Additionally, this allows for more accurate performance estimations, independently of the exact random seed.

3.4 Operationalization

In the following section, we will explain our approach of investigating, evaluating and answering the previously defined research questions.

Let $\mathcal{O}$ be the set of all configuration options and let $\mathcal{C}$ be the set of all valid configurations as described in Section 2.2. We use the constraint solvers to find subsets of $\mathcal{O}$,

which corresponds to the usual procedure realized by sampling strategies. Those sample sets can be of any size, but, in general, represent a specific portion of the configuration space (e.g., 5%, 10%). To answer our research questions, we use four sample sizes and five random seeds.

$$\mathcal{S} \coloneqq \{0.05, 0.1, 0.2, 0.5\} \qquad \mathcal{R} \coloneqq \{1, 2, 3, 4, 5\}$$

We denote a sample set with the size "20% of the whole population", which was computed by a constraint solver with random seed 3 by $\texttt{Sample}(0.2, 3) \subseteq \mathcal{P}(\mathcal{C})$. Note, that sampling all configurations always results in the whole population regardless of the random seed:

$$\forall r \in \mathcal{R} \colon \texttt{Sample}(1, r) = \mathcal{C}$$

To answer **RQ 1.1**, we focus on the cardinality distribution of the configurations in the sample set. The cardinality of a configuration $c \in \mathcal{C}$ is defined by the number of selected configuration options.

$$\texttt{Card}(c) \coloneqq \big|\{o \in \mathcal{O} \mid c(o) = 1\}\big| \tag{3.1}$$

The frequency of a cardinality $k \in \mathbb{N}$ is defined by the number of configurations in a sample set with that exact cardinality.

$$\texttt{Freq}_1(k, s, r) \coloneqq \big|\{c \in \texttt{Sample}(s, r) \mid \texttt{Card}(c) = k\}\big| \text{ where } s \in \mathcal{S}, r \in \mathcal{R} \tag{3.2}$$

To be able to compare the frequencies across all subject systems, we define the relative frequency of a cardinality $k \in \mathbb{N}$ by its frequency relative to the size of the sample set.

$$\texttt{RelFreq}_1(k, s, r) \coloneqq \frac{\texttt{Freq}_1(k, s, r)}{\big|\texttt{Sample}(s, r)\big|} \text{ where } s \in \mathcal{S}, r \in \mathcal{R} \tag{3.3}$$

This way, we are able to draw conclusions about the rate of interactions within a sample set. To be able to compare the constraint solvers among each other, we compute the difference of the relative frequency for a cardinality $k \in \mathbb{N}$ in a sample set and the whole population.

$$\texttt{Diff}_1(k, s, r) \coloneqq \texttt{RelFreq}_1(k, s, r) - \texttt{RelFreq}_1(k, 1, r) \text{ where } s \in \mathcal{S}, r \in \mathcal{R} \tag{3.4}$$

Finally, we take the mean value for both frequencies to compensate for the randomness of the RNG.

$$\texttt{AvgFreq}_1(k, s) \coloneqq \frac{\sum_{r \in \mathcal{R}} \texttt{Freq}_1(k, s, r)}{|\mathcal{R}|} \text{ where } s \in \mathcal{S} \tag{3.5}$$

$$\texttt{AvgDiff}_1(k, s) \coloneqq \frac{\sum_{r \in \mathcal{R}} \texttt{Diff}_1(k, s, r)}{|\mathcal{R}|} \text{ where } s \in \mathcal{S} \tag{3.6}$$

To answer **RQ 1.2**, we focus on the distribution of the configuration options themselves. The frequency of a configuration option $o \in \mathcal{O}$ is defined by the number of configurations in a sample set containing that exact configuration option.

$$\texttt{Freq}_2(o, s, r) \coloneqq \big|\{c \in \texttt{Sample}(s, r) \mid c(o) = 1\}\big| \text{ where } s \in \mathcal{S}, r \in \mathcal{R} \tag{3.7}$$

To be able to compare the frequencies across all subject systems, we define the relative frequency of a configuration option $o \in \mathcal{O}$ by its frequency relative to the size of the sample set.

$$\texttt{RelFreq}_2(o, s, r) \coloneqq \frac{\texttt{Freq}_2(o, s, r)}{\left|\texttt{Sample}(s, r)\right|} \text{ where } s \in \mathcal{S}, r \in \mathcal{R} \tag{3.8}$$

Here we can see if the constraint solver varies in the choice of configuration options within the sampling process. Here we can see if the constraint solver varies in the choice of configuration options within the sampling process. To be able to compare the constraint solvers among each other, we compute the difference of the relative frequency for a configuration option $o \in \mathcal{O}$ in a sample set and the whole population.

$$\texttt{Diff}_2(o, s, r) \coloneqq \texttt{RelFreq}_2(k, s, r) - \texttt{RelFreq}_2(o, 1, r) \text{ where } s \in \mathcal{S}, r \in \mathcal{R} \tag{3.9}$$

Finally, we take the mean value for both frequencies to compensate for the randomness of the RNG.

$$\texttt{AvgFreq}_2(o, s) \coloneqq \frac{\sum_{r \in \mathcal{R}} \texttt{Freq}_2(o, s, r)}{|\mathcal{R}|} \text{ where } s \in \mathcal{S} \tag{3.10}$$

$$\texttt{AvgDiff}_2(o, s) \coloneqq \frac{\sum_{r \in \mathcal{R}} \texttt{Diff}_2(o, s, r)}{|\mathcal{R}|} \text{ where } s \in \mathcal{S} \tag{3.11}$$

To answer **RQ 1.3**, we focus on the results of the previously described research questions, since we already perform each sampling procedure multiple times using different random seeds. We take the standard deviation of both difference functions (Equation 3.4 and Equation 3.9) for a sample size $s \in \mathcal{S}$ to see, if the results are affected by the randomness.

$$\sigma_{r \in \mathcal{R}}\big(\texttt{Diff}_1(k, s, r)\big) \text{ where } k \in \mathbb{N} \tag{3.12}$$

$$\sigma_{r \in \mathcal{R}}\big(\texttt{Diff}_2(o, s, r)\big) \text{ where } o \in \mathcal{O} \tag{3.13}$$

To answer **RQ 2.1**, we measure the time it takes to find all valid configurations, i.e., the time it takes to traverse the entire search space. Due to the different programming languages and runtime environments (native, JVM and CLR), we add a warmup phase for every constraint solver.

$$\texttt{Runtime}\big(\texttt{Sample}(1, r)\big) \coloneqq \begin{cases} \text{first, run warmup phase,} \\ \text{then, measure runtime for } \texttt{Sample}(1, r) \end{cases} \tag{3.14}$$

We perform the measurements five times and use the mean runtime for our evaluation to compensate for measurement errors.

$$\texttt{AvgRuntime}\big(\texttt{Sample}(1, r)\big) \coloneqq \frac{\sum_{1 \leq 5} \texttt{Runtime}\big(\texttt{Sample}(1, r)\big)}{5} \tag{3.15}$$

Finally, we take the mean runtime to compensate for the randomness of the RNG.

$$\frac{\sum_{r \in \mathcal{R}} \texttt{AvgRuntime}\big(\texttt{Sample}(1, r)\big)}{|\mathcal{R}|} \tag{3.16}$$

To answer **RQ 2.2**, we focus on the results of the previously described research question RQ 2.1, since we already perform each sampling procedure multiple times using different seeds. We take the standard deviation of the runtime values (Equation 3.15) to see if the results are affected by the randomness.

$$\sigma_{r \in \mathcal{R}} \Big(\texttt{AvgRuntime}\big(\texttt{Sample}(1, r)\big) \Big) \tag{3.17}$$

We aim at giving recommendations for every research question regarding the choice of constraint solver. The Mann-Whitney U test [MW47] enables us to rank the constraint solvers in a way, that respects all subject systems, sample sizes, and random seeds. Note, that for this test, we use the absolute values from Equation 3.4 and Equation 3.9, because it does not matter if a constraint solver under- or overrepresents configuration options. The simple fact of deviating from the whole population is deciding, if a constraint solver can produce a better sample set than another one.

Experimental Dependencies

In our experiments, the *independent variables* are the subject systems, the sample sizes, and the random seeds for the random number generator. The *dependent variables* are listed below.

- the distribution of the cardinalities for RQ 1.1
- the distribution of the configuration options for RQ 1.2
- the standard deviation values for both frequency distributions for RQ 1.3
- the runtime values for RQ 2.1
- the standard deviation values for the runtime values for RQ 2.2

3.5 Subject Systems

In our experiments, we consider 14 real-world configurable software systems from different domains and of different sizes. In Table 3.2, we provide an overview of the subject systems.

Table 3.2: Overview of the subject systems including domain, number of configuration options ($|O|$) and number of valid configurations ($|C|$).

Subject System	Domain	$\lvert O \rvert$	$\lvert C \rvert$
7z	File archive utility	44	68 640
Apache	Web server	19	580
Berkeley DB-C	Embedded database	18	2 560
Brotli	Compression tool	30	180

Continued on next page

Continued from previous page

Subject System	Domain	$\|O\|$	$\|C\|$
Dune	Multigrid solver	32	2 304
ExaStencils	Code Generator	47	86 058
HIPA^{cc}	Image processing	54	13 485
HSQLDB	Database Management System	18	864
Java GC	Garbage collector	39	193 536
LLVM	Compiler infrastructure	17	65 536
Polly	Code optimizer	40	60 000
TriMesh	Multigrid system	68	239 360
VP9	Video encoder	42	216 000
x264	Video encoder	16	1 152

7-Zip (7z) is a file archiver written in C++. Configuration options include various compression methods, different sizes of the dictionary, and several compression options, for example, whether or not timestamps for files should be included.

Apache is an HTTP server. It ships with a selection of Multi-Processing Modules (e.g., `prefork` and `worker`) and includes several directives to set the limit on the number of simultaneous requests that will be served.

Berkeley DB-C is an embedded database engine written in C. We consider configuration options defining, among others, the page and cache size or the use of encryption.

Brotli is a generic-purpose lossless compression tool. Configuration options include the size of the sliding window and the compression level.

Dune is a geometric multigrid solver for partial differential equations. As configuration options, we consider different algorithms for smoothing and different numbers of pre-smoothing and post-smoothing steps to solve Poisson's equation.

ExaStencils is a highly automatic code generator for a large variety of efficient implementations via the use of domain-specific knowledge. For instance, it offers configuration options for the polyhedron model, which is used for loop parallelization.

The HIPA^{cc} solver is an image processing framework written in C++. It provides, among others, configuration options for different numbers of pixels calculated per thread and different types of memory (e.g., texture, local) as configuration options.

HSQLDB is a relational database engine written in Java. We consider configuration options defining transaction control, cache parameters and the logging mechanism.

Java GC is the garbage collector of the Java VM, which provides several configuration options, such as disabling the explicit garbage collection call, modifying the adaptive garbage collection boundary, and adjusting the policy size.

LLVM is a popular compiler infrastructure written in C++. Configuration options that we considered concern code optimization, such as enabling inlining, jump threading, and dead code elimination.

Polly is a loop optimizer that rests on top of LLVM. It provides various configuration options that define, for example, whether or not code should be parallelized or the choice of the tile size.

TriMesh is a library for the usage and manipulation of 3D triangle meshes. It comes with many configuration options to control pre-smoothing and post-smoothing steps.

VPXENC (VP9) is a video encoder which uses the VP9 video coding format. It provides different configuration options, such as adjusting the quality and bitrate of the coded video, as well as the number of threads to use.

x264 is a video encoder for the H.264 compression format. Relevant configuration options include the number of reference frames, enabling or disabling the default entropy encoder, and the number of frames for rate control and lookahead.

4. Evaluation

We used 14 real-world configurable software systems from different domains and of different sizes to increase our external validity and evaluate the constraint solvers. Concerning this vast amount of data, we refrain from depicting all individual results and instead focus on important parts to illustrate the big picture. Furthermore, we provide several rankings for all constraint solvers that consider all 14 subject systems.

4.1 Representativity

The quality of a sample set describes, how well the entire configuration space is represented. As described in Section 3.4, we use two metrics to evaluate those sample sets: the distribution of the cardinalities and the distribution of the configuration options.

For this section, we focus on the file archive utility 7-Zip (7z) to present our results. Its variability model is illustrated in Figure 4.1. 7-Zip offers configuration options for various compression methods such as BZip2 and Deflate, different sizes for the compression dictionary, and several compression options such as, whether or not timestamps for files should be included. In total, 7z has 44 (binary) configuration options and 68 640 valid configurations.

As described in Section 3.4, we sampled multiple subsets of the configuration space and analyzed the resulting sample sets. In more detail, Z3, Choco, JaCoP and OptiMathSAT each had to compute 20 sample sets (four sample sizes in $\mathcal{S}$ and five random seeds in $\mathcal{R}$). MSF and OR-Tools had to compute four sample sets (four sample sizes in $\mathcal{S}$) because they do not support a custom random seed. We now compare these sample sets to the whole population to draw conclusions about their representativity.

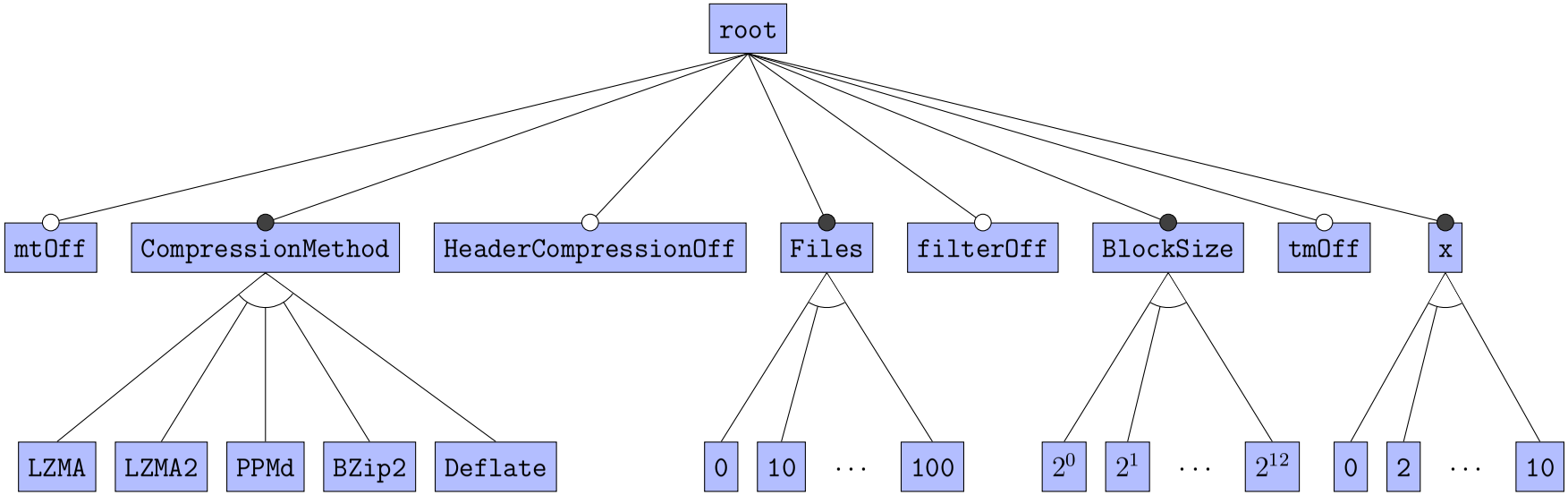


Figure 4.1: Variability model of `7z` consists of four optional configuration options and one alternative group (`CompressionMethod`). `Files`, `BlockSize` and `x` are also represented via alternative groups, because in the initial (numeric) variability model, those have been of numeric type and were converted into (binary) alternative groups. In total, `7z` has 44 configuration options and 68 640 valid configurations.

RQ 1.1

How representative are sample sets with respect to the whole population in terms of the cardinality distribution of the configurations?

In Figure 4.2, we illustrate the cardinality distribution of the configurations for all four sample set sizes (see Equation 3.5). For this subject system, there exist configurations with 4 to 8 selected configuration options (x-axis). For instance, there are 25 740 configurations with cardinality 6 in the whole population (sample size 100%). The graph representing the whole population (black line) is the same for all six constraint solvers, as its shape is solely defined by the variability model and not influenced by any external factors such as, for example, the constraint solvers. The actual sample sets (5%, 10%, 20%, and 50%) can have a different shape depending on the constraint solver. An optimal constraint solver (concerning the cardinality distribution) would compute a sample set which has a similar cardinality distribution as the whole population. This comes from the fact, that an optimal constraint solver uniformly selects configurations from the entire search space, which then have a cardinality distribution corresponding to the distribution of the whole population. This is not the case, as the graphs of MSF are shifted to the left, i.e., MSF prefers configurations with only a few selected configuration options.

To be able to compare the frequencies across all subject systems and to see the difference to the whole population, we present the difference of the relative frequencies in Figure 4.3. Again, for MSF, the configurations with less than 6 selected configuration options are overrepresented for all sample sizes, whereas the configurations with at least 6 selected configuration options do not occur as often as they do in the whole population. A similar pattern is seen with the sample set computed by Z3. Choco, JaCoP, and OptiMathSAT, on the other hand, were able to sample a very good subset of the configuration space, that is very similar to the whole population. In most cases (all constraint solvers, all cardinalities), bigger sample sizes imply less deviation from the whole population. In particular, this can be seen in those cases with a bigger difference in the smaller sample sizes like MSF and Z3.

Comparing the average difference from the whole population, Choco and JaCoP, and OptiMathSAT compute the best sample sets concerning the configuration cardinality. MSF computes a less-than-ideal sample set (difference greater than 20%), even when sampling 20% of the whole population.

Cardinality Distribution — Summary

We presented the cardinality distribution for several sample sizes for 7z. We saw that some sample sets are representative for the whole population, while others deviate heavily. This is particularly the case when only a small portion ($\approx 5\%$) of the entire configuration space is sampled.

To summarize the results concerning the cardinality distribution, we provide a ranking for all six constraint solvers, which is based on the deviation from the whole population. For this ranking, we took all subject systems and all random seeds into

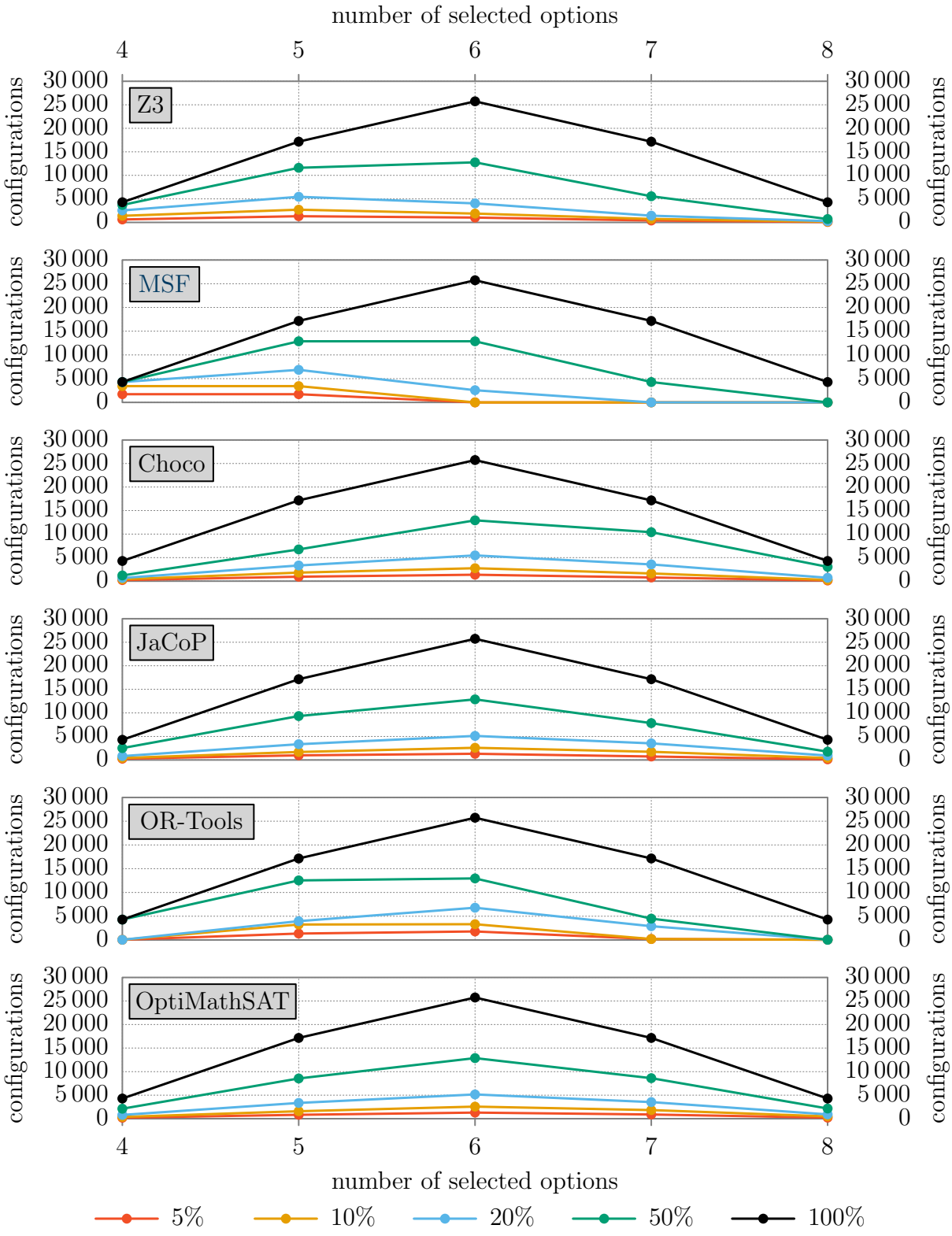
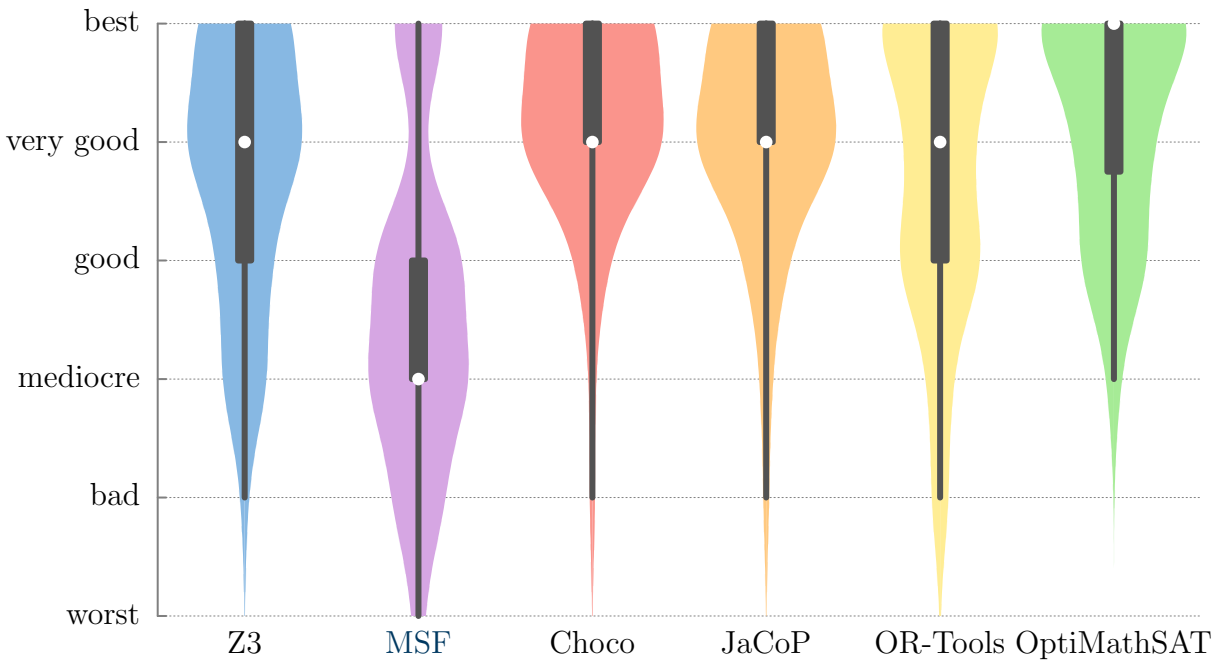


Figure 4.2: Cardinality distribution for 7z. We illustrate the average cardinality across all random seeds (see Equation 3.5). Every line represents a different sample size relative to the whole population.

Figure 4.3: Relative frequency difference of the cardinalities for 7z. Every bar represents a different sample size relative to the whole population and illustrates the average relative frequency across all random seeds (see Equation 3.6). The whiskers indicate the standard deviation caused by the random seeds (see Equation 3.12).



(a) Distribution of the rankings. By using the Mann-Whitney U test, we take all 14 subject systems, 4 sample sizes and 5 randoms seeds into account. The white dots represent the median rank for every constraint solver.

Cardinality Distribution Ranking

1. OptiMathSAT, Choco, JaCoP
2. OR-Tools, Z3
3. MSF

(b) On average, OptiMathSAT, Choco, and JaCoP compute the best sample sets concerning the distribution of the cardinalities. OR-Tools and Z3 both can compute a sample set just as good, but there are cases, where they fail to do so. MSF does not deliver representative results.

Figure 4.4: Constraint solver ranking based on the cardinality distribution.

account and used the Mann-Whitney U test to decide if one sample set deviates less than another one. The ranking is shown in Figure 4.4.

The worst constraint solver concerning the cardinality distribution is MSF. 75% of the sample sets computed by MSF are less representative than those computed by any other constraint solver. OptiMathSAT, on the other hand, is the best constraint solver, when sample sets are required where the configurations have similar cardinalities compared to the whole population. The other constraint solvers behave almost identical in the average case, whereby Choco and JaCoP compute better results in edge cases.

RQ 1.2

How representative are sample sets with respect to the whole population in terms of the configuration option frequency?

In Figure 4.5, we illustrate the configuration option frequency in a sample set for all four sample sizes (see Equation 3.10). As illustrated in Figure 4.1, the variability model for `7z` contains 44 binary configuration options. Every variability model has a root option, which is always part of every configuration. Every alternative group also has a parent option, which is always selected, if a child option is selected, too. Hence, those configuration options do not provide any added value to our analysis, which is why we excluded them for the sake of readability. For `7z`, this leaves us with 39 "important" configuration options spread over the x-axis. For instance, there are 13 728 configurations in the whole population (sample size 100%) that have the configuration option `Deflate` selected. Note, that a configuration option can be selected in more than one configuration.

Similar to the previous figures, the graph representing the whole population (black line) is the same for all six constraint solvers, as its shape is solely defined by the variability model and not influenced by any external factors like the constraint solvers. The actual sample sets (5%, 10%, 20%, and 50%) can have a different shape depending on the constraint solver. An optimal constraint solver (concerning the configuration option frequency) would compute a sample set which has a similar configuration option frequency as the whole population. For example, MSF computes quite ideal sample sets where only a few configuration options deviate from their optimal frequency.

To better visualize the actual difference, we illustrate the difference of the relative frequencies with respect to the whole population in Figure 4.6. All six constraint solver deviate from the frequency distribution of the whole population, whereas some do more than others. Comparing the average deviation from the whole population, MSF and OR-Tools seem to able to compute the best sample sets concerning the configuration option frequency.

Configuration Option Frequency — Summary

We presented the frequency distribution of the configuration options for several sample sizes for `7z`. We saw that every sample set deviates from the whole population, some more than others.

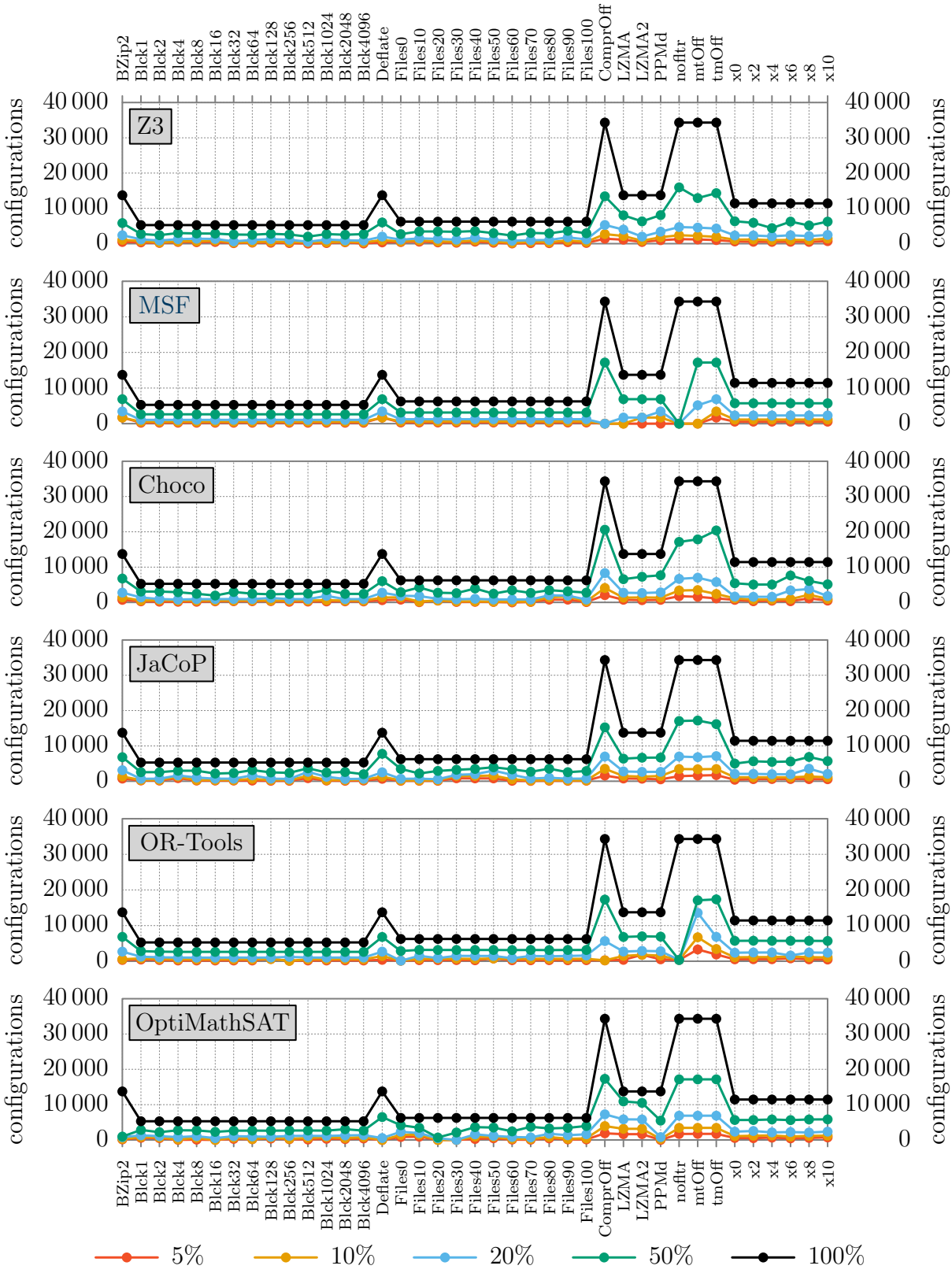


Figure 4.5: Configuration option frequency for 7z. We illustrate the average frequency across all random seeds (see Equation 3.10). Every line represents a different sample size relative to the whole population.

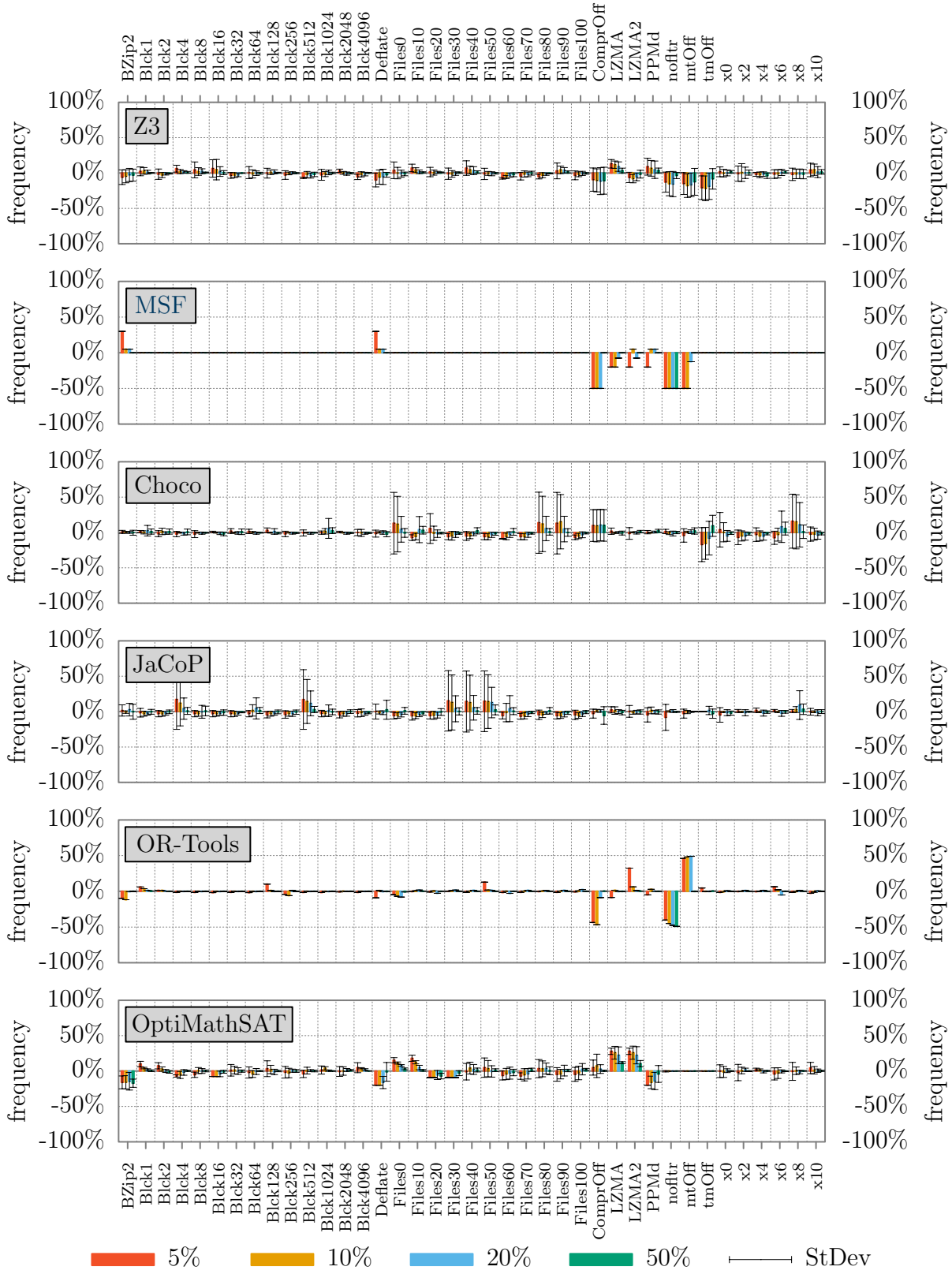
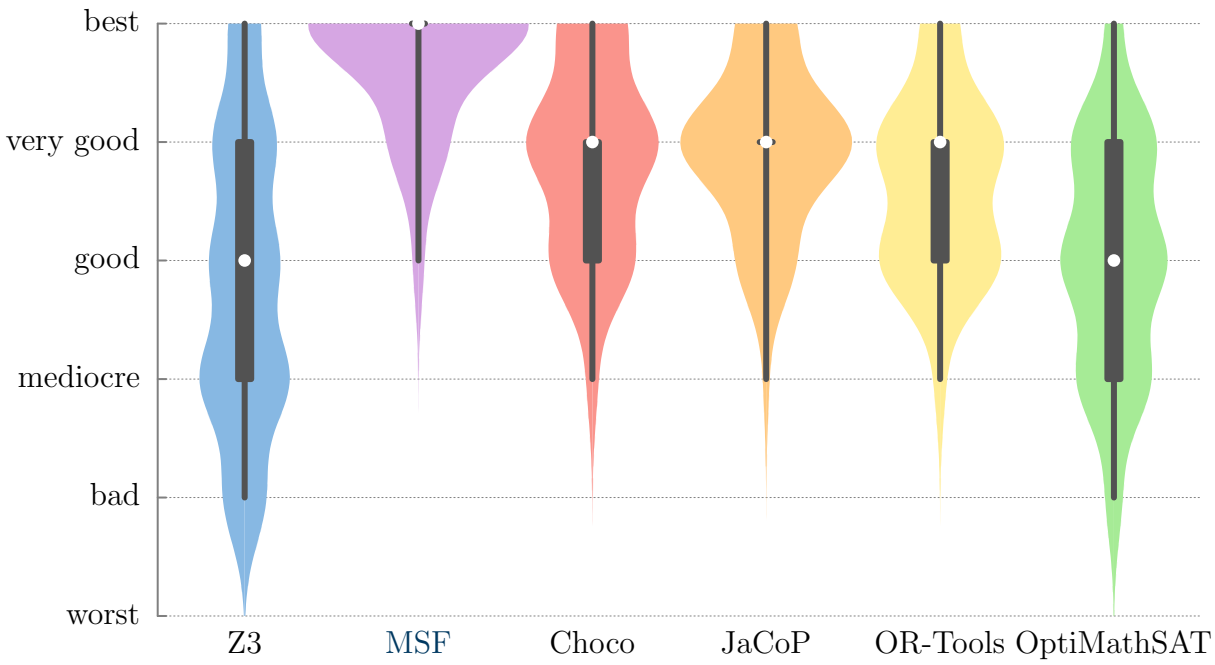


Figure 4.6: Relative frequency difference of the configuration options for 7z. Every bar represents a different sample size relative to the whole population and illustrates the average relative frequency across all random seeds (see Equation 3.11). The whiskers indicate the standard deviation caused by the random seeds (see Equation 3.13).



(a) Distribution of the rankings. By using the Mann-Whitney U test, we take all 14 subject systems, 4 sample sizes and 5 randoms seeds into account. The white dots represent the median rank for every constraint solver.

Configuration Option Frequency Ranking

1. MSF
2. JaCoP, Choco, OR-Tools
3. OptiMathSAT, Z3

(b) On average, MSF computes the best sample sets concerning the frequency of the configuration options. JaCoP, Choco, and OR-Tools deliver slightly less representative results. While OptiMathSAT and Z3 can compute good sample sets, in most cases they fail to do so.

Figure 4.7: Constraint solver ranking based on the configuration option frequency.

To summarize the results concerning the configuration option frequency, we provide a ranking for all six constraint solvers, which is based on the deviation from the whole population. For this ranking, we took all subject systems and all random seeds into account and used the Mann-Whitney U test to decide if one sample set deviates less than another one. The ranking is shown in Figure 4.7.

Z3 and OptiMathSAT cover the full range, i.e., it completely depends on the subject system, if the sample set has a similar configuration option frequency as the whole population. MSF turns out to win this ranking by far because it computes an almost perfect sample set for many subject systems.

RQ 1.3

How robust is the representativity in terms of randomness?

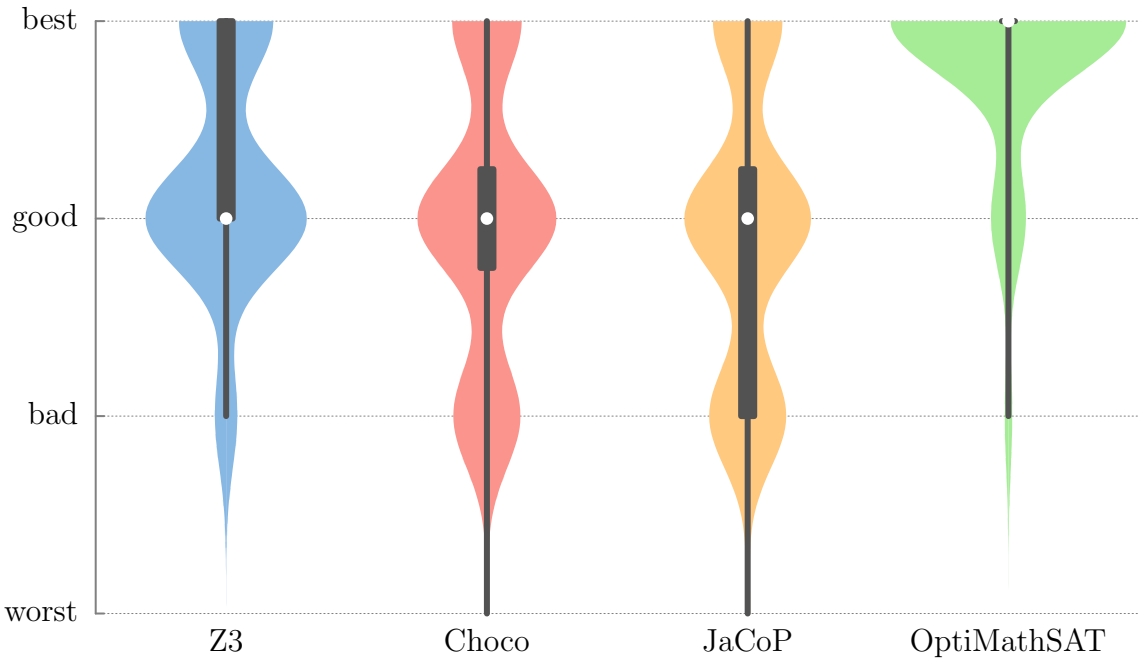
Another important part for the quality of the sample sets is the robustness of the constraint solvers against randomness. Most constraint solvers, namely Z3, Choco, JaCoP, and OptiMathSAT allow the user to set a random seed. Although OR-Tools offer a method to adjust the random seed, it does not affect the constraint solver. MSF does not provide this ability at all.

As defined in Equation 3.12 and Equation 3.13, the standard deviation of the differences can be seen in both Figure 4.3 and Figure 4.6. Every bar is associated with a whisker indicating the standard deviation when different random seed values are used. The cardinality frequencies are not as heavily influenced as the configuration option frequencies and the sample sets computed by the Java-based constraint solvers (Choco and JaCoP) vary greatly when different random seeds are used. However, the random seed has increasingly less influence when bigger sample sizes are used.

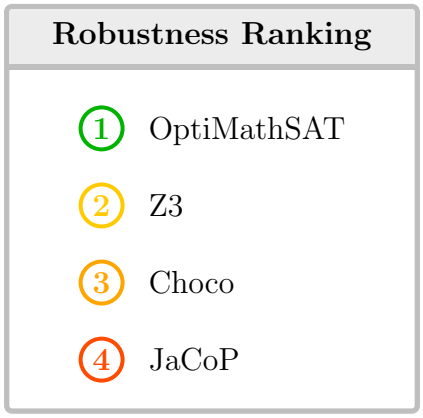
To summarize the results concerning the randomness, we provide a ranking for the four constraint solvers, which support a custom random seed. For this ranking, we took all subject systems and all random seeds into account and used the Mann-Whitney U test to decide if one sample set deviates less than another one. The ranking is shown in Figure 4.8. As mentioned above, OptiMathSAT is least influenced by the random seed, followed by Z3. The representativity of the sample sets computed by Choco and JaCoP can vary greatly, depending on the seed for the Random Number Generator.

4.2 Performance

In SPL Conqueror there are several sampling strategies, which ask for a configuration with a specific set of selected configuration options hundreds of times until some condition is met. Hence, a constraint solver needs to deliver a solution as fast as possible to be usable in practice. We explicitly refrained from using such scenarios to evaluate the performance of a constraint solver, because SPL Conqueror provides many sampling strategies, which all use the constraint solver in slightly different ways. Instead, we measure the time it takes to find all configurations in the entire



(a) Distribution of the rankings. By using the Mann-Whitney U test, we take all 14 subject systems, 4 sample sizes and 5 randoms seeds into account. The white dots represent the median rank for every constraint solver.



(b) On average, OptiMathSAT is least influenced by the randomness, followed by Z3, Choco and JaCoP. Note, that MSF and OR-Tools do not provide the ability to set a custom random seed.

Figure 4.8: Constraint solver ranking based on their robustness.

search space because this way we are independent of the actual sampling strategy. All experiments were executed on a machine with an octa-core Intel Xeon E7 with 2.4 GHz and 32 GB RAM.

RQ 2.1

How fast can a constraint solver find all configurations?

The runtime results for the compiler infrastructure LLVM can be seen in Figure 4.9. We measured two aspects:

- Initialization: before a constraint solver instance can be used, the constraint solver has to be set up by inserting all variables (i.e., configuration options) and constraints into the solver object.
- Sampling: once the constraint solver has been set up, we can use it to traverse the configuration space and find all (here: 65 536) solutions defined by the variability model.

We found, that MSF, OR-Tools, and OptiMathSAT can be used instantly because those happen to be static libraries. Static linking compiles all of the library code directly into the executable, which results in a reduced overhead from no longer having to call functions from a library and thus leads to faster load times. Choco and JaCoP — being Java-based constraint solver — need to load the appropriate Jar-files, i.e., function calls are found in shared code libraries, which have to be loaded at runtime due to the dynamic nature of Java. While Z3 is written in C++, we use the official C# bindings, which creates a small overhead during the setup phase.

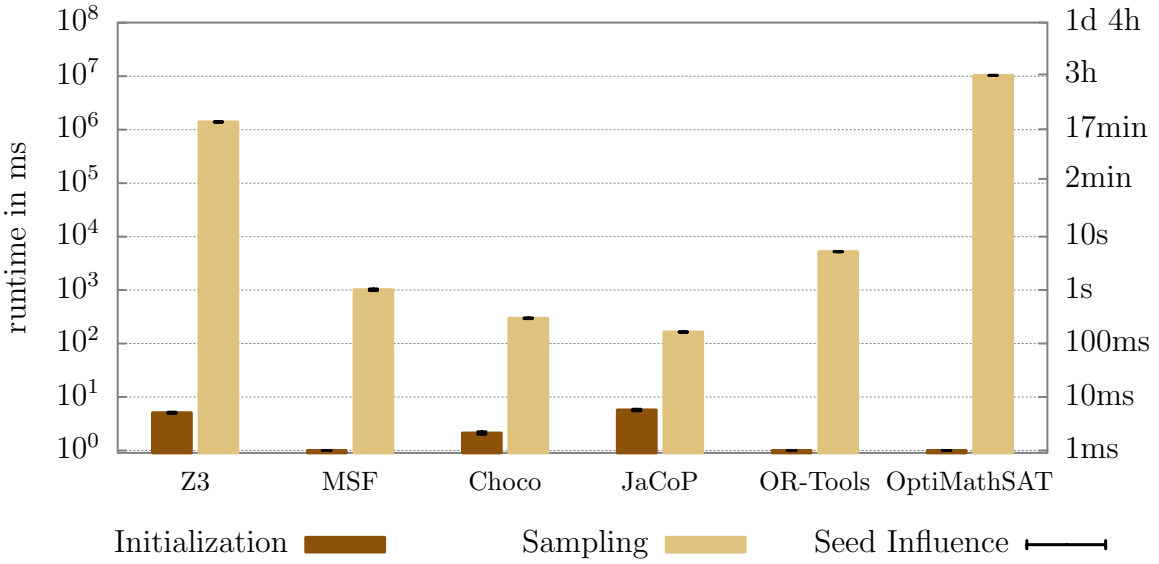


Figure 4.9: Comparision of the constraint solver performance for LLVM. We illustrate the average runtime across all random seeds (see Equation 3.16). The whiskers indicate the standard deviation caused by the random seeds (see Equation 3.17).

For the actual sample phase, JaCoP outperforms all other constraint solvers with just 164 milliseconds. Choco runs about twice as long (297 milliseconds), followed by MSF (1 second), and OR-Tools (5 seconds). Z3 (17 minutes) and OptiMathSAT (3 hours) still run a considerable amount of time longer. A similar picture can be seen for all other subject systems.

RQ 2.2

How robust is the performance of a constraint solver in terms of randomness?

We executed the experiments for five different random seeds and illustrate the standard deviation of the runtime in Figure 4.9. For all subject systems, that we used, we found that the influence of the randomness is negligible ($\approx 2\%$).

Performance — Summary

We saw, that there is a significant difference in the runtime of each constraint solver. Some operate in the range of milliseconds while others can take up to several hours to complete the same task.

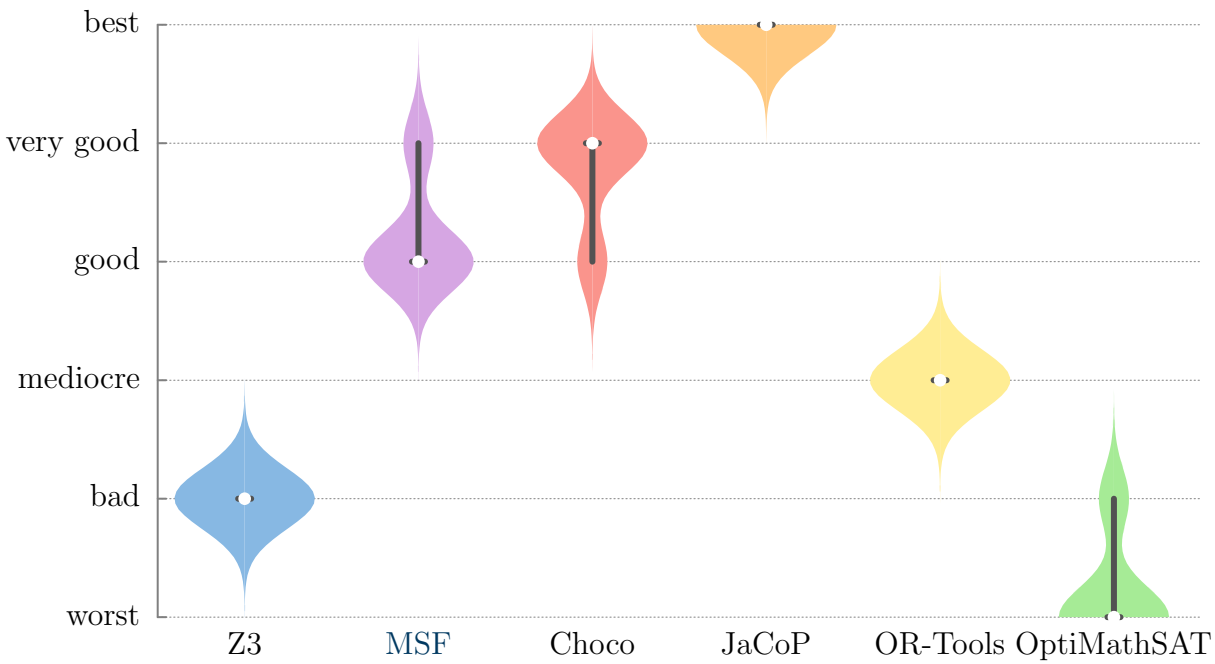
To summarize the results regarding the performance, we provide a ranking for all six constraint solvers, which is based on the time it takes to traverse the configuration space. For this ranking, we took all subject systems and all random seeds into account and used the Mann-Whitney U test to decide if one constraint solver runs faster than another one. The ranking is shown in Figure 4.10.

JaCoP is the clear winner for all subject systems, followed by Choco, MSF, and OR-Tools. On the other hand, Z3 and OptiMathSAT took the most time to complete their tasks. For bigger variability models (at least 60 000 configurations), those two regularly ran into our maximum runtime limit of ten hours.

Further Findings

In addition to the subject systems listed in Table 3.2, we challenged JaCoP further by using substantially bigger subject systems.

- The subject system DellFM has 131 configuration options and in total 1 128 674 configurations. JaCoP is still able to find all configurations in about 10 seconds.
- The subject system LargeAutomotiveFM has 18 641 configuration options and 633 631 constraints. We increased the memory limit of the JVM to 100 GB and JaCoP was able to sample up to 50 000 configurations in less than one minute before it exceeded the memory limit.



(a) Distribution of the rankings. By using the Mann-Whitney U test, we take all 14 subject systems, 4 sample sizes and 5 randoms seeds into account. The white dots represent the median rank for every constraint solver.

(b) On average, every constraint solver has its own performance characteristics. JaCoP operates in the order of hundreds of milliseconds, Choco and MSF in the order of few seconds, and OR-Tools compute the whole population within a couple of seconds. Z3 can take several minutes to traverse the entire configuration space. Finally, OptiMathSAT comes in last, as for many subject systems it can take hours to complete the search. Note, that both Z3 and OptiMathSAT were not able to sample the whole population within ten hours for variability models with more than 60 000 configurations.

Figure 4.10: Constraint solver ranking based on their performance.

4.3 Threats to Validity

In this section, we present different factors that could affect the validity of our work. We divide them into internal factors, which threaten our implementation and evaluation and external factors, which threaten the generalizability of our work.

Internal Validity

Whenever executing performance measurements of all kinds, it has to be ensured that the results do not get distorted by random fluctuation between different iterations. Therefore, we measured the runtime to find all configurations five times and took the mean value. We ran all performance-related experiments on the same cluster (see Section 4.2) to eliminate hardware influences.

Due to our selection of constraint solvers, which were written in different programming languages, we had to standardize the measuring of time. That is why, we added a warmup phase to every execution, such that language-dependent initialization, constraint solver initialization and additional loading time for dynamic libraries are not included in our measurements.

We thoroughly tested the implementation, which integrates the new constraint solvers (Choco, JaCoP, OR-Tools, and OptiMathSAT) into SPL Conqueror and compared them with the existing constraint solvers (MSF and Z3) based on their output. This allows us to minimize the risk of programming errors, which would threaten our work.

External Validity

Every constraint solver has its own set of parameters which can slightly adapt the search to special cases or change the search strategy entirely. This allows the user to tune the performance of the constraint solver. However, as Xu et al. [XJF+15] state, too many knobs do come with a cost: users encounter tremendous difficulties in knowing which parameters should be set among the large configuration space. We decided to not make use of such knobs because not all constraint solvers did offer the same set of tools. Additionally, for the CSP solvers, most parameters only make sense when numeric values are used and since we only cover binary configuration options, this is outside of the scope of this thesis.

We evaluated the constraint solvers using numerous subject systems from different domains. The different number of configuration options and constraints further increase our external validity and allows us to generalize our findings.

5. Related Work

Benavides et al. [BTRC05] describe how a variability model can be mapped onto a Constraint Satisfaction Problem. They also compare Choco and JaCoP in the automated analyses of variability models and come to the conclusion, that JaCoP is on average 54% faster than Choco in finding a solution. This is in agreement with our results, where JaCoP is on average 62% faster than Choco.

Marten [Mar18] compares a SAT solver with a CSP solver and a Binary Decision Diagram (BDD) using artificial variability models, which he creates by varying several attributes such as the number of configuration options, the feature tree depth and the number of cross-tree constraints. He finds, that the BDD approach is best suited for SAT problems and that the performance of the CSP solver and the SAT solver does not directly depend on the types of configuration options (binary/numeric), rather the number of valid configurations defined by the variability model.

Jomu George and Aït Mohamed [JGAM11] measure the effectiveness of VCS2009.06 against other commercially available constraint solvers to analyze test coverage results and adapt the test generation process to improve the coverage. They find, that VCS2009.06 is not only powerful but does also provide a simple and rich syntax to describe the problem.

Benavides et al. [BSTRC05] attach additional attributes to configuration options and use constraint programming for automated reasoning on those extended variability models. This allows for answers to questions such as how many potential products a model has or which the best product according to some criteria is.

Murashkin et al. [MAG+15] aim at finding all optimal Automotive Safety Integrity Level (ASIL) allocations using off-the-shelf constraint solvers. They implement their approach using three major classes of state-of-the-art solvers: Choco for Constraint Satisfaction Problem, Z3 for Satisfiability Modulo Theories, and CPLEX ILP Solver for Integer Linear Programming (ILP). However, in their approach, Z3 outperforms Choco. Compared to our results, this can be explained by the different version of the constraint solvers. Murashkin et al. use Z3 in version 2.0, while we use Z3 in version 4.8.1. As mentioned in Section 3.2, Z3 gained the ability to work

with optimization in version 4.4.1. This means, that Murashkin et al. use a highly optimized SAT solver, while we use an OMT solver. Additionally, they use numeric constraint in their work, in contrast to our exclusively binary variability models.

6. Conclusion and Future Work

6.1 Conclusion

Variability models are an integral part of the analysis of highly configurable software systems. They define the configuration options of a system together with numerous constraints among them. However, deriving all valid configurations (whole population) is usually infeasible for complex systems. Instead, one obtains a small, representative sample set which covers the configuration space. There exist various strategies on how to select the configurations for the sample set. However, simple random sampling is challenging, because most random samples do not satisfy the constraints, due to the highly constrained configuration spaces. More sophisticated strategies make use of a constraint solver, whose purpose is to find new valid configurations, which can then be incorporated into the individual sampling strategy. Of course, the properties of the constraint solver can heavily influence the performance of the sampling process (both runtime and quality of the outcome).

In this work, we compared six off-the-shelf constraint solvers: Z3, Microsoft Solver Foundation (MSF), Choco, JaCoP, OR-Tools, and OptiMathSAT. We integrated them into SPL Conqueror (a software suite for variability analysis) and used them to obtain sample sets from 14 variability models of different size and complexity. We analyzed those sample sets and ranked the constraint solvers based on their ability to provide representative configurations, whereby representative configurations can refer to different metrics based on the research question.

First, we focused on the overall shape of the sample set: the number of selected options in a configuration, i.e., the cardinality distribution of the sample set. We found, that MSF does not compute representative sample sets compared to the whole population, but prefers configurations with few selected configuration options. Sampling strategies which aim at projecting the cardinality distribution of all valid configurations onto the sample set should refrain from using Microsoft Solver Foundation. The best constraint solver (out of our six selected ones) for this task is OptiMathSAT. The other constraint solvers all perform equally as good in the average case, with Choco and JaCoP usually surpassing Z3 and OR-Tools.

Second, we analyzed the configurations themselves, i.e., the frequency of the configuration options in the sample set. All six constraint solvers deviate from the frequencies in the whole population — some more than others. Again, JaCoP and Choco perform very well compared to all other constraint solvers. Only MSF draws better distributions, which comes not as a surprise, as this constraint solver prefers configurations with few selected configuration options. This way, it traverses through most configuration options and creates a distribution similar to that of the whole population.

Third, we took randomness into account, since both cardinality distribution and the frequency of the configuration options can be influenced by a random seed. MSF and OR-Tools do no support a custom random seed. The other four constraint solver all compute different results when the seed value changes. We found, that OptiMathSAT and Z3 are more robust against randomness as JaCoP and Choco.

Finally, we measured the runtime of the sampling process. The time for the initialization of the constraint solver is negligible compared to the actual sampling. We found, that there is a clear ranking among the individual constraint solvers. This order is (almost) consistent over all 14 variability models and hence is independent of their size and complexity. JaCoP outperforms all other constraint solvers in all cases and is even (to a certain degree) able to handle significantly larger variability models (more than 1 million configurations). Choco — ranked second — also performs very well, but fails to obtain sample sets from the huge variability models due to memory overflow. Both Java-based constraint solvers complete the sampling process in a matter of seconds. Microsoft Solver Foundation has a similar, but slightly worse performance than Choco. For our subject systems, OR-Tools operate in the order of tens of seconds. Performance-wise, Z3 and OptiMathSAT were not able to complete all tasks that the other constraint solvers could do. Both regularly hit our maximum time limit of ten hours for bigger variability models (more than 60 000 configurations). If they completed the sampling in time, Z3 did that in the order of tens of minutes while OptiMathSAT needed several hours.

In summary, every constraint solver has different characteristics in the different aspects that we chose to evaluate them. If special abilities (e.g., representativity in the cardinalities of the configurations) are required, there are constraint solvers for those areas, but they come with the drawback of poor performance. In particular, OptiMathSAT might compute representative sample sets but does that in a time frame that is not acceptable in most scenarios. On the other hand, JaCoP and Choco performed quite well in both representativity of the sample set and the performance to compute those results. Additionally, JaCoP can handle far bigger variability models. This makes them a good starting point for every sampling strategy regardless of its needs. We recommend to replace Z3 by JaCoP as the default constraint solver in SPL Conqueror, since it performs well in all aspects.

6.2 Future Work

In our work, we only made use of variability models with binary configuration options. Some of the subject systems initially used numeric configuration options, but we converted those to use exclusively binary configuration options. This does not

limit the applicability of our experiments, because a numeric variability model can be converted to only use binary configuration options and constraints. Since the support for numeric variability models in SPL Conqueror is currently in an experimental state, we did not cover these areas.

Similarly, we did not evaluate the constraint solvers with variability models containing mixed constraints, i.e., constraints containing both binary and numeric configuration options. Once SPL Conqueror and in particular, the sampling strategies support numeric configuration options, this evaluation can be refined.

Finally, we did not take Integer Linear Programming (ILP) solvers into account, because of their fundamentally different approach using equations for constraints. This makes the conversion process (variability model to formula) more expensive but does not disqualify them for variability analysis, since they fulfill all requirements as described in Section 3.1. Based on the promising results of Murashkin et al. [MAG+15], even a performance improvement over CSP solvers may be expected.

A. Appendix

A.1 CSP Solver Listing

Table A.1: CSP solver candidates for SPL Conqueror integration.

Solver	Language	Decision	Notes
Choco 4	Java	✓	—
iZplus		✗	only available in chinese
JaCoP	Java	✓	—
OR-Tools	C++	✓	—
Picat SAT		✗	custom language
sunny-cp^{-}		✗	constraint solver combination
Yuck	Scala	✗	no library

A.2 SMT Solver Listing

Table A.2: SMT solver candidates for SPL Conqueror integration.

Constraint Solver	Language	Decision	Notes
ABsolver	C++	✗	no library
Alt-Ergo	OCaml	✗	no support for optimization
Barcelogic	C++	✗	no library
Beaver	OCaml	✗	no library
Boolector	C	✗	no support for optimization
CVC4	C++	✗	no support for optimization

Continued on next page

Continued from previous page

Constraint Solver	Language	Decision	Notes
iSAT		✗	no library
Microsoft Solver Foundation	NET	✓	—
MathSAT	C, Python, Java	✓	via OptiMathSAT extension
MiniSmt		✗	no library
Norn		✗	no library
OpenCog	C++, Python	✗	not intended for end-users
OpenSMT	C++	✗	no support for optimization
raSAT		✗	no library
SMTInterpol	Java	✗	no support for optimization
SMCHR	C	✗	no library
SMT-RAT	C++	✗	toolbox for constraint solver composing
SONOLAR	C	✗	no library
Spear		✗	no library
STP	C, C++, Python, OCaml, Java	✗	no support for optimization
SWORD		✗	no support for optimization
veriT	C/C++	✗	decent efficiency
Yices	C	✗	no support for optimization
Z3	C/C++, NET, OCaml, Python, Java	✓	—

Bibliography

[ABKS16] Sven Apel, Don Batory, Christian Kästner, and Gunter Saake. *Feature-Oriented Software Product Lines*. Springer, 2016. (cited on Page 4)

[BPF15] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. νZ — An Optimizing SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199. Springer, 2015. (cited on Page 8)

[BSTRC05] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Java CSP Solvers in the Automated Analyses of Feature Models. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 399–408. Springer, 2005. (cited on Page 1 and 33)

[BSTRC06] David Benavides, Sergio Segura, Pablo Trinidad, and Antonio Ruiz-Cortés. A First Step Towards a Framework for the Automated Analysis of Feature Models. *Managing Variability for Software Product Lines: Working With Variability Mechanisms*, pages 39–47, 2006. (cited on Page 1)

[BTRC05] David Benavides, Pablo Trinidad, and Antonio Ruiz-Cortés. Using Constraint Programming to Reason on Feature Models. In *SEKE*, pages 677–682, 2005. (cited on Page 1 and 33)

[dMB08] Leonardo de Moura and Nikolaj Bjørner. Z3: An Efficient SMT Solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008. (cited on Page 8)

[dMB11] Leonardo de Moura and Nikolaj Bjørner. Satisfiability Modulo Theories: Introduction and Applications. *Communications of the ACM*, 54(9):69–77, 2011. (cited on Page 3)

[dMDS07] Leonardo de Moura, Bruno Dutertre, and Natarajan Shankar. A Tutorial on Satisfiability Modulo Theories. In *International Conference on Computer Aided Verification*, pages 20–36. Springer, 2007. (cited on Page 3)

[FW74] Jay Fillmore and Gill Williamson. On Backtracking: A Combinatorial Description of the Algorithm. *SIAM Journal on Computing*, 3(1):41–55, 1974. (cited on Page 3)

[GCR19] Alberto Griggio, Alessandro Cimatti, and Sebastiani Roberto. MathSAT. Website, July 2019. Available online at http://mathsat.fbk.eu; visited on July 15th, 2019. (cited on Page 9)

[Goo19] Google. Operations Research Tools. Website, April 2019. Available online at https://developers.google.com/optimization; visited on April 2th, 2019. (cited on Page 9)

[HNRW19] Matthias Heizmann, Aina Niemetz, Giles Reger, and Tjark Weber. International Satisfiability Modulo Theories Competition 2018. Website, March 2019. Available online at http://smtcomp.sourceforge.net/2018/index.shtml; visited on March 18th, 2019. (cited on Page 8)

[JGAM11] Mani Paret Jomu George and Otmane Aït Mohamed. Performance Analysis of Constraint Solvers for Coverage Directed Test Generation. In *ICM 2011 Proceeding*, pages 1–5. IEEE, 2011. (cited on Page 33)

[KGS+19] Christian Kaltenecker, Alexander Grebhahn, Norbert Siegmund, Jianmei Guo, and Sven Apel. Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering*, pages 1084–1094. IEEE Press, 2019. (cited on Page 1 and 5)

[KS19] Krzysztof Kuchcinski and Radoslaw Szymanek. JaCoP Solver. Website, March 2019. Available online at https://osolpro.atlassian.net/wiki/spaces/JACOP/overview; visited on March 11th, 2019. (cited on Page 9)

[LVRK+13] Jörg Liebig, Alexander Von Rhein, Christian Kästner, Sven Apel, Jens Dörre, and Christian Lengauer. Scalable Analysis of Variable Software. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 81–91. ACM, 2013. (cited on Page 5)

[MAG+15] Alexandr Murashkin, Luis Silva Azevedo, Jianmei Guo, Edward Zulkoski, Jia Hui Liang, Krzysztof Czarnecki, and David Parker. Automated Decomposition and Allocation of Automotive Safety Integrity Levels Using Exact Solvers. *SAE International Journal of Passenger Cars-Electronic and Electrical Systems*, 8(2015-01-0156):70–78, 2015. (cited on Page 33 and 37)

[Mar18] Adrian Marten. A Comparison Study of Domain Constraint Solver for Model Counting. *Master's Thesis, University of Passau*, 2018. (cited on Page 33)

[Mic19] Microsoft. Microsoft Solver Foundation. Website, July 2019. Available online at https://www.nuget.org/packages/Microsoft.Solver.Foundation; visited on July 10th, 2019. (cited on Page 8)

[MW47] Henry Mann and Donald Whitney. On A Test Of Whether One Of Two Random Variables Is Stochastically Larger Than The Other. *The Annals of Mathematical Statistics*, pages 50–60, 1947. (cited on Page 14)

[MZ09] Sharad Malik and Lintao Zhang. Boolean Satisfiability From Theoretical Hardness to Practical Success. *Communications of the ACM*, 52(8):76–82, 2009. (cited on Page 3)

[PFL19] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. Choco Solver. Website, March 2019. Available online at http://www.choco-solver.org; visited on March 26th, 2019. (cited on Page 9)

[Res19] Microsoft Research. Z3 Theorem Prover. Website, July 2019. Available online at https://github.com/Z3Prover/z3; visited on July 10th, 2019. (cited on Page 8)

[SGAK15] Norbert Siegmund, Alexander Grebhahn, Sven Apel, and Christian Kästner. Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 284–294. ACM, 2015. (cited on Page 4 and 5)

[ST19] Roberto Sebastiani and Patrick Trentin. OptiMathSAT. Website, March 2019. Available online at http://optimathsat.disi.unitn.it/index.html; visited on March 25th, 2019. (cited on Page 9)

[TS19] Guido Tack and Peter J. Stuckey. MiniZinc Challenge 2018. Website, March 2019. Available online at https://www.minizinc.org/challenge2018/results2018.html; visited on March 11th, 2019. (cited on Page 8)

[VK86] Marc Vilain and Henry Kautz. Constraint Propagation Algorithms for Temporal Reasoning. In *Proceedings of the National Conference on Artificial Intelligence*, pages 377–382, 1986. (cited on Page 3)

[XJF+15] Tianyin Xu, Long Jin, Xuepeng Fan, Yuanyuan Zhou, Shankar Pasupathy, and Rukma Talwadker. Hey, You Have Given Me Too Many Knobs! In *Proceedings of the Joint Meeting on Foundations of Software Engineering*, pages 307–319. ACM, 2015. (cited on Page 32)

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, den 20. September 2019