Saarland University

Faculty of Mathematics and Computer Science

Master's Thesis

Grammar-Based Sampling

submitted by

Kallistos Weis

submitted on

September 11, 2020

Advisor:

Christian Kaltenecker, M.Sc.

Supervisor

Prof. Dr. Sven Apel

Reviewers

Prof. Dr. Sven Apel

Prof. Dr. Andreas Zeller

UNIVERSITÄT
DES
SAARLANDES

Erklärung

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

Statement

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken,______________________ ______________________________
(Datum/Date) (Unterschrift/Signature)

Abstract

Recent software systems tend to be highly configurable. Configurations of these systems are combinations of configuration options, e.g., on-off switches for runtime or compile-time options that might enable or disable certain functionality of the program. These configurations have not only an impact to the functionality of the software but also to its performance. Performance in this context has many different meanings, e.g., execution time or power consumption. Finding the best performing configuration in a brute-force manner by measuring all configurations is almost always infeasible. State-of-the-art solutions create a sample set (i.e., a subset of all valid configurations), measure the sampled configurations, and apply machine-learning algorithms to predict the performance of configurations in the configuration space. Creating a valuable sample set for machine-learning algorithms present many challenges.

In this thesis we present a new sampling strategy that is lightweight in terms of memory footprint and execution time but selects the configurations uniformly over the valid configuration space. To evaluate the quality of our sampling strategy, we compare the prediction results to other state-of-the-art sampling strategies, e.g., t-wise sampling, on 10 variability models of real-world software systems. For the prediction results we use the most promising three different ML algorithms, i.e., random forest, multiple linear regression, and support vector machines, to learn on our sample sets. In a second step we investigate the performance of our approach regarding execution time. In our results we show also the connection between the valid configuration space and the grammar configuration space.

Acknowledgements

To begin with, I would like to express my sincere gratitude to Professor Sven Apel to supervising this research work and providing thoughtful comments and recommendations on this master's thesis. I would like to thank Professor Andreas Zeller for reviewing it.

Furthermore I'd like to express my very great appreciation to my advisor Christian Kaltenecker for his patient guidance, enthusiastic encouragement and useful critiques of this research work. His willingness to give his time so generously has been very much appreciated.

I would also like to extend my thanks to the members of the current Chair of Software Engineering for their helping suggestions and offering me a pleasant and productive working environment.

It is whole-heartedly appreciated that Niklas Metzger, Ricardo Lang and Valentin Seimetz always took the time for discussions and assistance regarding this research work as well as proofreading this thesis.

Finally, I wish to thank everyone else who supported and encouraged me throughout my study.

Contents

List of Figures

1. Introduction

Modern software development evolves to create highly configurable software systems. The flexibility of the software system is used to adapt the software product to different hardware and the needs of the user. Moreover, it is easier to deploy bug fixes to a whole range of products, as the developer needs to fix it only once in the software system and not in each software product. Accordingly, the software developer has to introduce configuration options to the software system to enable and disable parts of the software system to configure the software product.

As the software system evolves, it acquires more and more configuration options that also share dependencies among themselves. We call the combination of enabled and disabled configuration options a configuration. Moreover, configurations that meet all dependencies among the configuration options are called valid configurations. All other combinations of configuration options are called invalid configurations. Henceforward, we call the set of all valid configurations of a software system the configuration space of the software system. Variability models are used to describe the configuration space of a software system [2].

Since different valid configurations enable and disable functionality of the software system, they also lead to a different performance of the software system. In particular, configuration options influence the performance, i.e., execution time [21], of the software product, by modifying the functionality of the software system. To argue about the performance of a software product, we need a performance model of the configuration space, which means we need a performance number for every valid configuration. However, since the size of the configuration space potentially grows exponentially in the number of configuration options, it is not feasible to measure the performance for every configuration. Instead, machine-learning algorithms are used to learn on a small subset of a configuration space, i.e., a sample set, and generalize to performance prediction models for the whole configuration space.

In this thesis, we present an approach to create sample sets of configuration spaces. This approach does not require any kind of insight into the code or structure of the software system, i.e., it is a black-box sampling approach. Instead it requires knowledge of the variability model and access to the software system. As a consequence of the black-box approach, we are interested in producing a sample set that is uniformly spread across the configuration space. In our approach we sample truly random, without enumerating all valid configurations but by radically reducing the exponential space of possible configurations. To reduce the exponential space of possible configurations we abuse the structure of the variability model to build a context-free grammar. The language of the context-free grammar describes a superset of the configuration space. Furthermore, we define a bijective function to convert an integer to a word in the language of the grammar, i.e., a configuration. However, we need to verify the produced samples to be valid configurations, as our reduction is an over approximation of valid configurations.

The unique selling point of our approach is its performance and sample set quality. On the one hand, our sampling strategy keeps up with the performance of state-of-the-art approaches, e.g., t-wise [16][10], or even outperforms them, e.g., randomized solver-based sampling [9]. On the other hand, the quality of sample sets is estimated by the accuracy of the prediction model that is built with that sampling set. As a consequence of the structure of our approach, we reach the same quality as the baseline, i.e., random sampling from the whole population.

2. Background

In this chapter, we explain basic knowledge on context-free grammars, SPL Conqueror and three machine-learning algorithms that we use throughout this thesis. In our approach, we make use of the concept of context-free grammars to cast variability models. To evaluate our approach we make use of SPL Conqueror and three machine-learning algorithms. Therefore, we explain shortly what SPL Conqueror is and give an intuition for the used machine-learning algorithms.

2.1 Context-free grammar

Context-free grammars are used to describe context-free languages. We use the structure of variability models to cast a context-free grammar that describes an over approximation of the configuration space as its context-free language. We over approximate the configuration space of a variability model, since not all cross-tree constraints are representable in context-free grammars without an exponential growth. Accordingly, the words in our language represent configurations in the configuration space, if they are valid configurations.

A context-free grammar is a 4-tuple $\mathcal{G} = (\Sigma_n, \Sigma_t, S, \delta)$, where Σ_n is the set of non-terminal symbols, Σ_t is the set of terminal symbols, S is the start symbol and δ is the set of productions. A production is a recursive rule that allows to rewrite a non-terminal to a string of non-terminal and terminal symbols. In this thesis we use the terminal symbol ϵ to symbolize the empty string. The root of the variability model is always the start symbol of our context-free grammar.

To conclude this section, we give the context-free grammar $\mathcal{G}_e$ of the variability model depicted in Figure 2.1:

- $\Sigma_t = \{\text{txt, png, jpeg, encryption, bzip2, LZ77, RLE, ppm}\}$
- $\Sigma_n = \{\langle\text{root}\rangle, \langle\text{compression}\rangle, \langle\text{fileformat}\rangle, \langle\text{encryption}\rangle\}$
- $S = \langle\text{root}\rangle$.

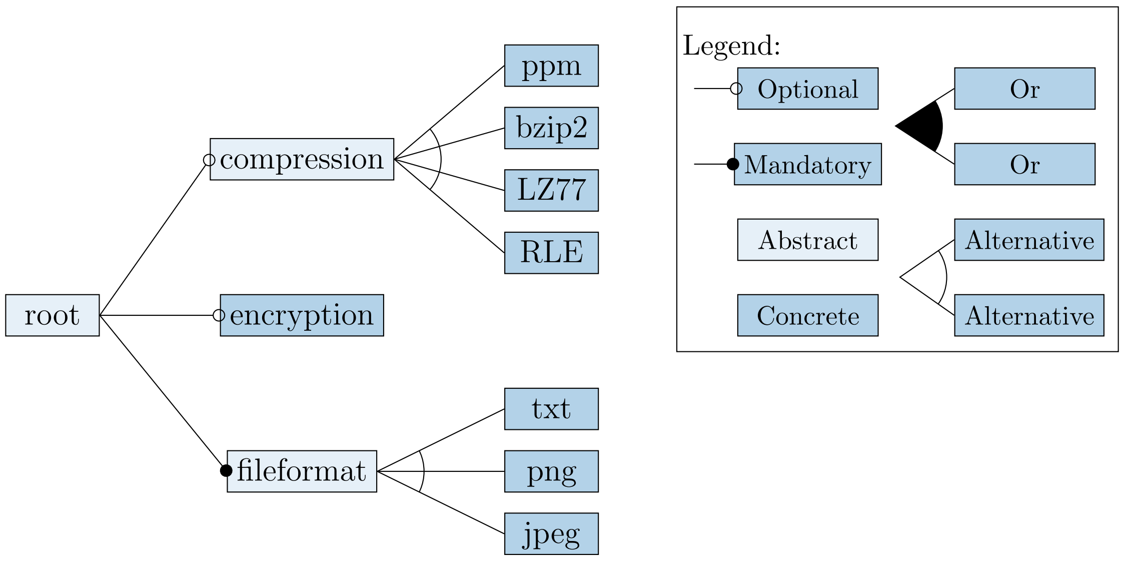


Figure 2.1: Example variability model

where we enclosed all non-terminal symbols with ⟨⟩, to ensure that each symbol in the grammar is unique. We built the set of productions δ as follows:

$$\begin{aligned}
\langle\text{root}\rangle &\models \langle\text{compression}\rangle\,\langle\text{encryption}\rangle\,\langle\text{fileformat}\rangle\\
\langle\text{compression}\rangle &\models \epsilon \mid \texttt{ppm} \mid \texttt{bzip2} \mid \texttt{LZ77} \mid \texttt{RLE}\\
\langle\text{encryption}\rangle &\models \epsilon \mid \texttt{encryption}\\
\langle\text{fileformat}\rangle &\models \texttt{txt} \mid \texttt{png} \mid \texttt{jpeg}
\end{aligned}$$

An example word from this grammar, i.e., *⟨root⟩ ⟨compression⟩ ⟨encryption⟩ ⟨fileformat⟩ ppm encryption jpeg*, represents the valid configuration *root, compression, encryption, fileformat, ppm, jpeg* where we removed all enclosing ⟨⟩ and resulting duplicates to create the valid configuration.

2.2 SPL Conqueror

SPL Conqueror [22] is a tool-suite to execute performance predictions for configurations of software systems. In particular, SPL Conqueror is able to produce sample sets of valid configurations for a variability model of a software system. To generate sample sets, SPL Conqueror provides different sampling strategies, e.g., t-wise [16][10], diversified-distance-based [11], or randomized-solver-based sampling [9]. Additionally, it provides an interface to parse existing sampling sets of arbitrary sampling strategies.

Also, SPL Conqueror uses sample sets to learn performance prediction models with different machine-learning algorithms. SPL Conqueror implements the multiple linear regression algorithm and supports python machine learning algorithms, like support vector machines, forest regression, or decision trees.

2.3 Machine-Learning

In this section, we give a short introduction of the three machine-learning algorithms that we use to evaluate our approach.

2.3.1 Multiple Linear Regression

Multiple linear regression (MLR), also known as multiple regression, is a statistical technique to generate a model with several variables that serve to predict the value of a response variable. The outcome model of MLR can be written in the following form:

$$y = X\beta + \epsilon \tag{2.1}$$

where y and X are input to the MLR, whereas y is the vector of response variables and X is the matrix with all predictor variables. In our evaluation y is the performance value of the software system, in particular we have an entry in y for all n configurations in the sample set. Accordingly, each row of X represents wether a configuration option was enabled during the performance measurement of the corresponding y_i. As it is not always possible to produce a fitting β such that $\epsilon = 0$, ϵ describes the residuals. MLR aims to find the best fitting β to minimize the sum of residuals $\sum_{i=1}^{n} e_i$.

2.3.2 Random Forests

Random Forests [12] make use of multiple classification trees, it constructs them to get different classifications for one input vector. Each classification tree runs independently on the input vector, classifies the input vector and votes for its classification. The forest acts as a supervisor and will select the classification with the most votes.

2.3.3 Support Vector Machines

Support Vector Machines (SVM) [5] are supervised learning models that classify input data into different categories. Additionally, SVMs are able to interpret the input data as a high-dimensional feature space and perform a non-linear classification on them. As a consequence, SVMs represent the examples in a high-dimensional space and divide each separate category by a gap that is as wide as possible. The resulting model maps an input to the same space and predicts the belonging category based on the position in the space.

3. Related Work

In this chapter, we discuss different sampling strategies related to this thesis. We differentiate between two different classes of sampling strategies, i.e., random-based and solver-based.

3.1 Random-based approaches

One way of producing random samples is by generating a random number between zero and the number of valid configurations. This number is then used as index of the configuration. That means we have to enumerate all of the valid configurations, which is exponential in time.

Another approach is to model the individual configuration option as bits and encode the configuration by a bit string. In this bit string, a bit is set to one if the corresponding configuration option is enabled and set to zero otherwise. Now, we can produce random samples by getting a random number in the range from one to N — where N is the number of valid configurations. Lastly, we have to check if the produced sample is actually valid. This step is important since most of the possible configuration option combinations are not valid [9].

Counting Binary Decision Diagrams (CBDD) are used to count all valid configurations in a sophisticated manner [18]. Creating a CBDD and mapping a number to a configuration via traversing the CBDD is straightforward [19]. But the creation of a BDD may take some time and the memory consumption of it is huge if the formula of the variability model is large [1, 17].

In Figure 3.1 we see an example for two CBDDs with 6 variables. This example illustrates that the size of the CBDD depends heavily on the ordering of the variables. In the worst case, the size of the CBDD is exponential in the size of the variables. Additionally, the indices of the nodes in Figure 3.1 depict how we calculate the number of solutions in the CBDD. We can count the solutions in a CBDD by traversing the CBDD in a bottom-up manner, and add up the number of solutions of the children.

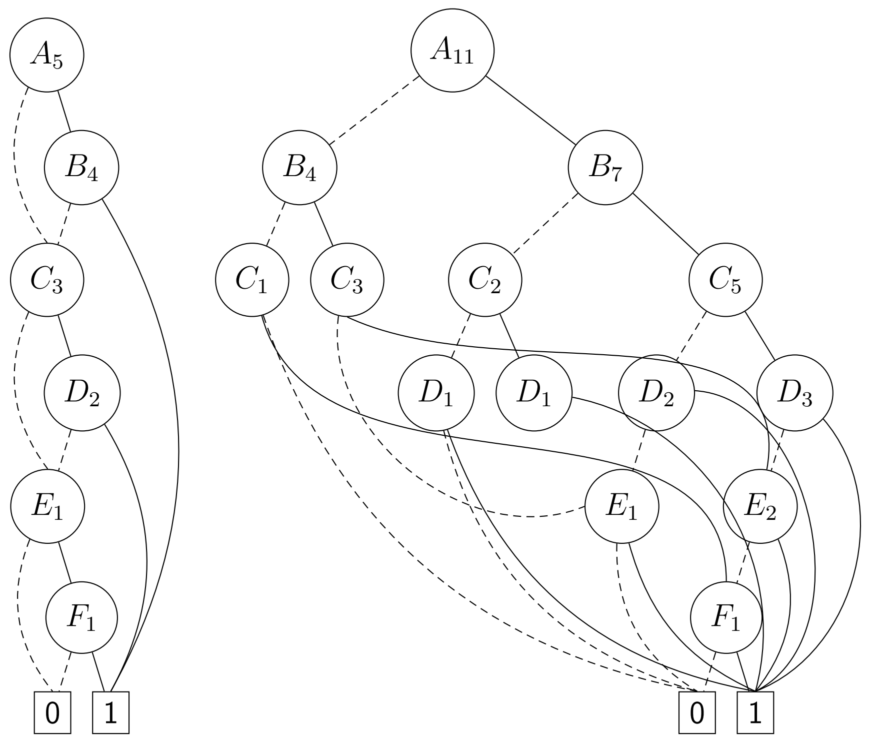


Figure 3.1: Example of variable ordering dependency to the BDD size, c.f. [6]. The left CBDD depicts the function $A \cdot B + C \cdot D + E \cdot F$, the right CBDD depicts the function $A \cdot D + B \cdot E + C \cdot F$. Note that the indices are not part of the variable names but represent the count of solutions for the CBDD.

Shubham et al. [20] introduced another random-based sampling approach. They produce uniform sample sets by transforming the variability model, given as CNF-formula, to a d-DNNF representation. The d-DNNF representation is then used to generate sample sets with any size by making two passes over it. The first pass annotates the representation with the set of variables for the sub-formula as well as the model count of the respective sub-formula. In the second pass, they produce all samples at once, which means, that they produce for each sample the set of choices for all sub-formulae. These sets of choices are then converted to the individual samples.

In contrast to the approach from Shubham et al. [20], where they use the CNF formula to create a d-DNNF representation by ignoring the hierarchical structure induced by the variability model, we focus on taking the additional information into account by creating a grammar out of the variability model.

3.2 Solver-based approaches

In this section we discuss algorithms that make use of SMT- or SAT-solvers to produce samplings, whereas SAT is a subproblem of SMT [3]. Unlike the random-based approaches this makes use of highly optimized software, e.g., Z3 [8]. The usage of that software ensures on one hand that all configurations obtained are valid. On the other hand, the solver is fast despite the theoretical bounds of the SAT decision problem.

SMT-solvers, like Z3, provide an interface to get solutions to a given SAT-formula, e.g., variability model. Each solution then represents a valid configuration of the variability model. Computing a sample set is relatively fast and uses little memory but the solver tends to produce solutions that are similar to already found solutions. That leads to low quality performance models generated by such sample sets [11]. To scatter the solutions Henard et al. [9] shuffle all clauses in the formula after a solution was found. Additionally, they change the order of all configuration options within the clauses. This forces the SAT-solver to rebuild its entire model from scratch after each found sample, which leads to a huge time increment. In the remainder of this thesis we call this approach “randomized solver-based”. The quality of the resulting performance model is better — but still worse than from a random sample set.

To further improve the quality of the performance model Kaltenecker et al. [11] introduced an approach that encodes additional constraints to the formula. These additional constraints ensure that a solution has a certain distance to a reference point. Kaltenecker et al. [11] use the Manhattan distance of a configuration to the origin of the configuration space, i.e., the configuration where all configuration options are disabled. Figure 3.2 shows the configuration space of a variability model with three configuration options (A, B and C) and no constraints between them. Each corner of that cube describes one valid configuration and is labeled with a bit string. In the bit string each bit represents a configuration option and is set to 1 if the feature is activated and 0 otherwise, e.g., 001 means that features A and B are not selected and feature C is selected.

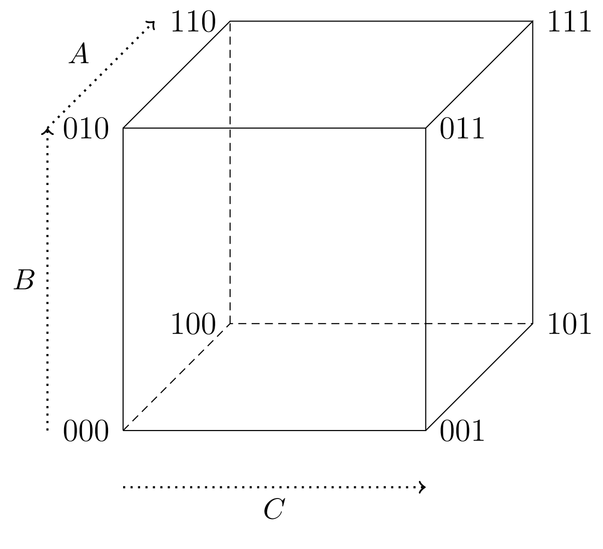


Figure 3.2: Example of Manhattan distance for 3 configuration options A, B and C with no constraints, c.f. [11]

The aim of the algorithm is that the distance distribution in the sample set is uniform. Therefore the distances for each individual sample is randomly chosen. Figure 3.3 shows the distance distribution of the example from Figure 3.2. Here we depict that if we sample randomly over the distance space we are not sampling uniformly over the configuration space as there are not equally many configurations per distance.

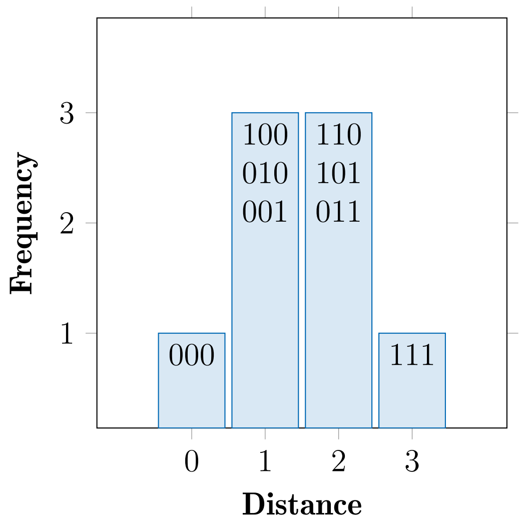


Figure 3.3: Distance distribution of configurations from Figure 3.2, c.f. [11]

In their evaluation, Kaltenecker et al. [11] show that unfortunately some features do not appear at all or only rarely in the sample set. They show that the performance model is worse than the performance model from a coverage-based approach.

A coverage-based approach, e.g., t-wise sampling [16][10], ensures that the sample set reaches a high coverage on certain criteria. Such criteria are based on configuration options, e.g. each configuration option has to appear at least in one configuration. These criteria are added as additional constraints to the SAT-formula, and then the SAT-solver finds a valid configuration based on the criteria. t-wise uses a parameter, t, to modify the criterion where t determines the size of the tuples. If $t = 1$ each single configuration option has to appear in at least one configuration inside the sample set. For $t = 2$ each pair of configuration options, provided that this pair appears in a valid configuration, has to appear in at least one configuration inside the sample set. Therefore, some interactions and influences from larger tuples of configuration options, i.e., tuples larger than t, are neglected. As a consequence of that, the prediction model is worse than the baseline [11], i.e., true random sampling.

As these last two approaches seem to complement each other, Kaltenecker et al. [11] adapted their algorithm. They added one additional constraint to each sample ensuring that the least frequently used configuration option in the sample set has to be included in the next sample if possible for the selected distance. That leads to a performance model which is only slightly worse than the baseline [11].

4. Grammar-based Sampling

In this chapter, we explain how grammar-based sampling works. Before we explain the sampling itself, we present the algorithm that is used to produce a context-free grammar from the variability model. We use the context-free grammar to narrow down the set of possible configurations to a small superset of the configuration space. This algorithm is sketched and explained in Section 4.1. Furthermore, we explain the limits we set for our context-free grammar. In Section 4.2, we explain how we generate a grammar-based sample-set and how we deal with the limitations of the context-free grammar.

4.1 Context-free Grammar

In this section, we sketch the main algorithm to generate a context-free grammar from a given variability model. We split the algorithm into two parts. The first part shows the starting point of the algorithm that has to traverse the variability model. The second part deals with the recursive traversal of the variability model.

Algorithm 1: Algorithm to generate a grammar from a variability model

```
  Input: vm
  Output: grammar
1 Function GenerateGrammar(vm):
2     root = getRoot(vm)
3     addStartSymbol(grammar, root)
4     if hasChildren(root) then
5         addNonterminal(grammar, root)
6         GenerateGrammarRecursively(root, grammar)
7     else
8         addTerminal(grammar, root)
9     return g
```

Algorithm 1 shows the starting point of the conversion of a variability model (*vm*) to a context-free grammar (*grammar*).

First, we set the root of the variability model as our start symbol S of the context-free grammar, c.f. Chapter 2. In addition, each configuration option in the variability model is either a non-terminal of the context-free grammar, if it is an inner node, or a terminal of the context-free grammar, if it is a leave node. After that, we use Algorithm 2 if the variability model consists of more than just the root configuration option; otherwise we are done.

Since the variability model has a tree-like structure, we build the context-free grammar recursively. In Algorithm 2, we see two major differentiations.

Algorithm 2: Recursive part of the algorithm to generate a grammar from a variability model

```
Input: current, grammar
Function GenerateGrammarRecursively(current, grammar):
    children = getChildren(current)
    if isOptional(current) then
        addRule(grammar, current, $\epsilon_{current}$)
    if haveChildren(children) then
        if areAlternativeGroup(children) then
            foreach child in children do
                addNonterminal(grammar, child)
                addRule(grammar, current, child)
                GenerateGrammarRecursively(child, grammar)
        else
            addRule(grammar, current, children)
            foreach child in children do
                addNonterminal(grammar, child)
                GenerateGrammarRecursively(child, grammar)
    else
        if areAlternativeGroup(children) then
            foreach child in children do
                addTerminal(grammar, child)
                addRule(grammar, current, child)
        else
            addRule(grammar, current, children)
            foreach child in children do
                addNonterminal(grammar, child)
                addRule(grammar, child, child)
                addRule(grammar, child, $\epsilon_{child}$)
                addTerminal(grammar, child)
```

First, we check if the node is optional, (Line 3), and add an ϵ if this is the case. Then, we differentiate between leaf and inner nodes, (Line 5 and 16). In the case that we are at an inner node, we have to call Algorithm 2 for every child, while we build our grammar, (Line 10 and 15). In the other case we just build the remainder of the grammar. The second differentiation takes care of alternative groups, (Line 6 and 17). An alternative group means that we can choose one of many options, whereas an or group gives us the possibility to choose some out of many options. The main difference in building the context-free grammar for these two kinds of edges between a node and its children is that in the case of an alternative group we have an extra rule for each child, (Lines 8 – 10 and 19 – 20). While, in the case of an or group we only have a single rule, containing all children, (Lines 12 – 15 and 22 – 27). As a consequence, we can not prevent that no option is selected, as we have to allow each option to be disabled. Thereby, we add a configuration to the language that is not valid in the variability model. Lastly, the function *addRule* ensures that a rule contains either terminals or non-terminals. If we add a Terminal to a rule that contains, or will contain, a non-terminal, we introduce a placeholder non-terminal and add a rule for the placeholder with the terminal.

To conclude this section, we explain how we calculate the number of words in the context-free grammar.

Algorithm 3: Algorithm to calculate the number of configurations in the language of the grammar

```
   Input: name
   Output: number
1  Function CalculateNumberConfigurations(name):
2      rule = getRule(name)
3      if isTerminal(rule) then
4          return numberSymbols(rule)
5      if isOrRule(rule) then
6          number = 0
7      else
8          number = 1
9      foreach symbol in rule do
10         if isOrRule(rule) then
11             number = number + CalculateNumberConfigurations(symbol)
12         else
13             number = number · CalculateNumberConfigurations(symbol)
14     return number
```

First of all, we have two important invariant characteristics of the generated context-free grammar. The first characteristic is that a rule either contains terminals or non-terminals. This characteristic arises during the generation of the context-free grammar. The second characteristic is induced by the structure of the variability model. We know that all configuration options, that share the same parent, either exclude each other, or can be arbitrarily combined.

To calculate the number of words in the grammar, we propose the algorithm `CalculateNumberConfigurations` in Algorithm 3. Henceforward, we call rules that allow the activation of exactly one configuration option *OrRules* and rules that activate all configuration options in it *AndRules*. Configuration options that are optional in *AndRules* can be disabled if the ϵ-symbol is chosen as its terminal.

The initial input to call Algorithm 3 is the root configuration option. In particular, we traverse the grammar in a depth-first search manner and accumulate the number of configurations bottom-up. In each recursive call we consider two main cases. On the one hand, we return the number of terminals in the rule if the rule contains only terminals, (Line 4). On the other hand, we build the sum over the returned numbers for each rule of the non-terminals in *OrRules* and the product over the returned numbers for each rule of the non-terminals in *AndRules* recursively if the rule contains only non-terminals, (Line 11 and 13). Therefore, we initialize *number* with the neutral element of addition or multiplication respectively, (Line 6 and 8).

4.2 Generating Samples

In the last section, we introduced how we build a context-free grammar from the variability model. Now, we use this grammar to obtain a sample set of valid configurations. First, we define what *grammar-valid configurations* are. Second, we propose a function to convert integers to configurations. After that, we define how we obtain samples for the sample set and proof that the sample set is uniformly distributed in the set of all valid configurations.

To begin with, we define the term *grammar-valid configurations*. It is vital to bear in mind that we are able to embed all structural constraints from the variability model in the context-free grammar, c.f. Section 2.1, besides the cross-tree constraints. In other words, the language $\mathcal{L}(\mathcal{G})$, that is described by the generated context-free grammar $\mathcal{G}$, contains exactly the valid configurations from the variability model, if there are no cross-tree constraints. Additionally, if the variability model contains cross-tree constraints, $\mathcal{L}(\mathcal{G})$ contains all valid configurations and also some invalid configurations.

Definition 4.1 (Grammar-valid configurations)**.** *Let C_o be the set of all configuration options. Let V be the set of all valid configurations. Then 2^{C_o} is the set of all possible configurations and $V \subseteq 2^{C_o}$. We reduce the set of all possible configurations to the set of all possible configurations that are part of $\mathcal{L}(\mathcal{G})$.*

The resulting set G is the set of all grammar-valid configurations and $V \subseteq G \subseteq 2^{C_o}$.

After defining what *grammar-valid configurations* are, we introduce an algorithm that implements a function `NumberToConfiguration` $: [0, N[\to G$ that converts a number to the corresponding grammar-valid configuration. The input range of `NumberToConfiguration` depends on the number of *grammar-valid configurations*, thus $N =$ `CalculateNumberConfigurations(start)`, where *start* is the start symbol of the grammar. In Algorithm 4, we describe algorithm `NumberToConfiguration`, that implements the function `NumberToConfiguration` $: [0, N[\to G$.

Algorithm 4: Algorithm to convert an integer to a configuration

```
   Input: number, symbol
   Output: configuration
1  Function NumberToConfiguration(number, symbol):
2      rule = getRule(symbol)
3      if isTerminal(rule) then
4          return List(getSymbol(rule, number))
5      if isOrRule(rule) then
6          foreach sym in rule do
7              num = CalculateNumberConfigurations(sym)
8              if number $\geq$ num then
9                  number = number $-$ num
10             else
11                 return NumberToConfiguration(number, sym).append(sym)
12     else
13         numberSymbols = length(rule)
14         configuration = List()
15         for i in range(0, numberSymbols) do
16             basis = 1
17             for j in range(i, numberSymbols) do
18                 sym = getSymbol(rule, j)
19                 basis = basis $\cdot$ CalculateNumberConfigurations(sym)
20             index = number / basis
21             number = number % basis
22             sym = getSymol(rule, i)
23             configuration.append(sym)
24             configuration.concat(NumberToConfiguration(index, sym))
25         return configuration
```

To convert a number in the range from $0 - N$, we traverse the grammar starting from the start-symbol, in a depth-first manner. The result of Algorithm 4 is a list of chosen symbols from the grammar where each symbol depicts a selected configuration option. In the beginning of Algorithm 4, we test if the rule consists of terminals, and return the matching terminal. In the case that the rule consists of non-terminals, we differentiate between *OrRules* and *AndRules*, as we did in Algorithm 3. If we consider an *OrRule*, we have to choose the matching non-terminal. In particular, the *number* tells us the index of the matching configuration, and each non-terminal represents an interval of possible configurations. To choose the correct configuration we identify the non-terminal with the matching interval. Then, we fit *number* to the selected interval and call Algorithm 4 recursively with the new *number* and the corresponding non-terminal *sym*. Afterwards, we return the result concatenated with *sym*, (Lines 6 – 11).

On the other hand, if we consider an *AndRule*, we add all non-terminals to the configuration. In addition, we identify for every non-terminal the *number*, that represents the index of the configuration in the corresponding rule, (Lines 15 – 22). As before, we call Algorithm 4 recursively, with the identified *number*s and corresponding non-terminals, and concatenate the results to our configuration, (Line 24).

As a consequence of Algorithm 4, we can now produce a random number r in the range from 0 to N and use the function f to convert r to a grammar-valid configuration s_t. In Definition 4.2 we define s_t as a temporary sample.

Definition 4.2 (Temporary Sample)**.** *Let $N = |G|$ and r be a random number from $[0, N[$. Then s_t is a temporary random sample with $s_t =$* `NumberToConfiguration`*(r).*

Unfortunately, we can not add a temporary sample to our sample set, as we ignore cross-tree constraints in our context-free grammar. For this reason, we define the term *sample* and how to obtain a sample from a temporary sample.

Definition 4.3 (Sample)**.** *Let $\mathtt{SAT} : C \rightarrow \{\top, \bot\}$ be a function that returns whether a configuration c is a valid configuration or not. Then s_t is considered a sample iff $\mathtt{SAT}(s_t) \equiv \top$.*

Definition 4.3 demands that the temporary sample s_t is a valid configuration to become a sample. We show the algorithm to create a grammar-based sample set in Algorithm 5, where n is the number of samples to generate, and S is the sample set.

Algorithm 5: Algorithm to create a sample set with n configurations

```
  Input: n
  Output: S
1 Function GenerateSampleSet($n$):
2     S = Set()
3     while $|S| < n$ do
4         $r$ = random$(0, |G|)$
5         $c$ = NumberToConfiguration(r)
6         if SAT($c$) then
7             $S = S \cup \{c\}$
8     return S
```

Depending on Algorithm 5, we have to make sure that we will always find valid configurations. Corollary 4.1 shows that we will eventually get a valid configuration.

Corollary 4.1. *Let $M = |V|$ be the number of valid configurations and $N = |G|$ the number of grammar-valid configurations. Then the following equation describes how the repetition of Definition 4.3 applies:*

$$P_V(g) = \lim_{n \to \infty} \sum_{k=0}^{n} \left(\frac{N-M}{N} \right)^k \frac{M}{N} = 1$$

To conclude this chapter, we show that our grammar-based sample set is uniformly distributed inside the set of valid configurations.

Proposition 4.1 (Uniform Grammar-Based Sample Set)**.**
Let $A'(v)$ be the event that a valid configuration v is added to the grammar-based sample set. For all samples s in the grammar-based sample set holds: $P(A'(s)) = \frac{1}{|V|}$. Therefore, the grammar-based sample set is uniformly distributed within the set of all valid configurations.

Proof. To proof the uniformity of the produced sampled set, we need to proof that the probability of adding a valid configuration to the sample set is equal for every valid configuration.

Let $A(v)$ be the event that configuration v is drawn. Let $B(v)$ be the event that the drawn configuration v is valid. Then, $P(A'(v)) = P(A(v)|B(v))$ as we only add a configuration v to our sample set if it is a valid configuration, (Algorithm 5, Line 5 and 6).

There are two cases to consider:

First, v is an invalid configuration, that gives $P(A(v)) = \frac{1}{|G|-|V|}$,$P(B(v)) = 0$ and $P(A'(v)) = P(A(v)|B(v)) = 0$.

Second, v is a valid configuration. Then, we know $P(A(v)) = \frac{1}{|V|}$, $P(B(v)) = 1$ and $P(A'(v)) = P(A(v)|B(v)) = \frac{1}{|V|}$.

In short, we only add valid configurations to our sample set and we choose a new sample randomly from the whole grammar-valid configuration space. Accordingly, we have only the second case to consider. Therefore, our sampling strategy produces an uniform sample set. $\square$

5. Methodology

In this chapter, we give an overview of our research questions, the real-world software systems that are used to evaluate our approach as well as the operationalization of our evaluation.

5.1 Research Questions

Our research questions, which build the foundation of our evaluation, will be further discussed in Chapter 6, but can be summarized as following. During our evaluation we investigate different characteristics of our sampling strategy and compare the results with other sampling strategies, i.e., random sampling [11], solver-based sampling [9], randomized solver-based sampling [9], t-wise sampling [16], and diversified distance-based sampling [11]. For each research question we explain our motivation behind and also go into details about it.

RQ$_1$: How many invalid configurations are drawn by the grammar-based sampling strategy?

In Definition 4.1, we define grammar-valid configurations and state how the sets of valid configurations, and grammar-valid configurations are related. In the first research question, we explore how often a SAT-solver rejects configurations that are selected by the grammar-based sampling strategy during the creation of sample sets.

RQ$_{2.1}$: What is the influence of the sampling strategy on the uniformness of the sample set?

Variability models describe the way in which software systems are configurable. Unfortunately, they do not provide additional information about configuration options, e.g., if they are performance critical. As a consequence of that, machine-learning algorithms learn the performance models on sample sets for these software systems without information about specific configuration options. In particular, we cannot make assumptions on the level of configuration options, to improve the sample set.

Therefore, we are interested in sample sets consisting of samples that are uniformly distributed over the set of valid configurations.

To answer this research question, we investigate the uniformness of the sample set produced by our sampling strategy and compare it with sample sets produced by different state-of-the-art sampling strategies.

RQ$_{2.2}$: What is the execution time of the different sampling strategies?

Besides the uniformness of the sample set, we are interested in the execution time of our sampling strategy. In this research question we want to compare the execution time to create sample sets with our sampling strategy to the execution time of other sampling strategies.

RQ$_{3}$: What influence has the chosen sampling strategy on the accuracy of performance prediction?

We use machine-learning algorithms to produce a prediction model for the performance of a software system. A sample set represents a subset of all configurations, which are used as input for the machine-learning algorithm. In our context we do not have any more information than the variability model of the software system and as such we are in a black box environment. In this black box environment, we do not know which feature has the most influence on the performance of the software system and therefore we have to uniformly sample over the whole configuration space.

Our grammar-based sampling strategy produces a sample set using a context-free grammar, c.f., Chapter 4, to convert random numbers to configurations, that are added to the sample set. This research question aims to compare the accuracy of the performance prediction model generated with grammar-based sample sets in contrast to performance prediction models created with other sample sets from other sampling strategies.

5.2 Variability Models

In this section, we describe 10 real-world software systems whose variability models we used in the evaluation of this thesis. Most of these software systems are well known and range from small to huge software systems whereas the variability models of these systems range only from small to medium, in terms of configuration options. The variability models as well as all measurement data can be found on github[1]. The hardware and experiment setup used for the measurements are listed at the respective paragraph below. We also give a short overview of each software system, including what the software system is about and how up to date it is.

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

7-zip

7-zip is a file archiver with a high compression ratio written in C++. It is a free open-source software, first released in 1999, and mainly developed by Igor Pavlov.

In our measurement we used version 9.20. Our variability model of 7-zipconsists of 44 configuration options and a total of 68 640 valid configurations. The only performance metric we were interested in was the time 7-zip needs to compress its workload. As our workload we used 60 times the canterbury corpus and 12 times the large corpus[2].

Our time measurements were done on an Intel Xeon E5–2690 CPU with 64GB of RAM; the operating system was Ubuntu 16.04.

BerkeleyDB

BerkeleyDB is an embedded database system. We considered the C interface of BerkeleyDB as our software system. It is a software library to provide a high-performance embedded database for key-value data. The first release was in 1994 and the latest stable release is from 2018 by the Oracle Corporation.

We used version 4.4.20 of BerkeleyDB for our measurements. Our variability model of BerkeleyDB consists of 18 configuration options and a total of 2 560 valid configurations. The performance metric we were interested in was the response time of the database for read and write queries.

Our time measurements were performed on an Intel Core 2 Quad Cpu at 2.66GHz with 4GB of RAM; the operating system was Windows Vista.

Dune

Dune is a modular C++ library which uses grid-based methods to solve partial differential equations. Its latest release was in 2020 and the development started in 2002 on the initiative of Prof. Bastian, Dr. Ohlberger, Prof. Rumpf [4].

We used version 2.2 of Dune for our measurements. Our variability model of Dune consists of 12 configuration options and a total of 2 304 configurations. The performance metric we were interested in was the time to solve partial differential equations, i.e., Poisson's equations, a workload that is provided in Dune.

Our time measurements were performed on an Intel i5–4570 CPU with 32GB of RAM; the operating system was Ubuntu 13.04.

Hipacc

Hipacc is a domain specific language and compiler that allows the user to design image processing kernels and algorithms in it. It provides a framework to run these high-level descriptions on low-level hardware. To compile these descriptions to different hardware they make use of LLVM.

[2]http://corpus.canterbury.ac.nz

Our variability model of Hipacc consists of 54 configuration options and a total of 13 485 valid configurations. The performance metric we were interested in was the time Hipacc needs to solve partial differential equations.

Our time measurements were done on a Nvidia Tesla K20 with 2496 Cores and 5GB of RAM; the operating system was Ubuntu 14.04.

Java Garbage Collection

The Java Garbage Collection is a program within the Java virtual machine that manages memory automatically. At runtime it routinely detects and reclaims unused memory.

We used the Java garbage collection from Java 8. Our variability model consists of 39 configuration options and a total of 193 536 valid configurations. The performance metric we were interested in was the execution time of the garbage collection on the benchmark and the workload for the measurements was the xalan benchmark from the DaCapo benchmark suite[3].

Our time measurements were done on an Intel Xeon E5–2690 CPU with 64GB RAM available; the operating system was Ubuntu 14.04.

LLVM

LLVM[4] is a compiler infrastructure that provides the tools to easily setup and extend compilers. It consists of modular compiler and optimization toolkit technologies that are developed by a huge and increasing community. The project was started around the year 2003 by Chris Lattner and Vikram Adve at the University of Illinois[14].

We used version 2.7 of LLVM for our measurements. Our variability model consists of 11 configuration options and a total of 1 024 valid configurations. The performance metric we were interested in was the compilation time and the workload was the clang frontend from the opt-tool benchmark.

Our time measurements were performed on an AMD Athlon64 Dual Core CPU with 2GB of RAM; the operating system was a Debian GNU/Linux 6.

lrzip — Long Range ZIP

lrzip is a compression tool optimized to compress large files. It provides the ability to choose between two advantages, one is extreme compression and the other is an extremely fast compression.

We used version 0.600 of lrzip for our measurements. Our variability model consists of 19 configuration options and a total of 432 valid configurations. The performance metric we were interested in was the compression time of uiq^8h generated files with a size of 632MB.

Our time measurements were done on an AMD Athlon64 Dual Core with 2GB of RAM; the operating system was Debian GNU/Linux 6.

[3]http://dacapobench.sourceforge.net
[4]https://llvm.org

Polly

Polly[5] is a LLVM framework for high-level loop and data-locality optimizations. Its name comes from the mathematical model of integer polyhedra that are used to analyze and optimize memory accesses. As for today, these optimizations are mostly classical loop transformations, but there is also work done to automate GPU code generation[6].

We used version 3.9 of Polly alongside LLVM version 4.0.0 and Clang version 4.0.0. Our variability model of Polly consists of 40 configuration options and a total of 60 000 valid configurations. The performance metric we were interested in was the run time of Polly to analyze and optimize the workload “gemm” from the polybench, a benchmark suite.

Our time measurements were done on an Intel Xeon E5–2690 CPU with 64GB of RAM; the operating system was Ubuntu 14.04.

vpxenc

vpxenc is a video encoding software that encodes to VP8 and VP9 encoding. We focused on the VP9 encoding part of the software.

Our variability model consists of 42 configuration options and a total of 216 000 valid configurations. The performance metric we were interested in was the time to encode the first two seconds from the Big Bug Bunny trailer.

Our time measurements were done on an Intel Xeon E5–2690 CPU with 64GB of RAM; the operating system was Ubuntu 14.04.

x264

x264 is an open-source software library[7] to encode video streams.

Our variability model consists of 16 configuration options and a total of 1 152 valid configurations. The performance metric we were interested in was the encoding time needed to encode the Sintel trailer, that is 734MB large.

Our time measurements were performed on an Intel Core Q6600 CPU with 4GB of RAM; the operating system was Ubuntu 14.04.

5.3 Experimental Dependencies

In this section, we define the empirical variables we consider in our experiments. We start with the independent variables, that arise from the research questions we already explained. Afterwards, we continue with the resulting dependent variables. To conclude this section, we explain the confounding variables and our counter measurements to mitigate them.

[5]https://polly.llvm.org
[6]http://polly.llvm.org/documentation/gpgpucodegen.html
[7]https://www.videolan.org

Independent variables	RQ_1	$RQ_{2.1}$	$RQ_{2.2}$	RQ_3
Random Seed		✓	✓	✓
Variability Model	✓	✓	✓	✓
Number of Grammar-Valid Configurations	✓			
Number of Valid Configurations	✓			
Machine-Learning Algorithm			✓	
Dependent variables	RQ_1	$RQ_{2.1}$	$RQ_{2.2}$	RQ_3
Number of Rejected Configurations	✓			
Uniformness		✓		
Execution Time			✓	
Error Rate				✓

Figure 5.1: Independent and dependent variables and their relation to our research questions

5.3.1 Independent Variables

Random Seed

In Chapter 2, we provide a rough overview of the state-of-the-art sampling strategies to which we compare our grammar-based strategy. Almost all of them, as well as our grammar-based strategy rely on randomness. Consequently, the answers to all of our research questions are influenced by using different random seeds.

To minimize the influence of a single random seed, we used for each experiment in our evaluation 100 different random seeds from 1 to 100.

Variability Model

Variability models are designed to describe the configuration space of software systems. Therefore, they vary in the number of configuration options, or-groups, alternative-groups, and cross-tree constraints. Additionally, they also vary in the size of children per node, and, as a consequence of this, they also vary in the depth of the tree that results from the variability model.

All these differences influence the results in the evaluation of our research questions. To take these influences into account, we use 10 different variability models, which are described in Section 5.2, to answer RQ_1, $RQ_{2.1}$, $RQ_{2.2}$, and RQ_3.

Number of Grammar-valid Configurations

We show in Section 4.1 how to build a context-free grammar of a variability model, and propose an algorithm to count the number of configurations in the language described by this grammar.

Number of Valid Configurations

We have two options to get the number of all valid configurations, the first option is to calculate the number of solutions for the variability model using #SAT. The second option is to use SAT-solver to enumerate all valid configurations. Since we already have enumerated all valid configurations, to calculate the error rate of all valid configurations, we use the second option to count all valid configurations.

Machine-Learning Algorithm

In RQ$_3$, we target the prediction error of the performance prediction model created by machine-learning algorithms using our sampling strategy. Since there are different machine-learning algorithms that create performance prediction models based on sample sets, we have to take the influence of the machine-learning algorithm on the prediction error in our evaluation into account. Therefore, we use 3 different machine-learning algorithms, which are briefly introduced in Section 2.3.

5.3.2 Dependent Variables

Number of Rejected Configurations

During the sampling, we store all sampled configurations that are rejected by the SAT-solver. The number of rejected configurations, that where sampled by the grammar-based strategy, depends on the ratio of grammar-valid configurations on valid-configurations, which is influenced by cross-tree constraints, as they are not depicted in the grammar.

Uniformness

For a good and diverse sample set, we aim at creating sample sets that are uniformly distributed inside the configuration space. Besides the theoretical proof that our sampling strategy samples uniform, a statistical test for uniformness, i.e., Chi-Squared test, helps us to answer RQ$_{2.1}$.

Execution Time

In our approach, we exploit structural constraints on configuration options in the variability model to reduce the space of possible configurations drastically. Accordingly, the execution time of our sampling approach depends on the presence of structural constraints in the variability model. In the absence of structural constraints, our approach performs like an random approach, as we cannot refine the space of possible configurations with cross-tree constraints.

Therefore, we measure the execution time of our sampling strategy on real-world variability models to evaluate the performance and answer RQ$_{2.2}$.

Error Rate

The accuracy of deterministic prediction models correlates with the mean error rate over all predictions of the model. Thus, we calculate the error rate for the predictions of all configurations from a software system and take the mean value over all error rates. We use the mean error rate as a dependent variable as it correlates directly with the accuracy of the prediction model to answer RQ$_3$.

5.3.3 Confounding Factors

Since we measure execution times of sampling strategies, we encounter hardware and software-specific confounding factors. These factors are interrupts and context switches in between program executions. To mitigate them, we used a minimal Linux OS, i.e., Debian 10, and slurm workload manager[8]. We used the slurm workload manager during the execution time measurements to ensures that each core only executes one program at a time to reduce context switches. Two other confounding factors for time measurements are scaling CPU frequency as well as timer interrupts. However, we repeated every time measurement 5 times to get the mean execution time and refrained from further mitigating these factors since they have only a negligible effect on our performance measurements.

5.4 Operationalization

In this section, we depict our experimental setup and formalize the methods that we use in Chapter 6 to answer our research questions. To evaluate our sampling strategy we use state-of-the-art machine learning algorithms to create performance prediction models of the software systems.

Experimental Setup

To evaluate our grammar-based sampling approach, we implemented it as a part of SPL Conqueror. That gives us the possibility to directly compare the prediction models that are produced by the different sampling strategies that are implemented inside SPL Conqueror. Additionally, we can compare execution times for the individual sampling strategies, as they are implemented in the same tool suite. We use this time comparison to evaluate if our sampling strategy is able to keep up with other state-of-the-art sampling strategies.

We used 100 random seeds, i.e., `seed` $\in [1, 100]$, for all 10 variability models. Furthermore, we produced for each `seed` three sample sets with different sizes. In particular, we used the sizes for the sample sets that t-wise sampling with $t \in \{1, 2, 3\}$ produces. Since all other sampling strategies can produce sample sets with arbitrary many members, we guarantee this way that all sample sets are of equal size.

All time measurements on the sampling strategies are performed on a cluster of twenty nodes. Each node contains 2 Intel(R) Xeon(R) CPU E5–2630 v4 @2.20GHz and 256GB of RAM. We ensured that on these nodes our time measurements had a minimal operating system running, i.e., Debian 10 with a version 4.19 Linux kernel and only necessary software installed to minimize background noise during the measurements.

Sampling Error

To discuss RQ_1 we define an error metric to measure the number of rejected samples per sample. We output all samples that were rejected by a SAT-solver. These data

[8]https://slurm.schedmd.com/documentation.html

are used to count the number of rejected samples for each case study over 3 sample set sizes and 100 random seeds each.

We weigh the number of rejected samples, we further denote this number as "rejected", to the number of samples to draw, we further denote this number as "samples", as follows:

$$\texttt{error} = \frac{\texttt{rejected}}{\texttt{samples}}$$

where we refer to the `error` as the sampling error of a case study.

Sampling Uniformity

To answer $\mathrm{RQ}_{2.1}$ we use the Kolmogorov Smirnov and Cramer-von Mises test [7] for goodness of fit on our sample sets to test whether they are uniform or not. Both tests check the hypothesis that a sample set, given as input, resembles a given distribution, i.e., a uniform distribution. The hypothesis is to reject if the returned p-value is less than a threshold, i.e., $p < 0.05$.

Sampling Time

To answer $\mathrm{RQ}_{2.2}$ we need to define a time metric that we can use to compare the execution time of our sampling strategy to the performance of other sampling strategies. To do that we measure the time SPL Conqueror needs to produce a sample set with a given sampling strategy and a specific size. Moreover, we include the time to set up the sampling strategy, i.e., to build the context-free grammar in the case of the grammar-based sampling.

We measure the time, by using the function "time", a sampling strategy ($\texttt{sample}_s(k)$) needs to produce a sample set with size k for a specific random seed ($s \in S$) as follows:

$$\texttt{K-STime}_s = \texttt{time}(\texttt{sample}_s(k))$$

We use `K-STime` time to compare the performance of the different sampling strategies on different sizes of variability models and different sizes of sampling sets.

Since our sampling strategy depends on a random seed, as well as most other sampling strategies we used to compare against, we define the mean k-sampling time over different random seeds ($s \in S$):

$$\overline{\texttt{K-STime}} = \frac{\sum_{s \in S} \texttt{K-STime}_s}{|S|}$$

Prediction Error

To discuss RQ_3 we use the performance prediction models that we created with different sampling sets to predict the performance for each valid configuration of the corresponding software system. We weigh for each configuration ($c \in C$) the distance of the predicted performance ($\texttt{predicted}_c$) to the measured performance ($\texttt{measured}_c$) as follows:

$$\texttt{error}_c = \frac{|\texttt{predicted}_c - \texttt{measured}_c|}{\texttt{measured}_c}$$

where we refer to the $\texttt{error}_c$ as the error rate of the performance prediction model regarding the configuration c. To rate the quality of the performance prediction model, we define the mean error rate over the whole population:

$$\overline{\texttt{error}} = \frac{\sum_{c \in C} \texttt{error}_c}{|C|}$$

To further discuss RQ_3 we enlarge our error rate definitions by the variance of error rates from different performance prediction models, as follows:

$$\widetilde{\texttt{error}} = \texttt{Var}(\{\texttt{error}_c | c \in C\})$$

It is to note that, a smaller error rate, a smaller mean error rate, as well as a smaller variance of the error rate is to prefer. To answer RQ_3, we use a Kruskal-Wallis test [13] to test for every sample size on every case study if the error rates of at least two sampling strategies differ significantly ($p < 0.05$). We then perform pair-wise and one-sided Mann-Whitney U tests [15] to identify on every case study and sample set size, if and which sampling strategy leads to the lowest error rate, that differs significantly from all others. Additionally, we determine the effect size using the $\hat{A}_{12}$ measure by Vargha and Delaney [23]. The values of $\hat{A}_{12}$ indicate small, medium and large effect sizes, if the are larger than 0.56, 0.64, and 0.71, respectively.

6. Evaluation

In this chapter, we evaluate our grammar-based sampling strategy by answering the research questions from Section 5.1. In each section, we present the results for the respective research question.

6.1 Invalid Drawn Configurations

To start our evaluation, we want to answer our first research question: RQ_1: How many invalid configurations are drawn by the grammar-based sampling strategy?

Observation

In RQ_1, we investigate how many invalid configurations are sampled and rejected by a SAT-solver during the creation of sample sets by the grammar-based strategy. To better understand the numbers of invalid drawn configurations, we first provide the ratio between grammar-valid and valid configurations for each case study.

In Figure 6.1, we see in the first column the name of the case study — all of them are explained in Section 5.2. Additionally, we listed the number of configuration options (CO) in the second column. The third and fourth columns state the number of valid configurations (V), and grammar-valid configurations (G) respectively, for the software system in column 1. The second to last column contains the ratio between valid (V) and grammar-valid (G) configurations. In the last column we list the sampling error, c.f., Section 5.4.

Before we talk about the results itself, it is important to note that a ratio below 1 means that there are less valid configurations than grammar-valid configurations. The closer the ratio to zero the more grammar-valid configurations are invalid configurations. If the ratio equals 1, we know that all grammar-valid configurations are valid. More importantly, the ratio cannot be greater than 1, as this would mean there are valid configurations that are not grammar-valid, which is contradicted by the construction of the grammar. In contrast, the closer the sampling error to zero, the less false samples were drawn.

Case Study	(CO)	(V)	(G)	(V) vs (G)	SE
7-zip	44	68 640	68 640	1	0
BerkeleyDB	18	2 560	2 560	1	0
Dune	32	2 304	2 352	0.98	0.026
Hipacc	54	13 485	28 560	0.47	1.338
Java GC	39	193 536	193 536	1	0
LLVM	11	1 024	1 024	1	0
lrzip	19	432	432	1	0
Polly	40	60 000	60 000	1	0
vpxenc	42	216 000	216 000	1	0
x264	16	1 152	1 152	1	0

Figure 6.1: Overview of all used case studies with the number of configuration options (CO), the number of valid configurations (V), the number of grammar-valid configurations (G) and the ratio between number of valid configurations and grammar-valid configurations (V) vs (G), as well as the sampling error (SE)

There are two main observations: The number of grammar-valid configurations is equal to the number of valid configurations for all but two case studies, i.e., for the case studies Dune and Hipacc the set of grammar-valid configurations is a superset of the set of valid configurations. Since all configuration options in our case studies are binary options, the number of possible configurations is $2^{|CO|}$, which means, that the number of grammar-valid configurations is much smaller as the number of possible configurations for all case studies.

We notice, that all case studies where the number of grammar-valid configurations equals the number of valid configurations have a sampling error of zero.

Discussion

From the 8 case studies, where the number of grammar-valid configurations equals the number of valid configurations and the sampling error equals zero, we conclude that our grammar-based approach is able to cast hierarchical structures of the variability model well into a context-free grammar. It is worth to note, that all 8 case studies do not contain cross-tree constraints, i.e., constraints between nodes that do not share their parent. Both other case studies, i.e., Dune and Hipacc, contain cross-tree constraints. We notice that our approach cannot cast a context-free grammar from the structure of cross-tree constraints. For the case study with the largest over-approximation of valid configurations, i.e., Hipacc, we see that the space of grammar-valid configurations is about twice the size of the space of valid configurations. In contrast, the size of the grammar-valid configuration space is way smaller than the size of all possible configurations in that case, i.e., 28 560 grammar-valid configurations vs. 2^{54} possible configurations. Over-approximations of the valid configuration space by the grammar-valid configurations are caused by cross-tree constraints. For this reason, the over-approximation is worse for more restrictive cross-tree constraints, e.g., in Dune we have less restrictive cross-tree constraints as in Hipacc which means the over-approximation in the Dune case study is closer to the valid configurations as in the Hipacc case study.

Both case studies, where we have a sampling error unequal to zero, we encounter a correlation between the sampling error and the ratio between valid configurations and grammar-valid configurations. That means, we can decrease the sampling error by closing the gap between valid configurations and grammar-valid configurations.

We conclude that our approach produces a grammar where the set of grammar-valid configurations is drastically reduced in contrast to the set of possible configurations. The sampling error to produce a sample set is strongly correlated to the ratio between valid configurations and grammar-valid configurations. In the presence of cross-tree constraints we are not able to cast a context-free grammar that meets all constraints from the variability model, which increases the sampling error. On the contrary, without cross-tree constraints we are able to cast a context-free grammar that represents the same configuration space as the variability model, which means that the sampling error equals zero.

6.2 Sample Set Uniformness

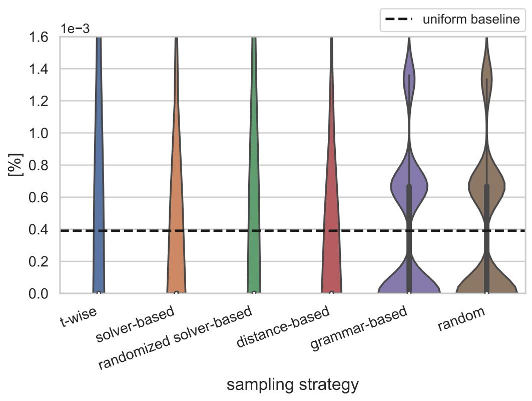
In this section, we evaluate the uniformness of the created sample sets from t-wise, solver-based, randomized solver-based, distance-based, grammar-based, and random sampling. We first depict the probability distribution of configurations for sample sets of different sizes. Then, we perform statistical tests to evaluate if the sample sets can be considered uniform, we use these tests to answer $\mathrm{RQ}_{2.1}$: How many invalid configurations are drawn by the grammar-based sampling strategy?

Observation

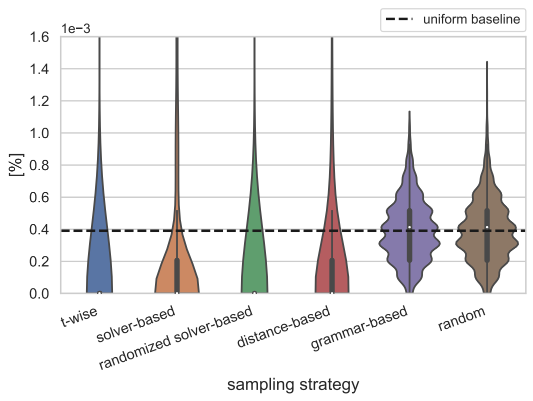
In Figure 6.2, we show the probability distribution of all configurations to be drawn as a sample in a sample set on three different sizes and across 100 random seeds each. We show the probability distribution of all configurations from the BerkeleyDB case study to appear in a sample set created by the different sampling strategies. We split the plot into three subplots, parted by the size of the underlying sample sets. The sample set sizes are determined by t-wise sampling for $t = 1$, $t = 2$, $t = 3$ and are pictured in Figure 6.2a, Figure 6.2b, and Figure 6.2c, respectively.

We see in all three plots that in the sample sets from t-wise, solver-based, randomized solver-based, and distance-based sampling a lot of configurations are not present, i.e., have a probability of 0. For grammar-based and random sampling, which share a quite similar distribution, this is only the case for the sample sets of $t = 1$.How many invalid configurations are drawn by the grammar-based sampling strategy? Note that the characteristics of violin plots lead to negative data points in Figure 6.2 although there are no negative probabilities in our results. This behaviour occurs due to the estimation functions which fit the violin around our data, including many zeros for all configurations that are not present in any sample set. We added in all plots a dashed line to indicate where the expected probability to draw a configuration lies, i.e., $\frac{1}{|V|}$ where $|V|$ stands for the number of valid configurations.

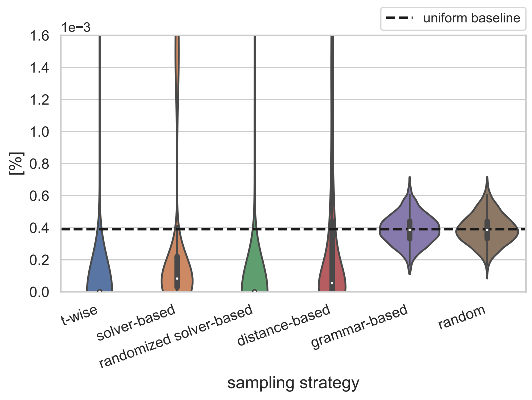
In Figure 6.3, we provide the results of the Kolmogorov Smirnov test. The columns represent the different sampling strategies, with 3 different sample set sizes, and the rows represent the case-studies. Each entry in the table represents the number of



(a) sample set size t=1



(b) sample set size t=2



(c) sample set size t=3

Figure 6.2: Probability distribution over configurations to appear in a sample set from different sampling strategies over 100 random seeds and 3 different sample set sizes of the BerkeleyDB case study

sample sets where the uniformness hypothesis is not rejected by the Kolmogorov Smirnov test ($p >= 0.05$). In our case the best result is 100, since we sampled for each case study, with each sample size, and every sampling strategy on 100 seeds. Note that the t-wise sampling strategy does not rely on random seeds, thus, they produce for a sample size always the same sample set, which means that we get either 100 or 0 in our table for t-wise sampling. Since we set random sampling as our baseline, we try to reach the same or better results with our sampling strategies as the random sampling provides. Therefore, we colored all **numbers** in green that are higher or equal than the number of positive results from the Kolmogorov Smirnov test on the random sample sets.

Discussion

In Figure 6.2, we see that grammar-based sampling behaves similar to random sampling. Most of the configurations have the same probability to appear in a sample set, with rising number of samples, whereas on low number of samples we see that a large number of configurations is not drawn at all and most of the others share the same probability. On all other sampling strategies this trend is, at best, much

	t-wise			solver-based			randomized solver-based			distance-based			grammar-based			random		
	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$
7-zip	0	0	0	20	1	0	4	**100**	0	23	0	0	**96**	**96**	92	94	96	97
BerkeleyDB	0	0	0	68	0	1	**100**	0	0	38	86	79	93	95	98	99	98	99
Dune	**100**	**100**	**100**	**96**	95	99	65	**100**	**100**	**99**	97	**100**	**94**	94	99	92	99	100
Hipacc	0	0	0	36	3	1	23	0	0	27	0	0	**94**	96	**98**	94	99	97
Java GC	0	0	**100**	91	23	1	1	0	0	19	0	0	**99**	95	**91**	96	98	91
LLVM	0	0	0	0	11	29	0	0	0	44	38	7	**95**	**95**	**98**	91	94	96
lrzip	0	0	0	80	74	94	0	0	0	88	52	14	97	95	**100**	99	100	100
Polly	**100**	**100**	**100**	92	**96**	90	**97**	93	**98**	92	94	**97**	94	95	**94**	97	96	94
vpxenc	0	0	0	1	3	0	0	76	0	17	12	0	**98**	**96**	**95**	96	95	94
x264	**100**	0	0	1	0	0	**99**	**100**	0	88	**96**	**99**	95	95	**98**	98	96	96

Figure 6.3: Results of the Kolmogorov Smirnov test if the distribution of the sample set is similar to a uniform distribution for 100 random seeds on different sampling strategies

slower. When we take a look at Figure 6.3, we detect that for all case studies the results of grammar-based sampling are compatible to the results of random sampling. However, the results of all other strategies are quite inconsistent, on some case study they produce many sample sets that are considerable uniform, on others they do not produce any sample set that is considerable uniform.

6.3 Sampling Strategy Performance

In this section, we will discuss $RQ_{2.2}$: What is the execution time of the different sampling strategies?

Observation

In Figure 6.4, we listed the mean sampling time of t-wise, solver-based, randomized solver-based, distance-based, and grammar-based sampling on 10 real-world case-studies. Each column contains the mean execution time of 3 sample sizes for every case study. We used the sample sizes of the sample sets generated with t-wise sampling and $t = 1$, $t = 2$, $t = 3$ and 100 means that all sample sets are uniform. However t-wise sampling does not rely on randomness and thus it only produces 1 sample set per sample set size and case study, which means it gets only 100 or 0 in our table.

We highlighted the least mean sampling time in green, iff the Mann-Whitney U test reported a significant difference ($p < 0.05$) to the mean sampling time of the other sampling strategies for that case study and sample size. We do not differentiate between mean sampling times that are smaller than 1s, because of frequency scaling and timer interrupts of the CPU. As a consequence, we do not highlight a sampling time < 1s, if at least one other sampling strategy has also a mean sampling time < 1s.

In Figure 6.4, we see that the solver-based sampling strategy needs less than 1s to produce a sample set or is significantly faster than all other sampling strategies on every case study. The randomized solver-based strategy is always slower than all other strategies. Grammar-based and t-wise sampling perform similar to each other

	t-wise			solver-based			henard			distance-based			grammar-based		
	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$	$t = 1$	$t = 2$	$t = 3$
7-zip	<1s	3.28s	37.03s	<1s	<1s	10.20s	5.75s	23m 32s	18h 14m	<1s	6.92s	2m 29s	<1s	5.22s	40.42s
BerkeleyDB	<1s	<1s	1.29s	<1s	<1s	<1s	<1s	6.35s	93.61s	<1s	<1s	<1s	<1s	<1s	2.93s
Dune	<1s	1.29s	9.71s	<1s	<1s	<1s	1.21s	2m 13s	39m 35s	<1s	<1s	1.41s	<1s	3.03s	13.08s
Hipacc	<1s	5.58s	72.77s	<1s	<1s	13.61s	14.38s	1h 11m	35h 36m	<1s	8.87s	93.91s	1.41s	24.74s	2m 57s
Java GC	<1s	2.89s	30.32s	<1s	<1s	**8.20s**	2.97s	11m 28s	10h 45m	<1s	9.23s	3m 53s	<1s	4.03s	30.64s
LLVM	<1s	<1s	<1s	<1s	<1s	<1s	<1s	1.02s	8.98s	<1s	<1s	<1s	<1s	<1s	1.27s
lrzip	<1s	<1s	<1s	<1s	<1s	<1s	<1s	7.43s	47.56s	<1s	<1s	<1s	<1s	1.08s	2.69s
Polly	<1s	2.90s	34.70s	<1s	<1s	**3.41s**	2.44s	6m 26s	4h 17m	<1s	7.74s	2m 14s	<1s	4.02s	25.88s
vpxenc	<1s	3.80s	45.84s	<1s	<1s	**10.69s**	3.28s	14m 18s	15h 23m	<1s	61.20s	36m 44s	<1s	4.19s	33.94s
x264	<1s	<1s	<1s	<1s	<1s	<1s	<1s	2.41s	24.82s	<1s	<1s	<1s	<1s	<1s	1.67s

Figure 6.4: Execution time of t-wise, solver-based, randomized solver-based, distance-based, and grammar-based sampling strategies on 10 variability models of real-world software systems

in terms of mean sampling time. However, we note that the mean sampling time of the grammar-based sampling strategy for Hipacc is significantly higher than the mean sampling time of t-wise sampling for that case study. Distance-based sampling performs similar to t-wise and grammar-based sampling in most of the case-studies, but is significantly slower on some case-studies, i.e., the vpxenc case study.

Discussion

In the absence of cross-tree constraints, we detect that the grammar-based sampling strategy is competitive in terms of execution time with the solver-based sampling strategy, i.e., the fastest sampling strategy. In the presence of cross-tree constraints, i.e., the Hipacc case study, the grammar-based sampling strategy needs significantly more time than the solver-based and t-wise strategy. However, the execution time of grammar-based sampling lies below 10 minutes for all case studies and sample set sizes in our evaluation.

6.4 Prediction Error

In this section, we will discuss RQ$_3$: What influence has the chosen sampling strategy on the accuracy of performance prediction?

Observation

In Figure 6.5, we listed the mean prediction error of t-wise, solver-based, randomized solver-based, distance-based, and grammar-based sampling on 10 real-world case-studies. Each column contains the mean prediction error of 3 sample sizes for every case study. We used the sample sizes of the sample sets generated with t-wise sampling and $t = 1$, $t = 2$, $t = 3$. In the last row we give the mean over all mean prediction errors over all 10 case-studies.

We highlighted the least mean prediction error in green, iff the Mann-Whitney U test reported a significant difference ($p < 0.05$) to the mean prediction error of the other sampling strategies for that case study and sample size.

	t-wise			solver-based			henard			distance-based			grammar-based			random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
7-zip	**71.7 %**	44.7 %	24.7 %	91.0 %	62.7 %	20.6 %	91.1 %	62.9 %	19.5 %	112.1 %	18.0 %	15.5 %	81.8 %	11.7 %	9.2 %	84.4 %	13.3 %	10.4 %
BerkeleyDB	126.8 %	7.1 %	**1.1 %**	50.3 %	47.2 %	42.7 %	**46.7 %**	40.4 %	11.1 %	533.8 %	9.2 %	4.9 %	297.6 %	6.5 %	3.3 %	297.5 %	5.1 %	3.4 %
Dune	20.7 %	9.9 %	10.4 %	43.9 %	15.0 %	10.2 %	65.0 %	21.1 %	10.7 %	32.8 %	10.5 %	9.5 %	**3.2 %**	**7.4 %**	**9.0 %**	20.4 %	10.1 %	9.3 %
Hipacc	61.9 %	19.3 %	17.2 %	61.3 %	15.2 %	12.1 %	33.7 %	12.2 %	10.4 %	39.0 %	11.0 %	10.6 %	**5.3 %**	**9.1 %**	**9.6 %**	26.6 %	10.0 %	10.0 %
Java GC	**38.0 %**	25.6 %	14.6 %	57.2 %	75.8 %	40.2 %	43.0 %	33.3 %	26.3 %	85.6 %	13.5 %	10.5 %	72.6 %	11.1 %	9.2 %	70.3 %	10.9 %	9.2 %
LLVM	8.3 %	4.7 %	2.5 %	8.8 %	3.5 %	2.1 %	**4.7 %**	4.0 %	4.2 %	5.4 %	2.8 %	2.2 %	5.2 %	2.2 %	**2.0 %**	5.8 %	2.2 %	2.1 %
lrzip	21.6 %	**2.5 %**	**2.7 %**	46.8 %	32.7 %	19.5 %	141.2 %	55.1 %	11.3 %	89.6 %	16.5 %	4.2 %	**15.7 %**	13.7 %	7.5 %	224.3 %	9.1 %	3.1 %
Polly	22.9 %	**4.4 %**	**3.3 %**	20.3 %	14.6 %	14.2 %	25.1 %	13.8 %	14.5 %	31.2 %	6.7 %	10.3 %	**1.8 %**	6.7 %	5.8 %	30.3 %	6.9 %	5.8 %
vpxenc	**207.3 %**	148.4 %	41.5 %	641.4 %	347.0 %	91.8 %	728.3 %	501.9 %	428.8 %	329.8 %	68.9 %	48.7 %	399.9 %	47.9 %	42.4 %	245.4 %	50.2 %	42.2 %
x264	**10.2 %**	6.4 %	**1.7 %**	21.0 %	20.8 %	21.3 %	14.9 %	8.6 %	11.1 %	14.5 %	6.4 %	5.6 %	15.1 %	4.2 %	3.7 %	16.4 %	4.0 %	3.7 %
Mean	**58.9 %**	27.3 %	12.0 %	104.2 %	63.5 %	27.5 %	119.4 %	75.3 %	54.8 %	127.4 %	16.4 %	12.2 %	89.8 %	12.1 %	10.2 %	102.2 %	12.2 %	**9.9 %**

Figure 6.5: Error rates of t-wise, solver-based, randomized solver-based, distance-based, grammar-based, and random sampling for multiple linear regression on 10 variability models of real-world software systems

In Figure 6.5, we see that on small sample sets t-wise sampling performs significantly better than the other sampling strategies on some case-studies. With an increasing number of samples per sample set the mean prediction error decreases for all sampling strategies. However, the mean prediction error decreases faster for random and grammar-based sampling.

Discussion

In a black-box environment we cannot make assumptions on the influence of single configuration options on the performance of the software product. As a consequence, uniform sampling strategies are used to produce the most accurate performance prediction models. In Figure 6.5, we see that grammar-based sampling has a similar prediction error as random sampling and with increasing sample set size the prediction error decreases faster than for other, non-uniform, strategies.

We see that the prediction error for sample sets with $t = 1$ from t-wise is lower than the prediction error for grammar-based or random sampling. Although random sampling is the baseline for black-box sampling, we note that if the sample set consists of only few samples, i.e., about the number of configuration options, t-wise sampling produces sample sets of higher quality. We suspect that this is due to the fact that t-wise sampling ensures that every configuration option is present in the sample set, whereas sample sets that are created with random or grammar-based sampling might not contain every configuration option at least once in small sample sets. As a consequence of that, the machine learning algorithm is not able to include the influence of the missing configuration option, and therefore the prediction error might increase significantly.

To conclude, we see that for small sample sets, where the sample set size is similar to the number of configuration options, t-wise sampling is superior to all other state-of-the-art sampling strategies, including grammar-based sampling. However, if the sample set size increases the grammar-based sampling strategy performs close to the baseline, and produces higher quality sample sets as all other state-of-the-art sampling strategies.

7. Validity

In this chapter we discuss the validity of our evaluation from Chapter 6. We split this discussion into two main parts — internal and external validity.

7.1 Internal Validity

In this section, we talk about internal threats to the validity of our evaluation results. For each threat we also explain what we did to prevent it from impacting our results.

One threat to the validity of our results for the performance measurements are interrupts from the OS and other external influences on performance. To minimize such influences we choose a minimal OS for our benchmark cluster. Additionally we made sure that all time measurements were done isolated on single cluster cores.

An other threat might be a bug in our implementation of the grammar-based sampling strategy. To find and fix all bugs in the implementation we tested our tool in-depth and checked the produced grammar against the variability model. Moreover, we monitored the conversion from index to configuration and vice-versa to verify its behaviour against our theory. Furthermore, we call a SAT-solver to check each sample if it is a valid configuration, to sort out invalid configurations.

To strengthen the internal validity of the used machine-learning, we used for two of the machine-learning algorithms public available python libraries to reduce the likelihood of implementation bugs on our side.

The used SAT-solver might also be a threat to validity of the evaluation results, as different solver use different search heuristics to break ties and identify solutions. However, this does only apply to sampling strategies that rely on SAT-solver to identify configurations as samples, and our approach only relies on SAT-solver to verify that a sample is a valid configuration. To guarantee the correct behaviour of the implemented Z3 solver we compared all results to a second SAT-solver.

7.2 External Validity

In this section, we explain what we did to improve external validity.

To oppose the possibility that our sampling strategy only produces good sampling sets on specific variability models we have chosen 10 different variability models of real-world software systems from different domains. The configuration spaces of these software systems range from 432 to 216 000 valid configurations and 11 to 54 configuration options.

Another threat to validity is the selection of the machine-learning technique. We used three different kinds of machine-learning techniques to minimize the influence of intrinsic behaviour of each single machine-learning algorithm to strengthen external validity.

8. Conclusion and Future Work

Measuring all configurations of a software system is almost always infeasible. Therefore, small sets of sample configurations are used to measure some configurations and predict the performance of all other configurations. The quality of the prediction depends heavily on the chosen samples in the sample set. Existing sampling strategies try to generate pseudo-random sample sets, because true random sampling is very expensive. In this thesis, we proposed a cheap sampling strategy to produce random samples.

In our approach we have presented a new theoretical approach to convert variability models into a context-free grammar, by preserving all hierarchical structures of the variability model. After this, we proposed an algorithm to convert integers to words in the context-free language. Each word in the context-free language represents a configuration in the variability model, iff all cross-tree constraints are fulfilled. To produce a sample, we check whether the drawn word violates any cross-tree constraints from the variability model. If it does, we discard the word and draw a new one until we have drawn a valid sample. We repeat this until we have drawn the given number of samples. To conclude our approach, we showed that our algorithm produces uniform samples.

In the evaluation, we visualized the distribution of all produced sample sets for one variability model to compare all sampling strategies to a random sampling approach, i.e., sampling randomly from the whole population. We then used statistical tests on all produced sample sets for all 10 variability models to check to which extend the different sampling strategies produce considerably uniform sample sets. Also, we showed the error rate on 10 real-world variability models and compared the performance as well as quality of our approach to 4 state-of-the-art sampling algorithms. We see that our approach is almost competitive with state-of-the-art SMT-solver, i.e. Z3, regarding the time that is needed to produce a sample set, but for sample set sizes $t = 2$ and $t = 3$ it outperforms all 4 state-of-the-art approaches in terms of quality of the performance prediction models that are built from the sample sets.

In the future we have to face two main challenges that we have not addressed in this thesis. First, we have to evaluate the scalability of grammar-based sampling on larger real-world variability models. Therefore, we need to find variability models that can be utilized in SPL Conqueror, especially variability models with more cross-tree constraints. The second challenge is to develop a system to encode all kinds of cross-tree constraints in our grammar, such that the grammar-based configuration space is always equal to the valid configuration space.

A. Appendix

We provide all data and the used version of SPL Conqueror in github repositories. The version of SPL Conqueror is available on https://github.com/Kallistos/SPLConqueror. All data and evaluation scripts are available on https://github.com/Kallistos/Distance-Based_Data.

	t-wise			solver-based			henard			distance-based			grammar-based			random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
7-zip	**62.2 %**	31.5 %	**7.6 %**	92.0 %	118.9 %	41.4 %	98.9 %	105.0 %	32.6 %	239.5 %	26.9 %	11.8 %	148.5 %	19.7 %	10.2 %	150.2 %	17.9 %	9.3 %
BerkeleyDB	70.9 %	25.1 %	2.5 %	57.7 %	47.2 %	38.9 %	**47.7 %**	54.0 %	11.1 %	307.6 %	6.8 %	0.7 %	327.9 %	12.1 %	0.7 %	326.6 %	7.9 %	0.6 %
Dune	**16.1 %**	11.0 %	5.6 %	28.9 %	14.4 %	6.5 %	64.9 %	16.4 %	6.9 %	25.2 %	9.8 %	5.4 %	17.4 %	**9.5 %**	5.2 %	17.5 %	9.9 %	5.3 %
Hipacc	22.9 %	14.8 %	6.5 %	48.3 %	12.9 %	5.1 %	39.9 %	8.3 %	2.1 %	29.3 %	8.4 %	3.1 %	21.0 %	**7.9 %**	1.9 %	20.2 %	8.9 %	2.1 %
Java GC	40.0 %	26.2 %	5.0 %	57.4 %	59.5 %	31.8 %	40.0 %	34.9 %	23.0 %	69.4 %	7.9 %	**3.4 %**	78.8 %	6.4 %	3.6 %	77.8 %	**6.4 %**	3.5 %
LLVM	9.7 %	8.6 %	6.3 %	8.6 %	5.2 %	4.1 %	4.9 %	**4.1 %**	4.0 %	5.1 %	4.3 %	3.9 %	5.4 %	4.3 %	3.9 %	5.4 %	4.2 %	3.9 %
lrzip	**39.0 %**	**10.8 %**	3.2 %	54.1 %	26.4 %	16.5 %	110.0 %	55.7 %	11.7 %	67.6 %	21.8 %	**2.6 %**	148.8 %	17.4 %	2.7 %	180.0 %	16.7 %	3.1 %
Polly	19.3 %	3.5 %	3.4 %	18.2 %	8.6 %	4.0 %	**15.3 %**	11.3 %	6.2 %	24.4 %	5.8 %	4.6 %	25.8 %	3.8 %	2.4 %	25.4 %	**3.3 %**	2.0 %
vpxenc	**187.9 %**	21.9 %	6.0 %	674.9 %	306.5 %	39.7 %	786.9 %	1014.3 %	272.4 %	312.2 %	22.2 %	4.8 %	275.8 %	10.6 %	3.2 %	251.8 %	10.6 %	3.6 %
x264	21.9 %	3.8 %	1.4 %	29.3 %	33.6 %	37.0 %	**16.7 %**	10.5 %	10.2 %	23.0 %	4.9 %	1.4 %	21.3 %	3.5 %	0.6 %	20.7 %	4.0 %	0.5 %
Mean	**49.0 %**	15.7 %	4.7 %	106.9 %	63.3 %	22.5 %	122.5 %	131.4 %	38.0 %	110.3 %	11.9 %	4.2 %	107.1 %	9.5 %	3.4 %	107.6 %	9.0 %	3.4 %

Figure A.1: Error rates of t-wise, solver-based, randomized solver-based, distance-based, grammar-based, and random sampling for random forests on 10 variability models of real-world software systems

	t-wise			solver-based			henard			distance-based			grammar-based			random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
7-zip	56.9 %	54.1 %	81.8 %	111.7 %	108.8 %	82.5 %	**54.9 %**	**53.6 %**	**53.5 %**	154.8 %	87.1 %	92.5 %	94.9 %	91.1 %	88.5 %	92.3 %	90.1 %	88.4 %
BerkeleyDB	61.3 %	**51.1 %**	**43.7 %**	72.0 %	55.3 %	60.6 %	**58.4 %**	53.8 %	52.1 %	204.6 %	75.7 %	50.9 %	193.8 %	96.0 %	75.7 %	186.5 %	90.9 %	74.4 %
Dune	**17.4 %**	**17.0 %**	16.8 %	19.2 %	18.2 %	17.2 %	32.8 %	17.1 %	**16.8 %**	18.9 %	17.7 %	17.3 %	17.6 %	17.3 %	17.1 %	17.6 %	17.3 %	17.1 %
Hipacc	24.8 %	15.6 %	13.5 %	56.1 %	17.6 %	13.6 %	26.2 %	14.7 %	12.9 %	31.2 %	14.7 %	13.6 %	**24.5 %**	13.9 %	12.7 %	25.0 %	14.0 %	12.8 %
Java GC	47.0 %	37.3 %	**32.8 %**	55.4 %	55.1 %	45.5 %	**41.5 %**	39.1 %	33.6 %	53.1 %	42.5 %	40.7 %	51.3 %	37.2 %	37.7 %	49.4 %	37.2 %	37.7 %
LLVM	11.7 %	9.2 %	7.4 %	8.8 %	6.0 %	4.4 %	6.7 %	5.2 %	4.3 %	6.5 %	5.0 %	4.3 %	6.4 %	5.1 %	4.2 %	6.4 %	5.1 %	4.1 %
lrzip	**59.3 %**	**62.6 %**	136.6 %	86.3 %	66.2 %	103.2 %	135.2 %	122.3 %	144.2 %	61.7 %	73.3 %	102.3 %	120.4 %	132.4 %	134.9 %	125.9 %	127.3 %	132.4 %
Polly	41.2 %	**21.4 %**	24.8 %	29.5 %	26.8 %	23.0 %	29.4 %	27.1 %	22.6 %	29.7 %	26.6 %	23.3 %	29.3 %	26.8 %	24.0 %	29.4 %	26.8 %	23.7 %
vpxenc	**133.9 %**	**68.2 %**	**53.8 %**	818.0 %	346.6 %	134.8 %	1393.3 %	1463.6 %	542.3 %	304.8 %	100.5 %	78.6 %	239.1 %	93.9 %	76.5 %	246.3 %	94.1 %	76.7 %
x264	**33.4 %**	**32.8 %**	30.6 %	45.5 %	42.6 %	42.4 %	35.1 %	33.0 %	31.8 %	36.2 %	34.9 %	30.0 %	35.8 %	33.7 %	30.4 %	35.9 %	33.8 %	30.7 %
Mean	**48.7 %**	**36.9 %**	44.2 %	130.3 %	74.3 %	52.7 %	181.4 %	182.9 %	91.4 %	90.1 %	47.8 %	45.4 %	81.3 %	54.7 %	50.2 %	81.5 %	53.7 %	49.8 %

Figure A.2: Error rates of t-wise, solver-based, randomized solver-based, distance-based, grammar-based, and random sampling for support vector machines on 10 variability models of real-world software systems

	Mann-Whitney U test [p value ($\hat{A}_{12}$)]																	
	t-wise			solver-based			henard			distance-based			grammar-based			random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
t-wise				10^{-21} (0.62)	10^{-81} (0.75)	10^{-37} (0.66)		10^{-80} (0.74)	10^{-51} (0.69)	10^{-16} (0.60)								
solver-based																		
henard				10^{-12} (0.59)	10^{-03} (0.54)					10^{-08} (0.57)								
distance-based		10^{-48} (0.69)	10^{-11} (0.59)		10^{-203} (0.89)	10^{-92} (0.76)		10^{-229} (0.92)	10^{-131} (0.81)									
grammar-based	10^{-40} (0.67)	10^{-147} (0.83)	10^{-65} (0.72)	10^{-74} (0.73)	10^{-259} (0.94)	10^{-162} (0.85)	10^{-41} (0.67)	10^{-282} (0.96)	10^{-209} (0.90)	10^{-69} (0.73)	10^{-57} (0.71)	10^{-46} (0.68)				10^{-39} (0.67)	10^{-04} (0.54)	
random		10^{-143} (0.83)	10^{-56} (0.70)	10^{-12} (0.59)	10^{-253} (0.94)	10^{-149} (0.84)		10^{-285} (0.97)	10^{-193} (0.88)	10^{-08} (0.57)	10^{-47} (0.69)	10^{-40} (0.67)						

Figure A.3: One-sided Mann-Whitney U test results whether the sampling strategy of the row has a significantly lower error rate than the sampling strategy of the column for multiple linear regression

	Mann-Whitney U test [p value ($\hat{A}_{12}$)]																	
	t-wise			solver-based			henard			distance-based			grammar-based			random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
t-wise				10^{-26} (0.64)	10^{-69} (0.73)	10^{-55} (0.70)		10^{-32} (0.65)	10^{-31} (0.65)	10^{-58} (0.71)			10^{-46} (0.68)			10^{-41} (0.67)		
solver-based										10^{-03} (0.54)								
henard				10^{-13} (0.59)	10^{-06} (0.56)	10^{-07} (0.56)				10^{-23} (0.63)			10^{-19} (0.61)			10^{-17} (0.61)		
distance-based		10^{-91} (0.76)	10^{-69} (0.73)		10^{-213} (0.90)	10^{-146} (0.83)		10^{-133} (0.82)	10^{-117} (0.80)									
grammar-based		10^{-139} (0.82)	10^{-94} (0.77)		10^{-238} (0.93)	10^{-177} (0.87)		10^{-160} (0.85)	10^{-146} (0.83)	0.04 (0.52)	10^{-13} (0.59)	10^{-15} (0.60)						
random		10^{-130} (0.81)	10^{-96} (0.77)		10^{-236} (0.92)	10^{-177} (0.87)		10^{-156} (0.84)	10^{-145} (0.83)	0.03 (0.52)	10^{-10} (0.58)	10^{-16} (0.60)						

Figure A.4: One-sided Mann-Whitney U test results whether the sampling strategy of the row has a significantly lower error rate than the sampling strategy of the column for random forests

	Mann-Whitney U test [p value ($\hat{A}_{12}$)]																	
	t-wise			solver-based			henard			distance-based			grammar-based			random		
	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$	$t=1$	$t=2$	$t=3$
t-wise				10^{-27} (0.64)	10^{-55} (0.70)		10^{-04} (0.54)	10^{-41} (0.67)		10^{-05} (0.55)	10^{-34} (0.66)		0.01 (0.53)	10^{-34} (0.66)		10^{-03} (0.53)	10^{-37} (0.66)	
solver-based																		
henard			10^{-10} (0.58)	10^{-05} (0.55)	10^{-09} (0.57)	10^{-13} (0.59)						10^{-09} (0.58)			10^{-14} (0.60)			10^{-13} (0.59)
distance-based				10^{-16} (0.60)	10^{-10} (0.58)	0.03 (0.52)	10^{-03} (0.54)											
grammar-based				10^{-21} (0.62)	10^{-07} (0.57)		10^{-07} (0.56)											
random				10^{-20} (0.62)	10^{-07} (0.57)		10^{-06} (0.56)											

Figure A.5: One-sided Mann-Whitney U test results whether the sampling strategy of the row has a significantly lower error rate than the sampling strategy of the column for support vector machines

Bibliography

[1] Akers, S. B. (1978). Binary Decision Diagrams. *IEEE Transaction on Computers*, 27(6):509516. (cited on Page 9)

[2] Apel, S., Batory, D., Kstner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines: Concepts and Implementation*. Springer Publishing Company. (cited on Page 3)

[3] Barrett, C. and Tinelli, C. (2018). *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing. (cited on Page 10)

[4] Bastian, P., Blatt, M., Dedner, A., Dreier, N., Engwer, C., Fritze, R., Gräser, C., Kempf, D., Klöfkorn, R., Ohlberger, M., and Sander, O. (2019). The DUNE Framework: Basic Concepts and Recent Developments. *CoRR*, abs/1909.13672. (cited on Page 23)

[5] Boser, B. E., Guyon, I. M., and Vapnik, V. N. (1992). A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Workshop on Computational Learning Theory*, COLT 92, pages 144–152. Association for Computing Machinery. (cited on Page 7)

[6] Bryant, R. (1986). Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691. (cited on Page xi and 10)

[7] Darling, D. A. (1957). The kolmogorov-smirnov, cramer-von mises tests. *The Annals of Mathematical Statistics*, 28(4):823–838. (cited on Page 29)

[8] de Moura, L. M. and Bjørner, N. (2008). Z3: An Efficient SMT Solver. In Ramakrishnan, C. R. and Rehof, J., editors, *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer. (cited on Page 10)

[9] Henard, C., Papadakis, M., Harman, M., and Le Traon, Y. (2015). Combining Multi-Objective Search and Constraint Solving for Configuring Large Software Product Lines. In *Proceedings of the International Conference on Software Engineering (ICSE)*, ICSE 15, page 517528. IEEE Press. (cited on Page 4, 6, 9, 11, and 21)

[10] Johansen, M. F., Haugen, O., and Fleurey, F. (2012). An Algorithm for Generating T-Wise Covering Arrays from Large Feature Models. In *Proceedings of the International Software Product Line Conference (SPLC)*, SPLC 12, page 4655. Association for Computing Machinery. (cited on Page 4, 6, and 12)

[11] Kaltenecker, C., Grebhahn, A., Siegmund, N., Guo, J., and Apel, S. (2019). Distance-Based Sampling of Software Configuration Spaces. In *Proceedings of the International Conference on Software Engineering (ICSE)*, ICSE 19, pages 1084 – 1094. IEEE Press. (cited on Page xi, 6, 11, 12, and 21)

[12] Kam, H. T. (1995). Random decision forest. In *Proceedings of the International Conference on Document Analysis and Recognition*, volume 1416, pages 278–282. Montreal, Canada, August. (cited on Page 7)

[13] Kruskal, W. H. and Wallis, W. A. (1952). Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):583–621. (cited on Page 30)

[14] Lattner, C. and Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO).* (cited on Page 24)

[15] Mann, H. B. and Whitney, D. R. (1947). On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):50–60. (cited on Page 30)

[16] Marijan, D., Gotlieb, A., Sen, S., and Hervieu, A. (2013). Practical Pairwise Testing for Software Product Lines. In *Proceedings of the International Software Product Line Conference (SPLC)*, SPLC 13, pages 227–235. Association for Computing Machinery. (cited on Page 4, 6, 12, and 21)

[17] Marten, A. (2018). A Comparison Study of Domain Constraint Solver for Model Counting. Master's thesis, University of Passau. (cited on Page 9)

[18] Oh, J., Batory, D., Myers, M., and Siegmund, N. (2017a). Finding Near-Optimal Configurations in Product Lines by Random Sampling. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, ESEC/FSE 2017, pages 61–71. Association for Computing Machinery. (cited on Page 9)

[19] Oh, J., Batory, D. S., Myers, M., and Siegmund, N. (2017b). Finding Product Line Configurations with High Performance by Random Sampling. (cited on Page 9)

[20] Shubham, Sharma and Rahul, Gupta and Subhajit, Roy and Kuldeep, S. Meel (2018). Knowledge Compilation meets Uniform Sampling. In Barthe, G., Sutcliffe, G., and Veanes, M., editors, *LPAR. International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 57 of *EPiC Series in Computing*, pages 620–636. EasyChair. (cited on Page 10)

[21] Siegmund, N., Grebhahn, A., Apel, S., and Kästner, C. (2015). Performance-Influence Models for Highly Configurable Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 284–294. Association for Computing Machinery. (cited on Page 3)

[22] Siegmund, N., Rosenmüller, M., Kuhlemann, M., Kästner, C., Apel, S., and Saake, G. (2012). SPL Conqueror: Toward optimization of non-functional properties in software product lines. *Software Quality Journal*, 20(3):487–517. (cited on Page 6)

[23] Vargha, A. and Delaney, H. D. (2000). A Critique and Improvement of the CL Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132. (cited on Page 30)