Bachelor Thesis in Computer Science

A Comparison of Optimization and Craig Interpolation for Generating Minimal and Maximal Configurations

Elisabeth Griebl

2017-10-30

Supervisor: Prof. Dr.-Ing. Sven Apel

Tutor: Andreas Stahlbauer, M.Sc

Griebl, Elisabeth:
A Comparison of Optimization and Craig Interpolation for Generating Minimal and Maximal Configurations
Bachelor Thesis, University of Passau, 2017.

We all need people who will give us feedback.
That's how we improve.

— Bill Gates

Dedicated to my family, my boyfriend,
and all the wonderful people
who have supported me and my work.

Abstract

This work provides a comparison of two instantiation algorithms, one based on Craig interpolation, the other on optimization. They are used to produce configurations for a software product line with the minimal or maximal number of features enabled using a SMT solver. To evaluate both approaches, we implemented the algorithms using the `Z3` solver and tested their performance for different problems. We use the algorithms, inter alia, in combination with different preceding sampling algorithms or different complexities of variability models. Overall, Craig interpolation performs better on some edge cases, but using optimization is most suitable for most test cases.

Contents

List of Figures

List of Tables

1 Introduction

Software Engineering is a challenging problem in times of rapid technological development and constantly changing demands. Using *software product lines* has proven as an applicable strategy to respond flexibly to the user's needs. In simple terms, software product lines are configurable products, which share a base of common functions and differ in other parts [Ape+13]. As a software product line often produces thousands of finely graduated different products, testing, inter alia, holds numerous difficulties. One of them is the sheer number of tests, which is needed to prove reliability and correctness of the whole product line. To deal with this problem, we need to continuously improve established testing techniques and try to find new solutions.

1.1 Motivation

Constantly testing all executable tasks of highly configurable software often exceeds time or calculating resources. Therefore, current research concentrates on more efficient ideas with fewer test cases, which still provide a high code coverage. One common approach in software testing is selecting some specific products (*instances* of the product line), which are tested to find bugs in the whole product line. Medeiros et al. found that some of those so-called "sampling strategies" are very efficient in selecting instances of the product line, which bring up multiple faults [Med+16].

Finding suitable instances of the product line for testing purposes becomes increasingly difficult depending on the number of "configuration options" and their dependencies. Considering all the constraints by constructing a suitable instance of a product line becomes a logical problem. Hence, SMT solvers are more and more important in this field.

1.2 Contributions of this Thesis

The main contribution of our work is to examine the connection between the time, which is needed by the solver to find suitable instantiations of the product line and the formulation of the problem, which needs to be solved. For this purpose, we use two concepts, optimization and Craig interpolation, which formulate, for example, the dependencies between the configuration options and further constraints. In theory, one of the concepts could ease it for the solver to help finding a suitable instantiation of the product line. To review this idea, the concepts are implemented in the framework `Simtopia`.

In detail, we will identify the performance differences for different complexities of variability models by answering research question 1 (**RQ1**). Furthermore, we will review if there are any performance differences for the algorithms in combination with T-wise and random sampling (**RQ2.1** and **RQ2.2**), and finally look for performance differences for different configuration modes (**RQ3**)[1]. In our work, we provide and discuss the results of the different time measurements.

1.3 Structure of this Thesis

This thesis is structured as follows: Chapter 2 on the facing page gives some background information about software verification, explains some logical concepts, and describes the framework `Simtopia`. In Chapter 3 on page 13, we describe how we use optimization and Craig interpolation as instantiation algorithms. This includes a detailed explanation of the implementation of both algorithms. Then, we evaluate our implementation and present our results in Chapter 4 on page 21. After that, we discuss the results in Chapter 5 on page 29, answer the research questions, present some possible improvements, and give an outlook to future research. We conclude with a summary of our work in Chapter 6 on page 33.

[1] Please find a detailed explanation of the terms, which are used in the research questions in the background chapter.

2

Background

In this chapter we present the background of this thesis. First, we will give some basic information about product lines and variability models. Then we will present the basics of satisfiability modulo theories briefly. Furthermore, we will explain some basic concepts like Craig interpolation, optimization, and sampling algorithms, which are relevant to understand the components of the framework we use for our work. Finally, we will introduce the framework `Simtopia`, which is used for our studies.

2.1 Product Lines and Variability Models

In general, *product lines* are groups of products, which have similar characteristics. Relating to computer science that means we have a software, which can perform several similar tasks and therefore use a base of shared code [Ape+13]. The particular task, which the software performs depends on its *feature selection* or *configuration*[1] „ which is a set of enabled or disabled *features*. Their *assignment* (id est, if the features are dis- or enabled) determines the task, which the product performs. Apel et al. defined a feature as follows [Ape+13]:

Definition 1. "A feature is a characteristic or end-user-visible behaviour of a software system. Features are used in product-line engineering to specify and communicate commonalities and differences of the products between stakeholders, and to guide structure, reuse, and variation across all phases of the software life cycle."

A popular example for a product line is the `Linux` kernel, which operates on many different hardware platforms and offers a consistent API to user programs across these platforms. To provide this functionality, it dis- or enables different features, depending on the platform, it is running on.

The features of the software are connected by *feature dependencies*, id est, they can influence each other. One simple example for features influencing each other is $(a \rightarrow b)$. That means we have two features, a and b. If a is enabled, b needs to be enabled as well. Those relationships of the features are collected in a *variability model*, which gives meta information about valid assignments of features in the program. A valid configuration of a program needs to satisfy the variability model, that means the assignment of

[1] We will use the term *configuration* in our work.

```
c This is an example for the dimacs format.
p cnf 3 2
2 -3 0
2 3 -1 0
```

Fig. 2.1 An example DIMACS document

the features may not contradict the variability model [Ape+13]. Assuming $(a \rightarrow b)$ is a complete variability model and there are only two features a and b, for example (a, b) or $(\neg a, \neg b)$ would be valid configurations.

2.1.1 Representation of Variability Models

Logical problems can be expressed in *conjunctive normal form* (*CNF*)[Preo9]. Since any propositional formula can be converted into an equivalent expression in CNF, the syntax is both simple and powerful.

A CNF formula consists of *literals* l_j, which can be either a Boolean variable or it's negation and represent a single feature in our case. We can connect two literals by logical "OR", which is represented by "$\vee$". Two literals, which are connected by $\vee$ are called disjunct. We can group several disjunct literals and get a *clause* c_i. CNF formulas are expressed as a conjunction of several clauses, id est, they are connected by logical "AND" ($\wedge$).

Summing up, we can formalize logical expressions E in CNF syntax as follows:

$$E = \bigwedge_i \bigvee_j l_{i,j}$$

The computational complexity of problems in CNF is depending on the minimal number of variables k per clause. For instance, expressions with $k = 2$ for all clauses or *2-SAT* problems, can be solved in polynomial time [Kro67]. Expressions with $k = 3$ for all clauses are called *3-SAT* problems. In 1971, Stephen Cook stated that computing 3-SAT problems is NP-complete [Coo71]. One year later, Richard Karp published a proof for the NP-completeness of 21 computational problems, including 3-SAT problems [Kar72]. k-SAT problems are also called *Boolean satisfiability problems*.

A wideley accepted format for expressing Boolean satisfyability problems in CNF is the DIMACS format, which simplifies automated solving [93]. As we can see in Figure 2.1, a file in the DIMACS format has three parts:

1. several lines of comments at the beginning of the file, indicated by the letter "c"

2. one line of meta data (defining the format, the number of variables, and the number of logical statements), indicated by the letter "p"

3. several lines of logical statements, which are presented in the rest of the document.

The syntax of those logical statements is quite similar to CNF: each line, delimited by the digit "0", represents one clause, each digit represents one literal. A minus ("-") before a digit is equivalent to a literal with a negation ("¬"). Like in CNF, the clauses are connected with logical "AND", the literals within the line with "OR".

As we now understand the syntax of DIMACS, we can explain the meaning of the file in Figure 2.1: The document starts with one comment line. The second line tell us that the file in CNF syntax has three different literals and two clauses. The rest shows us some examples for logical statements in the DIMACS format. Represented in CNF, their equivalent (representing each feature 1, 2, 3 by a literal l_1, l_2, l_3) is:

$$(l_2 \vee \neg\, l_3) \wedge (l_2 \vee l_3 \vee \neg l_1)$$

This simple document is an example for a "small" variability model. Concerning the *complexity* of variability models, we considered two parameters relevant for our work: the amount of features that appear in the variability model and the number of clauses.

2.2 Satisfiability Modulo Theories

First-order formulas consist of a set of variables, which are connected by $\wedge$ (AND), $\vee$ (OR), $\Longrightarrow$ (implies), quantifiers ($\forall$ which means for all and $\exists$ which means there exists), the equality symbol (=), negations ($\neg$) and parenthes [Bar77]. The field of *Satisfiability Modulo Theories* (*SMT*) deals with the satisfiability of those first-order formulas with respect to a first-order background theory [Seb07] [Bar+09]. SMT first occured in the late 1970s and is used for software verification purposes for many years [Bar+09] [CHN12]. SMT is a generalization of Boolean satisfiability problems, which is produced by "adding equality reasoning, arithmetic, fixed-size bit-vectors, arrays, quantifiers, and other useful first-order theories" [MB08].

SMT solvers are tools to decide if the given formula is satisfiable with respect to the given theory [MB08] [Bar+09]. Some examples for SMT solvers are MathSAT5, SMTInterpol or Z3[Cim+13] [CHN12] [MB08]. The framework Simtopia (2.6) uses SMT solvers to find valid assignments for features in configurations.

2.3 Craig Interpolation

Craig interpolation is a technique from logical mathematics, which has been established as a approximation tool in software verification. It is defined by the following theorem, which was proved by Wiliam Craig in 1957 [Cra57]:

Theorem 1. (Craig's interpolation theorem [Cra57]): Let φ^-, φ^+ be a pair of formulae, such that $\varphi^- \wedge \varphi^+$ is unsatisfiable. Then there exists a formula ψ (called Craig interpolant) that fulfills:

(i) the implication $\varphi^- \Rightarrow \psi$ holds,
(ii) the conjunction $\psi \wedge \varphi^+$ is unsatisfiable, and
(iii) ψ only contains symbols that occur in both φ^- and φ^+.

If we choose φ^- and φ^+ wisely, the Craig interpolant can be used in many domains. For example, Craig interpolation plays an important role in model checking [McM05]. Furthermore, many verification tools like SMT solvers use some ideas of Craig interpolation to create abstractions from state spaces or derive loop invariants, for example [CHN12]. In 2012, Christ et al. introduced `SMTInterpol`, an interpolating SMT solver, which works on the basis of Craig interpolation [CHN12].

2.4 Optimization

In general, optimization in mathematics is the process of finding the best assignment for one or more values with respect to an optimization criterion. A simple use case for optimization is finding an input value v for a function f, such that $f(v)$ is the total minimum of the function, if such v exists. Another use case for optimization is solving *constraint satisfaction problems* (*CSP*s). CSPs are optimization problems where we need to select the best assignment from a finite set of possible assignments for several variables [BPS99].

As the cost for solving CSPs is exponential in the number of variables there are special heuristical CSP-solvers [BPS99].

2.5 Sampling Strategies

Testing product lines is a challenging problem. If we have no further constraints we can generate 2^x different configurations for binary features, with x being the number of features of the product line. In theory, we should test all those configurations to verify that the software works faultlessly and has no configuration related faults. Since the number of configurations, which need to be tested tends to explode in practice, we need to select particular configurations. For this purpose, we use *sampling strategies* in software verification [Med+16]. A good sampling algorithm can help to find suitable configuration candidates to test at least big parts of the software. To meet this target, sampling algorithms generate configurations according to the following pattern:

1. At first, sampling algorithms generate configurations with an equal distribution of the enabled features. At best, all features are dis-/enabled with equal frequency. To reach this target, sampling algorithms have several differing strategies. Two of those strategies are described in the next section.

2. Additionally, sampling algorithms can involve *covering arrays*. Assuming that a feature is tested adequately by being tested once, the idea behind covering arrays is to enable each feature in only one produced

configuration at best. The covering arrays are used to record, which features are already enabled in a previously produced configuration. Depending on those records, the algorithm will try not to enable already covered features again. This leads to the least possible number of configurations, which covers all features at least once.

3. As not all combinations of dis- and enabled features are valid due to feature dependencies, sampling algorithms can involve the variability model in addition. This results in generating only valid configurations according to the variability model.

The usage of the term "sampling" is slightly different in literature. For our work, we use the term to name the strategy, which is used to achieve an equal distribution of enabled features (id est only the first point of the previous list). We decided this based on two points: first, we think that the sampling strategy is the "core part" of the sampling algorithm. Second, the term sampling names *at least* the core sampling strategy in literature. That means, using sampling to denominate the strategy is the lowest common denominator. The sampling algorithm, which is implemented in `Simtopia` also provides this functionality only.

There are several strategies, which have proven useful. In the following sections we want to introduce two common sampling strategies briefly.

2.5.1 T-Wise Samping

The first strategy we want to explain is called *T-wise sampling*. Technically, T is a variable, which defines how many features are enabled/disabled at once. For instance, pair-wise strategies (id est $T = 2$) group the features in equally assigned pairs. As we increase T, the sizes of the sample sets also increase [Med+16]. That means T is often assigned with small values (like 1, 2 or 3) in practice.

2.5.2 Random Sampling

In contrast to T-wise strategies, *random sampling strategies* do not select configurations systematically. The configurations are generated randomly, all features are enabled or disabled by chance.

2.6 The Framework `Simtopia`

`Simtopia` is a Java tool for deriving program variants from a product-line simulator. It gets a C program and a variability model in `DIMACS` format as input. It computes and returns valid product variants and their corresponding configurations, which satisfy the variability model. Therefore, it implements several algorithms to generate initial configurations and uses SMT solvers to find valid assignments for the features according to the variability model and prove the configurations' validity. Furthermore, one can select several

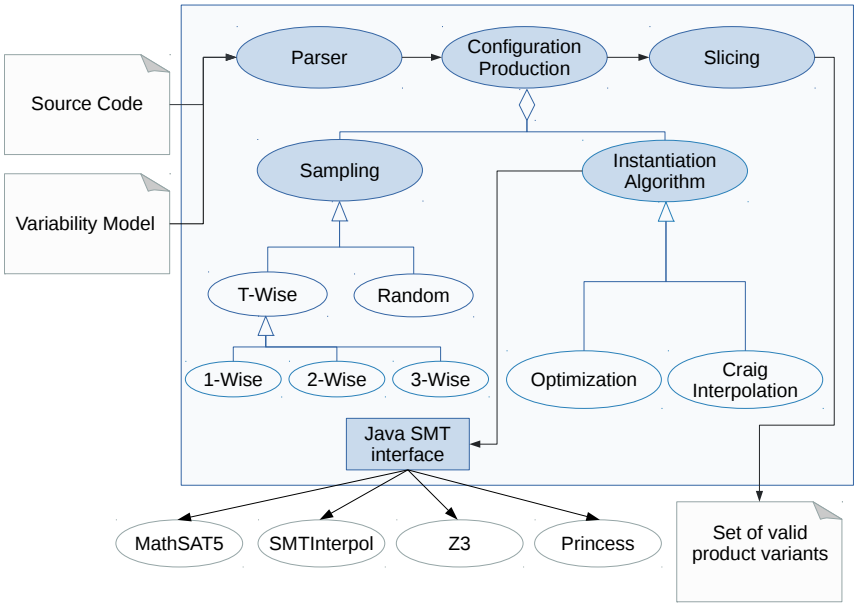


Fig. 2.2 Architecture of the framework `Simtopia`

options for the output configurations, for example considering only features in the output configuration, which are referenced by the C input program or delivering configurations with the minimal or maximal number of features enabled in the output configurations.

2.6.1 *Architecture of* `Simtopia`

Like we can see in Figure 2.2, `Simtopia` has three basic processing components, namely a parser, a configuration-production module, and a slicing module. Furthermore, `Simtopia` is configurable with respect to the following configuration options:

- *mode*: `Simtopia` produces suitable configurations for the input program. Medeiros et al. found, that testing software by using two configurations (one with most features enabled, the other with most features disabled) is most efficient with respect to the number of configurations, which need to be tested, and the number of faults, which are detected [Med+16]. For that reason, `Simtopia` enables the minimal or maximal number of features, which is possible based on the current initial configuration (which is produced by the sampling algorithm and does not consider the variability model, like explained in Section 2.5) and the constraints of the variability model. The configuration option of `Simtopia`, which expands the constraints for building output configurations, is called *mode* and can be assigned with the values MAX ENABLED or MAX DISABLED.
- *feature mapping*: `Simtopia` can consider the input program for producing the output configuration. If that option is activated, features,

which are not referenced by the C input program are not considered for building the output configuration. This includes to discard all clauses of the variability model, which do not include referenced features. In practice, this option is very useful if we have a valid pair of an input program and a corresponding variability model, for which we want a set of valid product variants as fast as possible. As we are not interested in complete and valid product instances in this thesis but only in the time, which is needed for the producing configurations, we deactivate this configuration option for our work.

- *feature selection*: `Simtopia` provides the possibility of selecting randomly a certain number of the features, which are mentioned in the variability model. Feature selection is an artificial simulation of the *feature mapping* option. Technically, it produces a list of features, which need to be considered for building an output configuration, similar to the list of features, which are referenced in the input program. Like in the feature mapping option, clauses of the variability model, which do not contain any features of that list are discarded. In contrast to the feature mapping option, we can control the number of features, which are part of the list. That makes feature selection more useful for scientific purposes, as we can use it to "norm the problem size". This configuration option is not useful in practice since it considers certain parts of the software randomly, without checking if this selection makes sense.

- *seeds*: `Simtopia` uses seeds for every (semi-)random computation for repeatability reasons. This configuration option can be used to set a specific seed. Please note, that `Simtopia` uses its standard seed, if the *seed* option is not set manually. That means "random" computations of `Simtopia` are always "semi-random" in practice.

These are not all, but all relevant configuration options for our work.

Considering `Simtopia`'s configuration, the input (C input program, corresponding variability model) passes through the consecutively arranged components (parser, configuration-production with solvers and slicing). In the following, we will explain those components more detailed.

The Parser

We use the `Eclipse CDT parser` in our framework. The parser converts the input in an abstract syntax tree and prepares some meta data, which is useful later. For example, it finds the corresponding features of the variability model and the C input program or collects data about the numbers of clauses in the variability model.

The Configuration-Production

The production of configurations is the core part of Simtopia. Starting with a sampling algorithm, which generates first configurations without considering the variability model, it passes through an instantiation algorithm afterwards. Simtopia provides two variants of sampling: T-wise and random sampling. They differ in the way they generate *initial configurations*. Initial configurations are the configurations with an equal distribution of dis- and enabled features, which do not consider the variability model[2]. Both strategies were introduced in Section 2.5. The initial configuration is transmitted to an *instantiation algorithm* afterwards.

Finding valid configurations, which satisfy the variability model is a highly complex Boolean satisfiability problem in many cases, so we use specialized solvers, which use some heuristics. The instantiation algorithm is responsible for phrasing the initial configuration (given by the sampling algorithm), the constraints of the variability model, and further constraints of the configuration of Simtopia (like the mode, which is described in Section 2.6.1) in a solver-processable way [3].

Solvers

Simtopia provides several SMT solvers. They are connected to the framework via JavaSMT, an "unified interface for SMT solvers in java" [KFB16]. Currently, Simtopia supports MathSAT5, SMTInterpol, Z3, and Princess [Cim+13; CHN12; MB08; Rüm17]. Please find more information about the concept of SMT solvers in Section 2.2.

For our work, we use Z3, which is freely available SMT Solver, developed by Microsoft Research[MB08]. It is written in C++ and supports several input formats like SMT-LIB [RT06]. Unlike many other SMT solvers, Z3 accepts CSPs as an input, which means it can perform optimization.

Slicing

The slicing module analyses the C input program and searches for conditions, which contain feature variables. If the module detects such an appropriate condition, it evaluates its current assignment using the specific configuration, which was produced by the configuration-production module. If the assignment of the condition is "false" due to the assignment of the variables, the affected code of the C program is removed. Overall, the slicing module creates one corresponding version of the input program for every configuration, which was computed previously.

Finally, Simtopia derives full and valid product variants, which include the variability model, one or more configurations (created by the configuration

[2]Please find a detailed explanation about the configurations, which are produced by the sampling algorithm, which is used in Simtopia in Section 2.5.

[3]Chapter 3 explains how the instantiation algorithms of Simtopia work in detail, the ideas behind the instantiation algorithms are explained in the Sections 2.4 and 2.3, respectively.

production module), and their corresponding processed C program (generated by the slicing module).

3

Instantiation Algorithms

In this chapter, we explain how we use optimization and Craig interpolation as instantiation algorithms. As we have introduced in Section 2.6, instantiation algorithms are used for expressing an initial configuration (which is produced by the sampling algorithm), the variability model, and all further constraints of `Simtopia` in a solver processable way[1].

The instantiation algorithm is part of the configuration-production module of `Simtopia`, so its goal is the generation of product configurations, which are valid with respect to the variability model and satisfy all further constraints. Informally spoken, the instantiation algorithm is the link between sampling and solving. From the sampling algorithm, we get a set of *initial* configurations, which do not consider the variability model. The instantiation algorithm combines the initial configurations, adds all further constraints (like the variability model), and gives it to the solver. The main difference of the different implementations is the embedding of the solver. This leads us directly to the core idea of our work: since the specific task, which the solver performs depends on the formulation of the problem, the solver might be able to compute a solution faster with one of the implementations.

In this chapter we want to introduce the different implementations of the instantiation algorithm, which are based on optimization and Craig interpolation. At first, we will introduce the identical in- and output of both algorithms (called "frame" in the following) and afterwards present the "core algorithm", which is different for optimization and Craig interpolation.

3.1 The Framework

In the following, we see the framework of both instantiation strategies.

As we can see in Interface 1, the instantiation algorithm has four input arguments, which we will explain in the following: the first argument is *VM*. *VM* is the variability model which is part of the input of `Simtopia`. As explained in the background section, it contains information about valid feature assignments and all dependencies between them.

The second argument is referenced in Interface 1 as a "bipartite (enabled, undefined) set" of predefined features. *IC* is one specific initial configuration of a set of configurations, which is produced by the preceding sampling algorithm. Spread over the whole set of initial configurations, all features

[1]The interactions between the components of `Simtopia` are introduced in Section 2.6

Interface 1 : Frame of Instantiation Algorithm (VM, IC, R, min)	
input	: *VM*: variability model *IC*: initial configuration, bipartite (enabled, undefined) set *R*: set of relevant features *min*: Boolean, true iff #*enabled features* should be minimized in the output config, maximized otherwise
output	: *OC*: tripartite (enabled, disabled, irrelevant) set of features, which satisfies *VM* and has the minimal (or maximal, depending on *min*) #*enabled features*

are enabled with an equal distribution by the sampling algorithm without considering the variability model or any further constraints. That means, *IC* is a temporary pre-stage of one result configuration. "Bipartite" means, that the features are partitioned in disabled and enabled ones.

The third argument (*R*) is described as a set of relevant features. Initially, all features, which are referenced in the variability model are relevant features, so the set includes all features in the first instance. The argument refers to the configuration options featureMapping and featureSelection of `Simtopia`[2]). Both configuration options restrict the number of relevant features in the variability model. By using at least one of those configuration options, some features become irrelevant. As they are not considered for the result configuration anymore, those features are removed from the list of relevant features.

The last argument is the Boolean parameter *min*. It refers to `Simtopia`'s configuration option mode. We have seen in the background that `Simtopia` generates configurations with the minimal or maximal number of features enabled. This depends on the configuration of `Simtopia`, which is defined by the user before program start. If `Simtopia` is configured in MAX DISABLED mode the parameter *min* is true, if we choose the MAX ENABLED mode *min* is false.

As a result, the instantiation algorithm produces a full and valid configuration with respect to all given constraints. Since the features can be enabled, disabled or irrelevant, the output configuration *OC* is tripartite. The following sections describe how the different implementations of the instantiation algorithm generate this output configuration in detail.

3.2 Optimization Instantiation Algorithm

In the background chapter, we have learned that optimization can be used for solving constraint satisfaction problems (CSPs), where we need to select

[2]Reminder: featureMapping means, that only features, which are referenced by the input program are considered relevant for creating output configurations, featureSelection "imitates" featureMapping by selecting randomly *n* features, which are considered relevant. The configuration options are introduced in Sections 2.6.1 and 2.6.1

the best assignment from a finite set of possible assignments for many variables [BPS99]. The idea of using optimization for configuration-production purposes is as follows: we want to formulate the "feature-assignment-problem" as a CSP. The "variables" are features, which can be disabled or enabled, so the "finite set of possible assignments" for the variables is $\{0, 1\}$, the optimization criterion is finding the configuration with the minimal/maximal number of enabled features, which satisfies the variability model. We will model the CSP as a constraint system with a weighted objective function, which is minimized by the solver in the end.

Since we have explained the in- and output parameters of the algorithm in the last section, we only need to introduce some naming conventions here: $setName_e$ be the subset of features, which are enabled, $setName_d$ be the subset of features, which are disabled and $setName_i$ be the subset of features, which are irrelevant. Furthermore features can be undefined. $setName_u$ be the subset of features, which are undefined.

Now, we will go through the optimization instantiation, which can be seen in Algorithm 2:

The algorithm starts with some declarations. U defines all undefined features, which are not enabled in IC but still relevant, I names the rest of the disabled features in IC, which are irrelevant according to the list R. A "delta valuation" is a data type for a temporary assignment of the features. In partitions the features in three groups (enabled, disabled, and irrelevant). DV defines the initial assignment of all features with respect to the result of the sampling algorithm, which is neither optimal nor correct with respect to the variability model or further constraints.

In the following steps, we formulate a constraint satisfaction problem: First, we convert VM in its Boolean formula representation bVm and model it as constraints in the constraint system cs. The relations between the features in the variability model define the constraints in cs. Then, generate the integer representation of every feature f in $DV_e \cup DV_d$ and determine that all features can only be assigned with 0 or 1. We add this as a constraint in cs. Thus, we create the finite set of possible assignments for the variables, which is needed for solving CSPs. Line 8 shows the core of the algorithm: at this point, we create an objective function t consisting of weighted features (more specific, the feature's integer representation), which are part of DV_e or DV_d. The weights are depending on the optimization goal. The solver always performs minimization. So if we want to maximize the number of enabled features we weight all of them with an equal negative integer (-1), so that enabling them improves the situation according to the optimization goal. If we want to minimize the number of enabled features we distinguish two cases:

1. the feature is disabled in the initial configuration DV. As we want to disable most of the features, we want the feature to stay disabled in the result configuration, except it needs to be enabled due to any further constraints (like the variability model). Hence, we weight it with an

Algorithm 2 : OptimizationInstantiationAlgorithm(VM, IC, R, min)

input : *VM*: variability model
IC: initial configuration, bipartite (enabled, undefined) set
R: set of relevant features
min: Boolean, true iff *#enabled features* should be minimized in the output config, maximized otherwise

output : *OC*: tripartite (enabled, disabled, irrelevant) set of features, which satisfies *VM* and has the minimal (or maximal, depending on *min*) *#enabled features*

begin

`// Initializations:`

1 $U \leftarrow (IC_u \cap R), I \leftarrow (IC_u \setminus R)$

2 $$DV \leftarrow \begin{cases} DV_e \leftarrow IC_e \\ DV_d \leftarrow U \\ DV_i \leftarrow I \end{cases}$$

`// Construct CSP:`

3 $bVM \leftarrow$ generateBooleanRep(VM)

4 $cs \leftarrow$ generateConstraintSys(bVm)

5 **for** $\forall$ *features* $f \in DV_e \cup DV_d$ **do**

6 $\bar{f} \leftarrow$ generateIntegerRep(f)

7 cs.addConstraint($\bar{f} \in \{0, 1\}$)

8 Create an objective function t for cs as:

$$t = \sum_{f \in DV_e \cup DV_d} \bar{f} * w_f \text{ where } w_f = \begin{cases} -1 \textbf{ if } \neg min \\ MAX\_INT \textbf{ if } f \in DV_d \\ 1 \text{ otherwise} \end{cases}$$

`// Create output configuration`

9 solver.minimize(t, cs)

10 $$OC \leftarrow \begin{cases} OC_e \leftarrow m.f_e \\ OC_d \leftarrow m.f_d \\ OC_i \leftarrow I \end{cases}$$

11 Return OC.

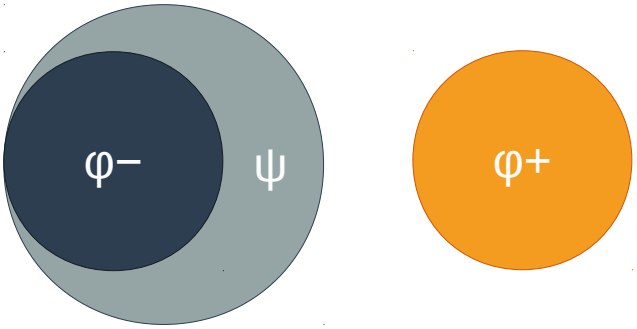


Fig. 3.1 Schematic representation of Craig interpolation

equal high positive integer (MAX_INT, id est, 2 147 483 647 in Java) in order to benefit if it stays switched off.

2. the feature is enabled in the initial configuration *DV*. Those features should stay enabled since otherwise we would discard the result of the sampling algorithm. Since we do not want that they are switched off, we give them a small weight, namely 1.

In the next step, the solver performs the optimization. Let $m.f$ be one specific feature f in a satisfiable model m for cs with respect to the solver. f_e is a feature, which is enabled, f_d is a disabled feature. We convert the solver's model, which satisfies all constraints and has the minimal solution for t into the result configuration *OC*. *OC* is returned and the algorithm terminates.

3.3 Craig Interpolation Instantiation Algorithm

Craig interpolation is a highly versatile technique, which has proven useful for many different purposes [Cra57; McM05; CHN12]. We use Craig interpolation as an approximation tool in the instantiation algorithm. In Figure 3.1, we see a schematic representation of Craig's interpolation theorem.

In our work, φ^- represents the constraints (the variability model) and φ^+ represents the initial assignment of the features, which is depending on the mode of `Simtopia`, and the partial configuration from the sampling algorithm. The initial assignment is not considering the variability model and therefore, is not necessarily valid. Hence, the solver checks if the assignment of the features in φ^+ contradicts the constraints in φ^-. If the assignment does not contradict the constraints we found the output configuration already, namely the assignment in φ^+.

If the assignment contradicts φ^- the solver creates the interpolant ψ, which tells us all features in φ^+ whose assignment is invalid with respect to the constraints in φ^-. So we change the assignment of those features in φ^+. Thus, φ^+ "moves closer" to φ^-. Then we start again checking if the "new" φ^+ and φ^- are satisfiable. We stop this process, when we find an assignment, which is satisfiable.

Algorithm 3 : CraigInterpolationInstantiationAlgorithm(VM, IC, R, min)

input : *VM*: variability model
IC: initial configuration, bipartite (enabled, undefined) set
R: set of relevant features
min: Boolean, true iff *#enabled features* should be minimized in the output config, maximized otherwise

output : *OC*: tripartite (enabled, disabled, irrelevant) set of features, which satisfies *VM* and has the minimal (or maximal, depending on *min*) *#enabled features*

begin

`// Initializations:`

1 $U \leftarrow (IC_u \cap R),\ I \leftarrow (IC_u \setminus R)$

2
$$IDV \leftarrow \begin{cases} IDV_e \leftarrow \begin{cases} \emptyset \textbf{ if } min \\ U \textbf{ else} \end{cases} \\ IDV_d \leftarrow \begin{cases} U; \textbf{if } min \\ \emptyset \textbf{ else} \end{cases} \\ IDV_i \leftarrow I \end{cases}$$

3 $W.push(IDV)$

4 **while** *W is not empty* **do**

`// Conversions:`

5 $BVM \leftarrow$ generateBooleanFormulaRep(VM)

6 $BIC \leftarrow$ generateBooleanFormulaRep(IC)

7 $prevDV \leftarrow W.getFirst()$

8
$$OC \leftarrow \begin{cases} OC_e \leftarrow (prevDV_e \cup (IC_e \setminus prevDV_d)) \setminus (prevDV_i \cup I) \\ OC_d \leftarrow prevDV_d \setminus (prevDV_i \cup I) \\ OC_i \leftarrow (prevDV_i \cup I) \end{cases}$$

`// Construct a Craig interpolation problem:`

9 $\varphi^- \leftarrow BVM,\ \varphi^+ \leftarrow$ setPhi+($prevDV$, OC)

10 $SAT \leftarrow$ solver.isSAT(φ^-, φ^+)

11 **if** $\neg\ SAT$ **then**

12 $itp \leftarrow$ solver.getITP(), $itpConfig \leftarrow$ generateFeatureSet(itp)

- Please note that the algorithm continues on the next page -

```
14          nextDV ← { nextDV_e ← (prevDV_e ∪ itpConfig_e) \ itpConfig_d
                     { nextDV_d ← (prevDV_d ∪ itpConfig_d) \ itpConfig_e
                     { nextDV_i ← I
15          checked ←isAlreadyChecked(nextDV)
16          if checked then
                        { nextDV_e ← prevDV_e \ itpConfig
17              nextDV ← { nextDV_d ← prevDV_d \ itpConfig
                        { nextDV_i ← prevDV_i ∪ itpConfig
18              W.delete(prevDV)
19              W.push(nextDV)
20      else
21          if OC_i ≠ ∅ then
22              BOC←generateBooleanFormulaRep(OC)
23              m = solver.getModel(BOC, BVM)
                      { OC_e ← m_e \ I
24              OC ← { OC_d ← m_d \ I
                      { OC_i ← I
25              W.delete(prevDV)
26  Return OC
```

As we now understand how to use Craig interpolation in our instantiation algorithm, we will take a closer look at the steps of the Craig interpolation instantiation in Algorithm 3.

Analogously to the naming conventions for the optimization instantiation algorithm, let $setName_e$ be the subset of features, which are enabled, let $setName_d$ be the subset of features, which are disabled, $setName_i$ the subset of features, which are irrelevant, and $setName_u$ the subset of features, which are undefined.

Again, the algorithm starts with some declarations. U defines all undefined features, which are not enabled in IC but still relevant, I represents the rest of the disabled features in IC, which are irrelevant according to the list R. IDV is the initial delta valuation, which defines the initial assignment of all features depending on the *min*. If the number of enabled features should be minimized all features are disabled initially and vice versa. We add IDV to a worklist W.

The following part is repeated until W is empty. We start with some conversions and initializations: BVM be the Boolean formula representation of VM and BIC for IC. *prevDV* is the first entry of W. Then we construct a provisional output configuration OC, which would be optimal with respect to IC and IDV.

Then we construct a Craig interpolation problem with φ^- being the Boolean representation of VM (BVM). φ^+ is generated from the delta valuation with the current feature assignment *prevDV* and OC (which additionally includes parts of IC for example). Now the solver checks if $\varphi^+ \wedge \varphi^-$ is satisfiable (SAT), namely if the assignment of the features in φ^+ does not contradict the variability model. If it is satisfiable we found a solution for the problem. In line 23, we get the assignment for the features of the solver's model m and can distribute the features in OC according to m. Then, we delete *prevDV* from W to end the loop and the return OC.

If φ^+ and φ^- is not satisfiable ($\neg$ SAT) the solver creates an interpolant *itp* and its binary feature representation *itpConfig*. According to Craig's interpolation theorem *itp* consists of all features, which occur in both, $\varphi+$ and $\varphi-$ (statement *(iii)* of theorem), but with different assignments (statement *(ii)* of theorem). So we build a new data valuation *nextDV* according to *itpConfig* in line 10. In line 11 we check if we already tested the same data valuation before to avoid infinite loops. If we did have the same delta valuation before, we create another delta valuation *nextDV*, which ignores all values in *itpConfig*, which lead to the contradiction of $\varphi+$ and $\varphi-$. In both cases, we now delete *prevDV* and push *nextDV* to W. As W is not empty here, we start again at line 4 of the algorithm.

4

Study

4.1 Research Questions

Our goal is the comparison of optimization and Craig interpolation for generating minimal and maximal configurations. As we have explained each algorithm in the previous chapter, we now want to compare both algorithms on several aspects to answer the following research questions:

- **RQ1.** Which instantiation algorithm is more efficient for different complexities (measured in number of features per variability model) of variability models?
- **RQ2.1** Which instantiation algorithm is more efficient for 1- or 2-wise sampling?
- **RQ2.2** Which instantiation algorithm is more efficient for random sampling?
- **RQ3.** Which instantiation algorithm is more efficient for different configuration modes (MAX DISABLED/MAX ENABLED)?

4.2 Operationalization

The basic structure of the study is as follows: for all experiments we have two corresponding runs, which use different instantiation algorithms but the same configuration of `Simtopia` apart from this. We measure the time of both runs and compare them. For designing our experiments in detail, we need to specify the values of the following terms (id est for the experiment variables), which are used in our research questions:

- For **RQ1.**: Which numbers of features do we use for testing different complexities?
- For **RQ2.**: How many configurations do we generate?

In the following paragraph, we shall answer these questions.

The first question concerns the complexity of the variability models. For our experiments we choose the values 10 and 100 for the number of features n to show the runtime differences for very small and bigger (parts of) variability models, which need to be considered by instantiation algorithm and

solver. There are multiple options how we can generate suitable test sets with exactly n features. We use the "featureSelection" option of `Simtopia`, which is discussed in Section 2.6.1 of the background chapter. In the following, we will reference this parameter by "# *features*" or "n considered features." We use the limited number of features as an indicator for the complexity of the problem, which has to be solved. A second indicator for complexity of variability models could be the number of clauses of a variability model. We decided not to run some extra experiments for different numbers of clauses, as we see that those two dimensions are not independent. Variability models with many features usually have many clauses and vice versa. That means we do not expect strongly differing results for both criteria. Still, we will mark the amount of clauses of the variability model per each experiment.

The second question concerns the number of generated configurations. For our studies we will use Random and T-wise sampling. Medeiros et al. found that "simple algorithms with small sample sets [...] are the most efficient in most contexts" [Med+16]. As the sample sets of T-wise-sampling are getting bigger for higher values of T, we have decided to use one- and pair-wise-sampling, id est $T = \{1, 2\}$. This assignment gives us a fixed number of returned configurations defined by the binomial coefficient $\binom{n}{k}$, for example $\binom{3}{2} = 3$ configurations for three features (defining n) with pairwise sampling (defining k), if we have no further constraints.

If we use random sampling (and do not have further constraints) our natural upper limit of configurations, which can be produced would be 2^n with n being the number of binary features. That means we can get eight different configurations for three features. As the number of possible configurations is exploding for bigger values of n we need an expedient upper bound b for the number of returned configurations. We make three experiments with $b = \{10, 100, 1000\}$.

Now, we have specified all variables and values, which we need for answering the research questions. Summing up, we get the following list of variables and values (where each line begins with the name of the variable and is followed by the values):

- the instantiation algorithm: {optimization, Craig interpolation}
- the sampling algorithm: {random, one-wise, pair-wise}
- the complexity of the variability model in features: $\{10, 100\}$
- the mode: {MAX ENABLED, MAX DISABLED}
- the variability model: {busybox, linux, ...} (80 models in total)

One test case consists of a combination of specific values, whereby every variable needs to be assigned with exactly one value. To answer **RQ3**, for example, we generate two experiments: one is including all test cases with MAX ENABLED mode, 100 features, and T-wise sampling with $T = 1$, the other

Experiment Variables			
RQ	Independent Variables	Dependent Variables	Controlled Variables
RQ 1	complexity of VM (# features = {10, 100}), instantiation algorithm	CPU-time	sampling algorithm (T-wise, T=1), mode (MAX DISABLED)
RQ 2.1	sampling algorithm (T-wise, T={1, 2}), instantiation algorithm	CPU-time	mode (MAX DISABLED), complexity of VM (# features = 100)
RQ 2.2	sampling algorithm (random, # configs = {10, 100, 1 000}), instantiation algorithm	CPU-time	mode (MAX DISABLED), complexity of VM (# features = 100)
RQ 3	mode (mode = {MAX ENABLED, MAX DISABLED }), instantiation algorithm	CPU-time	sampling algorithm (T-wise, T=1), complexity of VM (# features = 100)

Tab. 4.1 List of independent, dependent, and controlled variables

includes all test cases with MAX DISABLED mode, 100 features, and 1-wise sampling.

To ensure the validity of the result values, we categorized the variables in independent, dependent, and controlled variables as defined in Table 4.1. We use *VM* as a short term for variability model in the table. The first category (independent variables) includes the variables, which we deliberately determine in our experiments, the second category (dependent variables) determines the variables, which are depending on the changes in variables of the first category (this is the variable we use for measuring), and the third category (controlled variables) are the variables, which are fixed to not influence the result (id est the CPU time) in uncontrollable ways.

Our experiments are designed on the basis of this table.

4.3 Evaluation Environment

For our study we use the framework `Simtopia` and the `Z3` SMT Solver.

Although `Simtopia` can use several SMT solvers, we use the `Z3` Solver for our study because of the plurality of its features. As we have explained in the background chapter, `Z3` can both solve CSPs and problems in CNF syntax and therefore solve the problems created by both instantiation algorithms. That means we have only one solver for two types of problems, which makes our results more comparable.

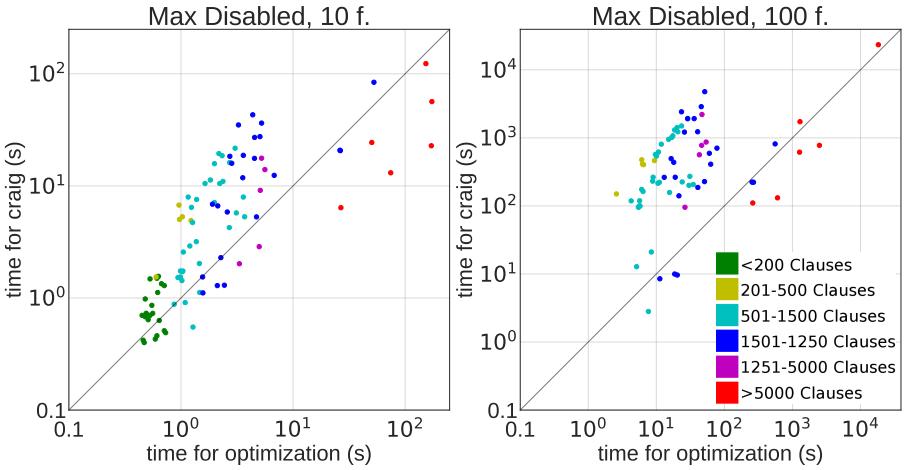
The evaluation is performed on a cluster consisting of eight machines with an i7-4770 processor with 4 CPU cores and 32 GB RAM each. The operating

system is Ubuntu. We use only 2 CPU cores, limit the memory usage per run to 5 000 MB and the execution time to 80 000 s.

To make our results more reliable we use two sorts of time measurement: The first one is used for the actual results of the study, the second one to review the actual results. The first time measurement is generated by the internal time stamp of Java, where we measure from the start of the instantiation algorithm until the delivery of the full valid configurations. The second one produced by the software `BenchExec`, which describes itself as "a framework for reliable benchmarking and resource measurement" [BLW15]. As `BenchExec` can only measure the time for a whole run, we run `Simtopia` twice: the first time we stop before the configuration production, the second time we stop after. Thus, we get two separate measurements, which we subtract. Hence, we measure the same time period as we measure using the first method.

4.4 Performance Evaluation

In the following, we analyze the performance of the algorithms for the different experiments. For our evaluation, we dropped all runs, which did not pass without faults. These are all runs, which stopped with a timeout (which was a marginal number) or a segmentation fault (which are caused by internal errors of the `Z3` solver).

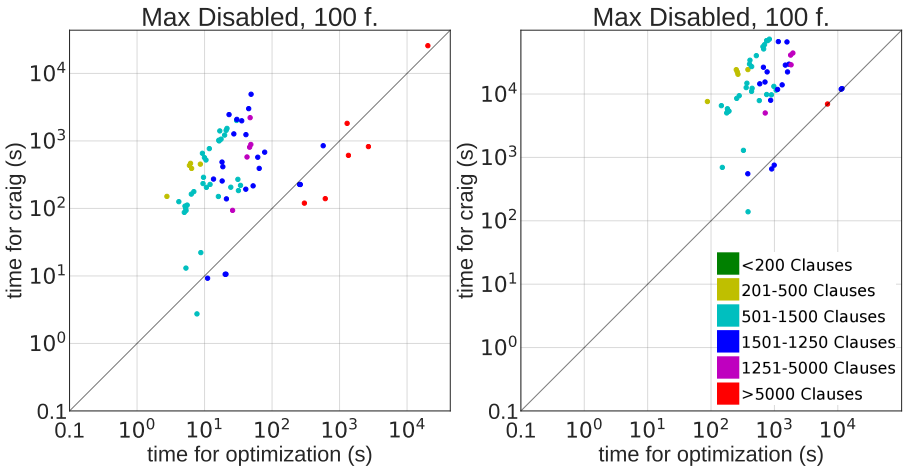


(a) CPU-time for little complexities of variability models (10 considered features)

(b) CPU-time for higher complexities of variability models (100 considered features)

Fig. 4.1 RQ1

In general, we use plots with logarithmic time axes. The y-axis represents the CPU-time in seconds which was needed for the same input of `Simtopia` by the Craig interpolation instantiation algorithm, the x-axis shows the CPU-time in seconds of the optimization instantiation algorithm. That means, one point is defined by the time that both instantiation algorithms needed



(a) CPU-time for 1-wise sampling **(b)** CPU-time for 2-wise sampling

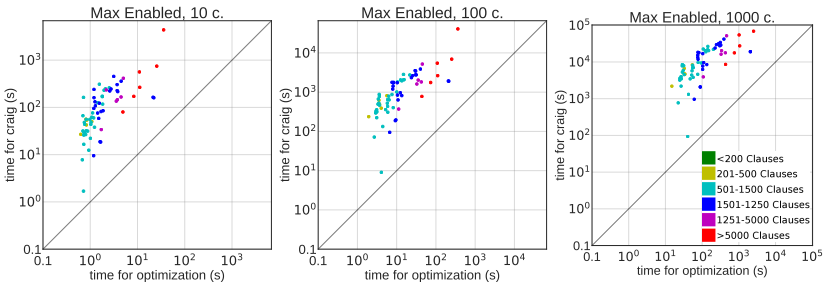
Fig. 4.2 RQ2.1

for a certain problem. The grey line in the middle helps us to identify the distribution of the problems, for which the Craig interpolation/the optimization instantiation algorithm is faster. The color of the points is defined by the number of clauses of the processed variability models. We can see the allocation of colors and the ranges of the number of clauses, for example, in Figure 4.1b.

In order to answer **RQ1**, we filter our test cases according to Table 4.1, id est we include all runs with MAX DISABLED mode, sampling algorithm is 1-wise, and # selected features is 10 in Figure 4.1a and 100 in 4.1b in our evaluation figure (Figure 4.1).

As we can see easily, the performance of optimization instantiation is better in most cases. It is faster for ca. 66% of all successful runs for 10 considered features. It performs even better for 100 features where it is faster in about 75% of the test cases. In general, all points are close together and ordered by the number of clauses (more ore less). But yet, the distribution of the red points is conspicuous: almost every run with a very complex variability model (with more than 5 000 clauses to consider) can be done faster with the Craig interpolation instantiation algorithm. Overall, we note that the complexity of variability models (measured in the number of considered features) has less influence on the runtime than the total number of clauses of the variability model.

As shown in Table 4.1, the second performance evaluation includes all runs with MAX DISABLED mode, 100 considered features, and 1- or 2-wise sampling. Using Figure 4.2 we can determine if one instantiation algorithm is faster in combination with T-wise sampling algorithms (**RQ2.1**). Figure 4.2a shows the results for the runs using 1-wise sampling, Figure 4.2b shows the results for runs with 2-wise sampling. Again, we can see that optimization performs better in most cases.



(a) CPU-time using Random Sampling (10 Candidates)

(b) CPU-time using Random Sampling (100 Candidates)

(c) CPU-time using Random Sampling (1 000 Candidates)

Fig. 4.3 RQ2.2

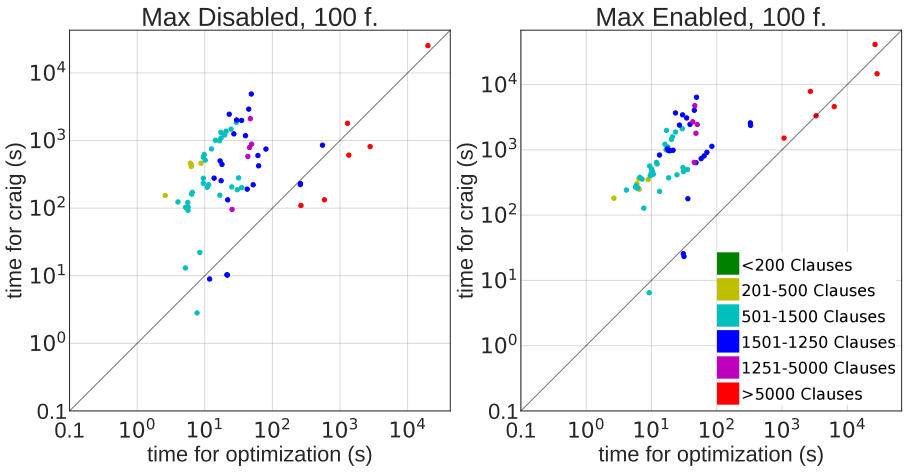
Unsurprisingly, the figure on the left looks very similar to Figure 4.1b. If we compare the experiments we will notice that the input values are the same both times.

The main difference between both experiments is the growing number of configurations, which is produced by the sampling algorithm for growing T. The problems, which have to be expressed by the instantiation algorithm are very similar for different T. By comparing both plots, we can see that Craig interpolation instantiation scales poorly for T-wise sampling. That means, if Craig interpolation instantiation is slower for 1-wise sampling, the performance difference will be even higher for 2-wise sampling. We can see this phenomenon in Figure 4.2b, as the "time-gap" between the instantiation algorithms is raising.

The results for the instantiation algorithms in combination with random sampling are really similar for each experiment. We use the experiments to answer **RQ2.2**. The main difference between the three experiments is the number of configurations, which are created by random sampling: in Figure 4.3a we see the time measurements for 10 configurations, Figure 4.3b shows the results for 100 configurations, Figure 4.3c shows the results for 1 000 initial configurations. We consider 100 features to create those configurations. The mode for the experiments is MAX DISABLED.

As a result of the growing numbers of configurations the total time, which is needed by the instantiation algorithms, is growing. However, unlike in Figure 4.2, the time differences between the different instantiation algorithms are not growing for different experiments. That means, that the Craig interpolation instantiation algorithm scales better for the configurations, which are produced by random sampling (compared to the experiments with T-wise sampling). One more difference is, that Craig interpolation instantiation can not perform better for a special type of variability models with respect to their number of clauses.

In Figure 4.4 we can see a comparison of the modes MAX DISABLED (Figure 4.4a) and MAX ENABLED (Figure 4.4b), which we use to answer **RQ3**. The



(a) CPU-time for MAX DISABLED mode **(b)** CPU-time for MAX ENABLED mode

Fig. 4.4 RQ3

number of considered features is 100. We use 1-wise sampling. The first experiment (shown in Figure 4.4a) is equivalent to the one in Figure 4.1b.

Like in all the other experiments, optimization instantiation performs better for most test cases. Furthermore, we see that the MAX ENABLED mode "tightens" the time values. The main part of the runs needed a similar time for the same instantiation algorithm, with the exception of the red points. Those red points represent very complex variability models with more than 5 000 clauses. We can not really explain this distribution without looking at the heuristics of the Z3 solver. As an analysis of the heuristics of Z3 is not in the scope of this work, we can just provide our observations here.

5 Discussion

In this chapter we want to discuss the results of our performance evaluation.

5.1 Valuation of the Results

In our work, we compared the performance of two different algorithms (based on Craig interpolation and optimization) for generating minimal and maximal configurations with respect to the variability model. In the last chapter we have presented several experiments, which test the algorithms in combination with different preceding sampling algorithms or under specific conditions, like different complexities of variability models.

In general, we have seen that both instantiation algorithms can solve the problem of generating minimal or maximal configurations. On average, the time is raising for both algorithms with growing numbers of configurations to create. As we can see in Figure 4.1a, both algorithms are able to create 10 configurations in less than one second at best. Using pair-wise sampling, the algorithms need to process about 4 950 initial configurations (given by the binomial coefficient n choose k with $n = 100$ and $k = 2$). As shown in Figure 4.2b, this took both instantiation algorithms about 0.25 h at least. However, there are big differences for the upper time-bound, which is needed: the Craig interpolation instantiation algorithm needs more than one day (as 10^5 s $\approx$ 27.778 hours) to process all configurations while the optimization instantiation algorithm needs about 2.8 h at most.

The answers for the research questions are as follows:

- **A.RQ1.** Optimization instantiation performs better for most of the test cases. However, there is some indication that creating a Craig interpolant is efficient for a special type of problems: for test cases whose variability model contains more than 5 000 clauses, in MAX DISABLED mode, and with initial configurations generated by T-wise sampling, Craig interpolation instantiation is at least as fast as optimization.
- **A.RQ2.-3.** Overall, we have seen that the optimization instantiation algorithm performs better for most cases in all experiments. This indicates that solving a CSP can be done more efficiently than creating an interpolant by the Z3 solver.

The results, which we have summarized above depend on some more environmental influences, which are discussed in the following section.

5.2 Validity

For our study, we strove general validity. For instance, to achieve better intern and extern validity we eliminated random (and therefore incontrollable) decisions by using a seed. By using the internal time measurement of Java and the benchmarking tool `BenchExec` in addition, we enhance the significance of the time measurements. Furthermore, we set the JVM flag "-Xcomp". This prevents the JIT compiler to perform different efficiency-optimizations on different test cases and therefore makes the results more comparable. We determined controlled variables (as we can see in Table 4.1), to ensure that only the desired variables of `Simtopia` influence the CPU time. Using the `Z3` solver, which can solve both types of problems (CSPs and creating an interpolant), helps to improve the comparability of the time measurements even more.

Still, there are some threats to validity, which we discuss in the following section.

5.2.1 Threats to Validity

As explained before, using the `Z3` solver has the advantage of making the time measurements more comparable. However, it has some disadvantages concerning internal and external validity: first, `Z3` causes many segmentation faults due to some internal errors. As the number of test cases stopping with segmentation faults constituted up to 20%, the number of segmentation faults had a major influence on the number of successfully executed runs. This has an negative influence on the internal validity of the experiments. Second, using `Z3` has a massive impact on the external validity: as all solvers use different heuristics to compute their results, we can not really tell if the time-distribution would be different if we would have used a different solver. Maybe, there are other solvers where Craig interpolation instantiation would have been faster for most test cases.

Another, but less serious threat on external validity is one detail of the implementation of `Simtopia`: the framework does not use covering arrays (which were introduced in this section of the background). That means, we produce more configurations than may be useful in practice. If it takes one instantiation algorithm a long time to produce a configuration, which is not needed in the end, this may influence the total runtime for a test case. We can not make any assumptions how the runtime would change if the sampling algorithm would use covering arrays for creating initial configurations.

5.3 Future Work

To counteract some threats to validity on the one hand, and get more generalizable results on the other hand, we discuss some possible improvements in this section.

At first, we would suggest to make more experiments to explore the trend that Craig interpolation instantiation is faster for especially complex variabil-

ity models with more than 5 000 clauses in some experiments. As we do not have significant numbers of such complex variability models, we first need to validate that there is any trend in fact. After that, making some specific experiments to reproduce this effect using more test cases and maybe find some more similarities between those problems can possibly even open up new fields for using the Craig interpolant.

Second, using different SAT solvers (like MathSAT5) or integrating a specialized CSP solver can improve the external validity of our experiments and generate interesting results concerning the runtime differences for the same problem for different solvers.

Also, performing a detailed runtime analysis on the instantiation algorithms can be useful in order to get a better performance for both algorithms. Eliminating useless calls or operations can improve (id est shorten) the time, which is needed and increase the correctness of the time measurement.

6

Conclusion

In this thesis, we introduced two algorithms for generating minimal and maximal configurations, namely optimization instantiation and Craig interpolation instantiation. We implemented them in the framework `Simtopia`. As both algorithms link the sampling algorithms and the SMT solver in different ways, the choice of the instantiation algorithm leads to differences in the times, which is needed to create the same set of configurations.

The evaluation and discussion of these differences indicates that Craig interpolation instantiation may have some specific strengths in processing very complex variability models (measured in the number of clauses). However, we found that optimization instantiation is the preferable algorithm for the purpose of generating minimal and maximal configurations using the `Z3` solver.

Appendices

7

Bibliography

[Ape+13] Sven Apel, Don S. Batory, Christian Kästner, and Gunter Saake. "Feature-Oriented Software Product Lines - Concepts and Implementation". Springer, 2013. ISBN: 978-3-642-37520-0 (*cited on pp. 1, 3, 4*).

[Bar+09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. "Satisfiability Modulo Theories". In: *Handbook of Satisfiability*. 2009, pp. 825–885 (*cited on p. 5*).

[Bar77] Jon Barwise. "Handbook of Mathematical Logic". North-Holland, 1977 (*cited on p. 5*).

[BLW15] Dirk Beyer, Stefan Löwe, and Philipp Wendler. "Benchmarking and Resource Measurement". In: *Model Checking Software - 22nd International Symposium, SPIN 2015, Stellenbosch, South Africa, August 24-26, 2015, Proceedings*. 2015, pp. 160–178 (*cited on p. 24*).

[BPS99] Sally C. Brailsford, Chris N. Potts, and Barbara M. Smith. "Constraint satisfaction problems: Algorithms and applications". In: *European Journal of Operational Research* 119.3 (1999), pp. 557–581 (*cited on pp. 6, 15*).

[CHN12] Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. "SMTInterpol: An Interpolating SMT Solver". In: *Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings*. 2012, pp. 248–254 (*cited on pp. 5, 6, 10, 17*).

[Cim+13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. "The MathSAT5 SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*. 2013, pp. 93–107 (*cited on pp. 5, 10*).

[Coo71] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. 1971, pp. 151–158 (*cited on p. 4*).

[Cra57] William Craig. "Linear Reasoning. A New Form of the Herbrand-Gentzen Theorem". In: *J. Symb. Log.* 22.3 (1957), pp. 250–268 (*cited on pp. 5, 17*).

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.* 1972, pp. 85–103 (*cited on p. 4*).

[KFB16] Egor George Karpenkov, Karlheinz Friedberger, and Dirk Beyer. "JavaSMT: A Unified Interface for SMT Solvers in Java". In: *Verified Software. Theories, Tools, and Experiments - 8th International Conference, VSTTE 2016, Toronto, ON, Canada, July 17-18, 2016, Revised Selected Papers.* 2016, pp. 139–148 (*cited on p. 10*).

[Kro67] M. R. Krom. "The Decision Problem for a Class of FirstOrder Formulas in Which All Disjunctions Are Binary". In: *Mathematical Logic Quarterly* 13.12 (1967), pp. 15–20 (*cited on p. 4*).

[McM05] Kenneth L. McMillan. "Applications of Craig Interpolants in Model Checking". In: *Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings.* 2005, pp. 1–12 (*cited on pp. 6, 17*).

[Med+16] Flávio Medeiros, Christian Kästner, Márcio Ribeiro, Rohit Gheyi, and Sven Apel. "A comparison of 10 sampling algorithms for configurable systems". In: *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016.* 2016, pp. 643–654 (*cited on pp. 1, 6, 7, 8, 22*).

[MB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings.* 2008, pp. 337–340 (*cited on pp. 5, 10*).

[Pre09] Steven David Prestwich. "CNF Encodings". In: *Handbook of Satisfiability.* Ed. by Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Vol. 185. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, pp. 75–97. ISBN: 978-1-58603-929-5 (*cited on p. 4*).

[RT06] Silvio Ranise and Cesare Tinelli. "The smt-lib standard: Version 1.2". Tech. rep. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at www. SMT-LIB. org, 2006 (*cited on p. 10*).

[Rüm17] Philipp Rümmer. 2017. URL: `http://www.philipp.ruemmer.org/princess.shtml` (visited on 2017-09-26) (*cited on p. 10*).

[93] "Satisfiability Suggested Format". Tech. rep. 1993 (*cited on p. 4*).

[Sebo7] Roberto Sebastiani. “Lazy Satisability Modulo Theories”. In: *JSAT* 3.3-4 (2007), pp. 141–224 (*cited on p. 5*).

Eidesstattliche Erklärung:

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

Passau, 30. Oktober 2017 _____

Elisabeth Griebl