The present work was submitted to the LuFG Theory of Hybrid Systems

MASTER OF SCIENCE THESIS

PARAMETER SYNTHESIS FOR ALGEBRAIC PROBLEMS WITH A BOOLEAN STRUCTURE

Nicolai Radke

Examiners:

Prof. Dr. Erika Ábrahám
Prof. Dr. Thomas Noll

Aachen, 17.03.2022

Abstract

The goal of regular satisfiability modulo theories (SMT) solving is to determine whether a logical formula is satisfiable or not. However, for industry purposes or to get a deeper understanding of the formula, it is helpful to look at larger parts of the parameter space. Parameter synthesis is concerned with finding regions of the parameter space satisfying the formula and regions which do not. Using SMT solving, it is possible to address the parameter synthesis problem with relatively simple algorithms. Such an algorithm is used by the tool PaSyPy [Wie21], which performs parameter synthesis for quantifier-free, real arithmetic. In this thesis, we develop the tool and its algorithms further by introducing sophisticated heuristics to improve running time.

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Nicolai Radke
Aachen, 17.03.2022

Acknowledgements

I am grateful for having had the opportunity to spend time on this topic and at the chair. I learned a lot and really enjoyed the time. Therefore, I want to first thank Prof. Dr. Erika Ábrahám for the opportunity to write this thesis at her chair. She always found time for discussions and was very supportive throughout the process. Further, I thank Prof. Dr. Thomas Noll for being the second examiner. I also want to thank Jasper Nalbach for the discussions and his technical support. Lastly, I want to thank Jonathan Müller for sharing his incredible C++ expertise with me.

Contents

1 Introduction **9**

2 Preliminaries **11**
2.1 SMT Solving . . . 11
2.2 Parameter Synthesis . . . 14

3 Parameter Synthesis **17**
3.1 Base Algorithm . . . 17
3.2 Sampling . . . 22
3.3 Splitting . . . 24
3.4 Incremental Solving . . . 25
3.5 Resulting Features . . . 26
3.6 Handling Quantifiers . . . 26
3.7 GUI Projection . . . 30

4 Implementation **33**
4.1 Overview . . . 33
4.2 Logical Encoding . . . 34
4.3 CLI . . . 35
4.4 GUI . . . 37

5 Experimental Evaluation **41**
5.1 Setup . . . 41
5.2 Strategy . . . 41
5.3 Experimental Results and Running Time Analysis . . . 44

6 Conclusion **53**
6.1 Summary . . . 53
6.2 Discussion . . . 53
6.3 Future Work . . . 54

Bibliography **55**

Chapter 1

Introduction

Satisfiability modulo theories (SMT) is a technique to check the satisfiability of logical formulas over different theories. One theory that is particularly important because of its real world applications is quantifier-free *nonlinear real arithmetic* (NRA). While NRA solvers can determine whether a formula is satisfiable or unsatisfiable, their goal is not to find larger sets of satisfying or unsatisfying assignments. This problem is covered by a process called *parameter synthesis*.

Given an initial parameter space, in NRA this could be a cross product of intervals as domain for the (free) variables, parameter synthesis tries to classify as large areas as possible of this parameter space as satisfying or unsatisfying. The formula $\varphi(x,y) = x + y \leq 2$, for example, is satisfied by all values within $[0,1] \times [0,1]$.

Related Work There have been different attempts to tackle the parameter synthesis problem in different contexts. For example, in the context of verification, there exist tools based on interval arithmetic. PROPhESY is a tool for parameter synthesis on parametric Markov chains [Deh+15]. Although also further work on PROPhESY by Junges et al. has parallels to algorithms and heuristics presented in this thesis, PROPhESY is not directly comparable due to the specific domain [Jun+19]. To the best of our knowledge, these previous approaches are only capable of handling conjunctions of constraints. However, in other applications, using conjunctions only is not expressive enough. It is therefore necessary, to address the parameter synthesis problem for arbitrary Boolean combinations of constraints. Wiegel has presented and implemented PaSyPy, a parameter synthesis tool tackling the parameter synthesis problem for arbitrary Boolean combinations of constraints [Wie21]. This thesis takes PaSyPy as a basis to improve it. This process of improvement includes eliminating design flaws and developing heuristics to reduce running time.

Contributions One issue with PaSyPy is that it does not use exact arithmetic. A focus of this thesis is therefore to switch to exact arithmetic. Because this implies rewriting significant parts of the code anyway, this thesis uses C++ as programming language to improve performance by a complete reimplementation of the Python-based PaSyPy tool.

PaSyPy uses the Z3 solver for SMT solving [MB08]. Since the number of solver calls correlates strongly with the general running time, reducing this number is a main

objective of this thesis. To achieve this goal, we develop sampling heuristics, splitting heuristics and other methods.

Furthermore, we implemented a GUI similar to the one in PaSyPy to visualize the results of a parameter synthesis.

The contributions of this thesis include

- a complete reimplementation of the Python-based PaSyPy tool in C++ and a switch from floating-point to exact arithmetic,
- improving scalability through advanced heuristics,
- extending previous algorithms by the capability of handling formulas with quantifier alternation, and
- analyzing the computational effort.

Structure In the following Chapter 2, we provide the background necessary to understand the concepts of this thesis. In Chapter 3, we then explain the parameter synthesis algorithm developed by Wiegel in [Wie21] as well as multiple novel approaches to reduce running time. Afterwards, in the same chapter, we suggest an approach to handle quantifiers. In Chapter 4, we present the features implemented during this thesis, which we then evaluate experimentally in Chapter 5. In Chapter 6, we finally conclude this thesis through summarizing and discussing the work done and giving an outlook on future work.

Chapter 2

Preliminaries

In this chapter, we provide definitions and explanations of concepts used in this thesis. At the basis of the parameter synthesis problem lies satisfiability modulo theories (SMT) solving, which is introduced first.

2.1 SMT Solving

SMT solving deals with satisfiability checking for formulas from extensions of propositional logic with theories. Therefore, we first give a short introduction to propositional logic and first order logic, adapted from a 2020 lecture by Ábrahám [Ábr20].

Propositional Logic

The atoms of propositional logic are propositions that can be either true or false. Consequently the domain of propositions is $\mathbb{B} = \{0,1\}$. An interpretation or assignment $\alpha : AP \to \mathbb{B}$ assigns Boolean values (elements of the domain) to a set AP of (atomic) propositions, also called variables. We assume in the following a fixed proposition set AP and use *Assign* to denote the set of all assignments.

Definition 2.1.1 (Propositional Logic)**.**

Syntax *Let* $p \in AP$.

$$\varphi := p \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \tag{2.1}$$

We define PropForm to be the set of all propositional formulas.

Semantics *We introduce the satisfaction relation* $\models \subseteq Assign \times PropForm$ *and say that* α *satisfies* φ *iff* $(\alpha, \varphi) \in \models$. *We use* $\alpha \models \varphi$ *equivalently. We define* $\models$ *recursively:*

$$\begin{array}{lll} \alpha \models p & \textit{iff} & \alpha(p) = \textit{true} \\ \alpha \models \neg\varphi & \textit{iff} & \alpha \not\models \varphi \\ \alpha \models \varphi' \wedge \varphi'' & \textit{iff} & \alpha \models \varphi' \,\textit{and}\, \alpha \models \varphi'' \end{array}$$

As syntactic sugar, we define $\varphi' \vee \varphi'' := \neg((\neg\varphi') \wedge (\neg\varphi''))$ as well as $(\varphi' \to \varphi'' := (\neg\varphi') \vee \varphi'')$. We may omit parentheses if it is possible to restore them through operator binding. $\neg, \wedge, \vee$ and $\to$ bind decreasingly in this order. If a formula φ is satisfiable, then we say "φ is SAT" and otherwise that "φ is UNSAT".

FO-Logic and NRA

For some purposes, propositional logic may not be expressive enough. Therefore, we extend it in first-order logic (FO) with theories. FO is not a fixed logic but rather a framework. Dependent on the specific elements put into the framework, the resulting logic can take different forms. There are different syntactical constructs in FO.

Theory symbols: constants, variables, function symbols

Predicate symbols: lift the theory to the logical level

Logical symbols: logical connectives and quantifiers

While the logical symbols are fixed, varying the other two gives different FO instances. In this thesis, we are only using the one specific FO instance of non-linear real arithmetic (NRA). Using NRA as an example, we now show how to build a specific FO logic.

Theory Symbols Theory symbols are constants, variables and function symbols.

NRA

Constants: 0,1

Variables: $x, y, z, ...$

Function symbols: $+, \cdot$ as binary function symbols

For constants c, variables v and an n-ary function symbol f, we define a *term* t inductively by

$$t = c \mid v \mid f(t, ..., t).$$

NRA '0', '1', or '$(1 + 1) \cdot 0$' are examples for terms.

Predicate Symbols Predicate symbols lift terms from the theory to the logical level.

NRA Binary '$<$' is the only necessary predicate symbol in NRA. However, '$\geq$','$>$', '$=$', '$\leq$', '$\neq$' could be added as syntactic sugar.

We define a *constraint* inductively by the following rule

- If P is an n-ary predicate symbol and $t_1, ..., t_n$ are terms, then $P(t_1, ..., t_n)$ is a constraint.

NRA '$1 \cdot 1 < 0$' is a constraint.

Logical Symbols As mentioned above, the logical symbols are the same for all FO logics. The logical connectives are exactly the ones we already know from propositional logic in Definition 2.1.1, namely $\neg$, $\wedge$, and ,if needed, $\vee, \rightarrow, ...$. Additionally, FO introduces quantifiers $\exists$ and $\forall$.

Inductive Definition of FO Syntax We are now ready to define an FO formula φ inductively. Let c be a constraint and x a variable. Similar to Definition 2.1.1, we define

$$\varphi = c \mid (\neg\varphi) \mid (\varphi \wedge \varphi) \mid (\forall x \varphi) \mid (\exists x \varphi). \tag{2.2}$$

In a formula φ, we call a variable x free, if it is not bound by any quantifier. We say that a formula φ is in *prenex normal form (PNF)* iff it has the form

$$\varphi = Q_1 x_1 ... Q_n x_n \varphi'(x_1, ..., x_n) \tag{2.3}$$

with quantifiers $Q_1, ..., Q_n \in \{\forall, \exists\}$ where $\varphi'(x_1, ..., x_n)$ quantifier free. We then call $Q_1 x_1 ... Q_n x_n$ the prefix of φ and $\varphi'(x_1, ..., x_n)$ its matrix. Every formula φ has an equivalent PNF formula $\hat{\varphi}$ [Grä21].

NRA An example for a valid formula is $\varphi = \forall x((x \geq 0) \rightarrow \exists y(y \cdot y = x))$. It is equivalent to the PNF formula $\hat{\varphi} = \forall x \exists y((x \geq 0) \rightarrow (y \cdot y = x))$, which has prefix $\forall x \exists y$ and matrix $((x \geq 0) \rightarrow (y \cdot y = x))$.

We can fix the set of non-logical symbols using a *signature* Σ.

NRA The signature of NRA is $\Sigma = (0{,}1, +\,, \cdot\,, <)$.

Semantics In order to give meaning to formulas, we define Σ-structures, for a given signature Σ. It is given by

- a domain D and
- an interpretation I of the non-logical symbols in Σ that maps each
  - constant symbol to a domain element,
  - n-ary function symbol to a function of type $D_n \rightarrow D$,
  - n-ary predicate symbol to a predicate of type $D_n \rightarrow \{0, 1\}$.

For free variables we additionally need an assignment α that maps each free variable to a domain element. Using the structure, we are now able to interpret terms. The satisfiability relation is defined equivalently to the one for predicate logic in Definition 2.1.1.

NRA In NRA, the domain is $\mathbb{R}$, and constants 0 and 1 are interpreted with the respective numbers. $+, \cdot$ and $<$ are interpreted as addition, multiplication and strictly less than, respectively.

Fragment: Quantifier-Free Formulas If we do not allow quantifiers, we get the quantifier-free fragment of NRA, short QF-NRA.

SMT and SMT-Solving

First order logics, as we have seen above, always require an underlying logic; in this thesis, real arithmetic. The field concerning these logics, their decidability, algorithms to find solutions of formulas and related problems is called *Satisfiability Modulo Theories (SMT)*, indicating its relation to solving formulas in predicate logic (SAT-solving).

There exist multiple SMT solvers for QF-NRA whith sophisticated solving strategies and features. One of these features is called incremental solving. Assume a solver is checking a formula φ for satisfiability. To come to a solution, solvers deduce logical implications called context. When the solver is then given an additional constraint ψ this context may still be useful when finding a solution for $\varphi \wedge \psi$. Many solvers therefore have an option to save the context in order to reuse it later. This technique is called incremental solving.

To the best of our knowledge, there is currently no SMT solver capable of handling arbitrarily quantified NRA formulas. However, there are SMT solvers capable of handling arbitrary quantifier-free NRA formulas. Generally, given a formula containing free variables, a solver will return "satisfiable" iff there exist values for the free variables such that the formula evaluates to true. Free variables are thus implicitly existentially quantified.

2.2 Parameter Synthesis

This section defines parameter synthesis – the problem this thesis addresses – and related terms.

Orthotopes

An orthotope or hyperrectangle is the generalization of a rectangle to arbitrary dimensions. Formally this means B is an orthotope iff it is a cartesian product of intervals. In this thesis, we only use closed intervals and thus closed orthotopes in $\mathbb{R}$, meaning

$$B = [l_1, u_1] \times ... \times [l_n, u_n] \subseteq \mathbb{R}^n.$$

For simplicity, we use the term *box* as a synonym for orthotope. If a box B is split into other boxes, we call each resulting box B_c a child box of B. We further introduce B as a formula with $B(x) := x \in B$. This means $B(x)$ holds iff x lies in B.

Defintions

Definition 2.2.1 (Parameter Synthesis Problem).

Input: – *an NRA formula* $\varphi(x_1, ..., x_n)$ *and*
– *an initial box* $B_{init} = [l_1, u_1] \times ... \times [l_n, u_n] \subseteq \mathbb{R}^n$.

Goal: – *to find* S_+, S_- *with* $S_+ \uplus S_- = B_{init}$ *such that* $\forall x(x \in S_+ \rightarrow \varphi(x))$ *and* $\forall x(x \in S_- \rightarrow \neg\varphi(x))$.

It is not immediately clear how S_+ and S_- should be specified. By definition, both are already determined by φ. However, the goal is to have a representation that helps understanding φ. The representation we chose in this thesis are boxes. For humans

it is immediately clear whether a point lies in a box or not. Additionally, depicting boxes is very easy if they have only one or two dimensions.

Having chosen boxes as an appropriate form of representation, it is obvious that the problem defined above in Definition 2.2.1 is generally not solvable: S_+ and S_- are then unions of boxes, but the boundaries between the satisfying and unsatisfying regions are not always axis parallel. Thus, we define the relaxed parameter synthesis problem.

Definition 2.2.2 (Relaxed Parameter Synthesis Problem)**.**

Input:
- *an NRA formula* $\varphi(x_1, ..., x_n)$ *and*
- *an initial box* $B_{init} = [l_1, u_1] \times ... \times [l_n, u_n] \subseteq \mathbb{R}^n$.

Goal:
- *to find* S_+, S_- *with* $S_+ \subseteq B_{init}, S_- \subseteq B_{init}$ *such that* $\forall x(x \in S_+ \to \varphi(x))$ *and* $\forall x(x \in S_- \to \neg\varphi(x))$, *and*
- *to rate a solution* S'_+, S'_- *"better" iff the volume of their union* $S'_+ \cup S'_-$ *is larger than the one of* $S_+ \cup S_-$.

For simplicity, we sometimes write $B \in S_+$ instead of $B \subseteq S_+$ for a box B. We do the same for S_-.

Additionally, we introduce naming conventions.

Definition 2.2.3 (Satisfying Box)**.** *We call a box B satisfying (or safe) iff* $\forall x(B(x) \to \varphi(x))$.

Definition 2.2.4 (Unsatisfying Box)**.** *We call a box B unsatisfying (or unsafe) iff* $\forall x(B(x) \to \neg\varphi(x))$.

Under consideration of Definitions 2.2.3 and 2.2.4, the relaxed parameter synthesis problem (Definition 2.2.2) could be summarized as "splitting the initial box in satisfying and unsatisfying boxes".

Chapter 3

Parameter Synthesis

This chapter describes the theoretical work of this thesis, which aims to tackle the relaxed parameter synthesis problem from Definition 2.2.2. First, in Section 3.1, we describe the base algorithm as developed by Wiegel [Wie21]. In Sections 3.2 and 3.3, we then focus on accelerating the base algorithm through integrating advanced splitting and sampling heuristics, respectively. In Section 3.4 we propose an idea on how to use incremental solving to reduce running times before we give a comprehensive list of all implemented features in Section 3.5. In Section 3.6, we go into the details of quantifier handling. Finally, in Section 3.7, we explain the projection of a solution to only two dimensions, which is necessary for the GUI.

3.1 Base Algorithm

This section describes the base algorithm as developed by Wiegel [Wie21]. Algorithm 3.1 shows its pseudocode, Figure 3.1 an equivalent flow-chart diagram. Below the algorithm description, we show an example to help understanding.

The algorithm's goal is to address the relaxed parameter synthesis problem from Definition 2.2.2: Given an initial box B_{init}, split this box iteratively into satisfying and unsatisfying boxes according to Definitions 2.2.3 and 2.2.4.

Definitions 2.2.3 and 2.2.4 contain universal quantifiers. However, as described in Section 2.1, SMT solvers are currently not able to handle these. To compensate for this shortcoming, we state the following theorem.

Theorem 3.1.1. $\forall x(B(x) \to \varphi(x)) \equiv B(x) \land \neg\varphi(x)$ *is UNSAT.*

Proof. Using the implicit existential quantification of free variables by solvers as described in Section 2.1 for the step labeled with $(*)$, we get

$$\begin{aligned} \forall x(B(x) \to \varphi(x)) &\equiv \neg\neg\forall x\ (B(x) \to \varphi(x)) \\ &\equiv \neg\exists x\ \neg(\neg B(x) \lor \varphi(x)) \\ &\equiv \neg\exists x\ (B(x) \land \neg\varphi(x)) \\ &\overset{(*)}{\equiv} B(x) \land \neg\varphi(x) \text{ is UNSAT.} \end{aligned}$$

□

Analogously we formulate a theorem for unsatisfying boxes.

```
1  synthesize(B_init, φ)
2  {
3      // initialize structures
4      B_init.depth := 0
5      Q_? := (B_init);
6      S_+ := ∅;
7      S_− := ∅;
8
9      // while unknown queue is not empty
10     while(B := Q_?.pop())
11     {
12         // break if maximal depth reached
13         if (B.depth >= max_depth) break;
14
15         // if satisfying coordinate exists in box
16         if (solve(B(x) ∧ φ(x)) = SAT)
17         {
18             // if unsatisfying coordinate exists in box
19             if (solve(B(x) ∧ ¬φ(x)) = SAT)
20             {
21                 // split box and increase depth if satisfying
22                 // and unsatisfying coordinates exist
23                 Q_new := B.split();
24                 Q_?.append(Q_new);
25             }
26             // no unsatisfying coordinate exists in box
27             else
28             {
29                 // box is satisfying
30                 S_+ := S_+ ∪ {B};
31             }
32         }
33         // no satisfying coordinate exists in box
34         else
35         {
36             // box is unsatisfying
37             S_− := S_− ∪ {B};
38         }
39     }
40
41     return S_+, S_−;
42 }
```

Algorithm 3.1: Parameter synthesis base algorithm. A flow-chart diagram for the this algorithm is depicted in Figure 3.1.

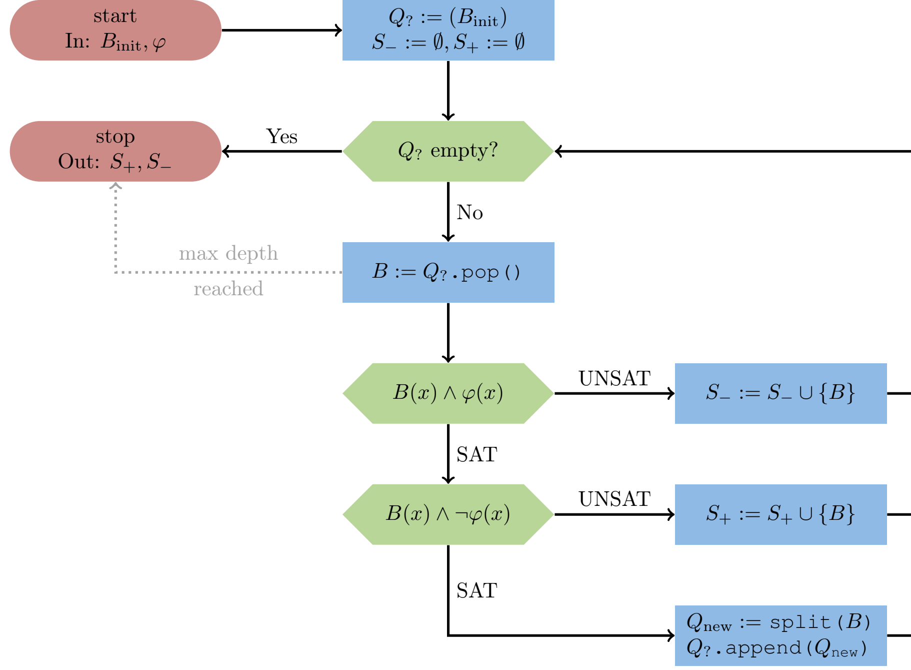

Figure 3.1: Flow-chart diagram for Algorithm 3.1.

Theorem 3.1.2. $\forall x(B(x) \to \neg\varphi(x)) \equiv B(x) \wedge \varphi(x)$ *is UNSAT.*

Proof. Using the implicit existential quantification of free variables by solvers as described in Section 2.1 for the step labeled with $(*)$, we get

$$\begin{aligned}\forall x(B(x) \to \neg\varphi(x)) &\equiv \neg\neg\forall x\ (B(x) \to \neg\varphi(x)) \\ &\equiv \neg\exists x\ \neg(\neg B(x) \vee \neg\varphi(x)) \\ &\equiv \neg\exists x\ (B(x) \wedge \varphi(x)) \\ &\equiv B(x) \wedge \varphi(x) \text{ is UNSAT.}\end{aligned}$$

□

It is thus sufficient to let the solver check $B(x) \wedge \neg\varphi(x)$ and $B(x) \wedge \varphi(x)$ in order to label a box B satisfying or unsatisfying, respectively. This observation, leads directly to the algorithm.

3.1.1 Algorithm Description

The central idea of the algorithm is the following: We check whether a box is satisfying, unsatisfying, or whether it contains both, satisfying and unsatisfying coordinates. In the first two cases we are done, in the last case, we split the current box into multiple smaller child boxes and then repeat the process.

First, in lines 5-7, we initialize the three main structures of the algorithm: A queue $Q_?$ of boxes for which it is not known whether they are satisfying, unsatisfying, or neither. As we initially do not know anything about any point in the initial box

B_{init}, which is passed as an input argument, $Q_?$ is initialized with B_{init}. We also initialize S_+ and S_-, two sets containing the boxes already identified as satisfying or unsatisfying, respectively. Initially, we do not have any (un)satisfying boxes and thus both sets are empty.

Line 10 marks the begin of the main loop. If all points of the initial box are classified as either satisfying or unsatisfying, resulting in $Q_?$ being empty, we can break the loop and jump directly to the return statement.

Otherwise, B is set to the first box in $Q_?$ and we check in line 13 whether it reaches the maximal depth. We also break the loop in this case. The depth is a simple attempt to measure progress in the algorithm. Let B_{child} be a box generated by splitting B. Then $B_{\texttt{child}}\texttt{.depth = }B\texttt{.depth + 1}$. The initial box B_{init} always has depth 0.

In line 16, we use an external solver to check whether B contains a point satisfying φ. If not, it is an unsatisfying box (see Theorem 3.1.2) and we add it to S_- in line 37.

If B does contain a point satisfying φ, we again use an external solver to check whether B also contains a point satisfying $\neg\varphi$ in line 19. Analogously to the case above, if B does not contain any such a point, it is a satisfying box (see Theorem 3.1.1) and we add it to S_+ in line 30.

In the remaining case, where B does contain (at least) one point satisfying φ and one point satisfying $\neg\varphi$, we cannot classify B as satisfying or unsatisfying. Thus, we split B into multiple new boxes in line 23, which together form a new queue Q_{new}. The new queue and thereby the new boxes, are then appended to $Q_?$ in line 24 as we do not know whether the new boxes are satisfying or unsatisfying (or neither). Splitting the boxes can be done in arbitrary ways. In the base implementation, a box is bisected in all dimensions. Given d dimensions, this results in 2^d new boxes. In Section 3.3 we have a detailed look at splitting heuristics.

Once $Q_?$ is empty or the maximal depth is reached, we break the loop in line 10 or 13, respectively. S_+ and S_- are then returned in line 41.

Interval Limitations Before going through the example, note that the current implementation does only support closed intervals: Assume a box $B = [-1{,}1] \times [-1{,}1]$ is bisected w.r.t. all dimensions. The resulting child boxes are then $[-1{,}0] \times [-1{,}0]$, $[-1{,}0] \times [0{,}1]$, $[0{,}1] \times [-1{,}0]$, and $[0{,}1] \times [0{,}1]$. Thus, the cuts themselves ($x = 0, -1 \leq y \leq 1$ and $y = 0, -1 \leq x \leq 1$ are covered by at least two boxes, the point at $x = 0, y = 0$ even by 4 boxes. Although this may seem like a drawback, this overlap is used for a heuristic called "clever sampling", which is described in Section 3.2.

3.1.2 Example

Let the input to the algorithm be the formula

$$\varphi(x, y) := (x \leq 0) \vee (y \geq x^3)$$

and the initial box

$$B_{\text{init}} := [-1, 1] \times [-1, 1].$$

We additionally assume the maximal depth to be five. The initialization (lines 5-7) will be

$$Q_? = (([-1, 1] \times [-1, 1], 0)),\ S_- = \emptyset,\ S_+ = \emptyset$$

where the 0 behind the box's dimensions indicates the depth. This initialization corresponds to Figure 3.2a. Now, we are entering the while loop with

$$B = Q_?\texttt{.pop()} = ([-1, 1] \times [-1, 1], 0).$$

As B does not reach the maximal depth (line 13), we go into the outer **if**-statement (line 16). The solver is now checking the satisfiability of the formula

$$B(x) \wedge \varphi(x) = ((-1 \leq x \leq 1) \wedge (-1 \leq y \leq 1)) \wedge ((x \leq 0) \vee (y \geq x^3)),$$

which is satisfiable, for example with $x = 0, y = 0$. We are thus entering the inner **if**-statement (line 19); the solver will therefore check

$$B(x) \wedge \neg\varphi(x) = ((-1 \leq x \leq 1) \wedge (-1 \leq y \leq 1)) \wedge \neg((x \leq 0) \vee (y \geq x^3)).$$

As this formula is satisfiable as well, e.g. through $x = 1, y = 0$, we end up in line 23 where B is now going to be split. The bisection in all dimensions will result in

$$\begin{aligned} Q_{\text{new}} = (&([-1, 0] \times [-1, 0], 1), \\ &([-1, 0] \times [\ \ 0, 1], 1), \\ &([\ \ 0, 1] \times [-1, 0], 1), \\ &([\ \ 0, 1] \times [\ \ 0, 1], 1)). \end{aligned}$$

We then append Q_{new} to $Q_?$ (line 24) and as $Q_?$ was empty before we have $Q_? = Q_{\text{new}}$. S_+ and S_- are still empty (Figure 3.2b).

Following the split, we begin a new iteration of the while loop (line 10), now with

$$B = Q_?\texttt{.pop()} = ([-1, 0] \times [-1, 0], 1).$$

The maximal depth of five is still not reached (line 13) and we enter the **if**-statement in line 16 where $B(x) \wedge \varphi(x)$ is checked for satisfiability. The formula is satisfiable, for example with $x = 0, y = 0$. We thus enter the inner **if**-statement (line 19) where the solver checks $B(x) \wedge \neg\varphi(x)$. This formula is not satisfiable since no point in B satisfies $\neg\varphi$ and we can conclude that B is a satisfying box according to Definition 2.2.3. We thus jump to line 27 where we add B to S_+ (line 30). At the end of this iteration we have

$$\begin{aligned} Q_? = &(([-1, 0] \times [\ \ 0, 1], 1), \\ &([\ \ 0, 1] \times [-1, 0], 1), \\ &([\ \ 0, 1] \times [\ \ 0, 1], 1)) \\ S_+ = &\{([-1, 0] \times [-1, 0], 1)\} \\ S_- = &\ \emptyset, \end{aligned}$$

depicted by Figure 3.2c.

In the following iteration, box $(([-1,0] \times [0,1],1)$ is assigned to S_+ as well. Boxes $([0,1] \times [-1,0],1)$ and $([0,1] \times [0,1],1))$ then get split in the preceding iterations. We

get, corresponding to Figure 3.2d,

$$
\begin{aligned}
Q_? = &(([\ \ 0, 0.5] \times [-1, -0.5], 2), ([\ \ 0, 0.5] \times [-0.5, 0], 2),\\
&([0.5, \ \ 1] \times [-1, -0.5], 2), ([0.5, \ \ 1] \times [-0.5, 0], 2),\\
&([\ \ 0, 0.5] \times [\ \ 0, \ \ 0.5], 2), ([\ \ 0, 0.5] \times [\ \ 0.5, 1], 2),\\
&([0.5, \ \ 1] \times [\ \ 0, \ \ 0.5], 2), ([0.5, \ \ 1] \times [\ \ 0.5, 1], 2))\\
S_+ = &\{([-1, \ \ 0] \times [-1, \ \ 0], 1), ([-1, \ \ 0] \times [\ \ 0, 1], 1)\}\\
S_- = &\ \emptyset.
\end{aligned}
$$

All boxes in $Q_?$ now have depth 2. The further process up to depth five is depicted in Figure 3.2. Additionally, Figure 3.2h shows the actual solution of the equation, which is reached for depth $\to \infty$ (borders of the boxes are omitted).

3.1.3 Correctness

Theorem 3.1.3. *The base algorithm is correct.*

Proof. To prove the correctness of the algorithm, we need to show that S_+ only contains satisfying boxes and S_- only unsatisfying ones.

S_+ is altered only in line 30, where the current box B is added. As line 30 is part of an **`else`** clause, it is only executed iff $B(x) \wedge \neg\varphi(x)$ is UNSAT. Theorem 3.1.1 shows that this is only the case iff B is a satisfying box. Thus, S_+ is correct.

Analogously, considering line 37 and Theorem 3.1.2, S_- is correct. □

3.2 Sampling

In this and the following section, we describe our approaches to boost the performance of the base algorithm above. The direction of our research was guided by testing results. Thus, this section occasionally refers to the experimental evaluation in Section 5 to motivate the research direction.

As we will see in Section 5, the most time consuming steps of the base algorithm from Section 3.1 are the solver calls in lines 16 and 19. To reduce running time, the main focus was thus to reduce the number of solver calls. One easy way to do so is through taking samples within a box B before calling the solver. In the base algorithm (Algorithm 3.1), this would mean adding a B`.sample()` in line 14.

For one or more $x \in B$, we check whether $\varphi(x)$ holds. If it does, we know that $B(x) \wedge \varphi(x)$ is satisfiable, as x is a solution. As a result, we can skip the solver call in line 16 and jump directly to line 19. If $\varphi(x)$ does not hold, then $\neg\varphi(x)$ does and it is possible to skip line 19 and jump directly to line 23. In case two samples x and y are found with $\varphi(x)$ and $\neg\varphi(y)$ both being true, it is even possible to skip both solver calls and directly jump from line 14 to 23.

Evaluating $\varphi(x)$ for a specific x is generally faster than a solver call on φ, which leads to overall better performance. However, due to implementation details (see Section 4.2), this acceleration factor was not as fast as hypothesized.

Model Saving Whenever the solver is called and terminates with SAT, it can also provide the model it has found. So, instead of taking a point and evaluating it, we could also make use of these models: A model for the solver calls in lines 16 and 19

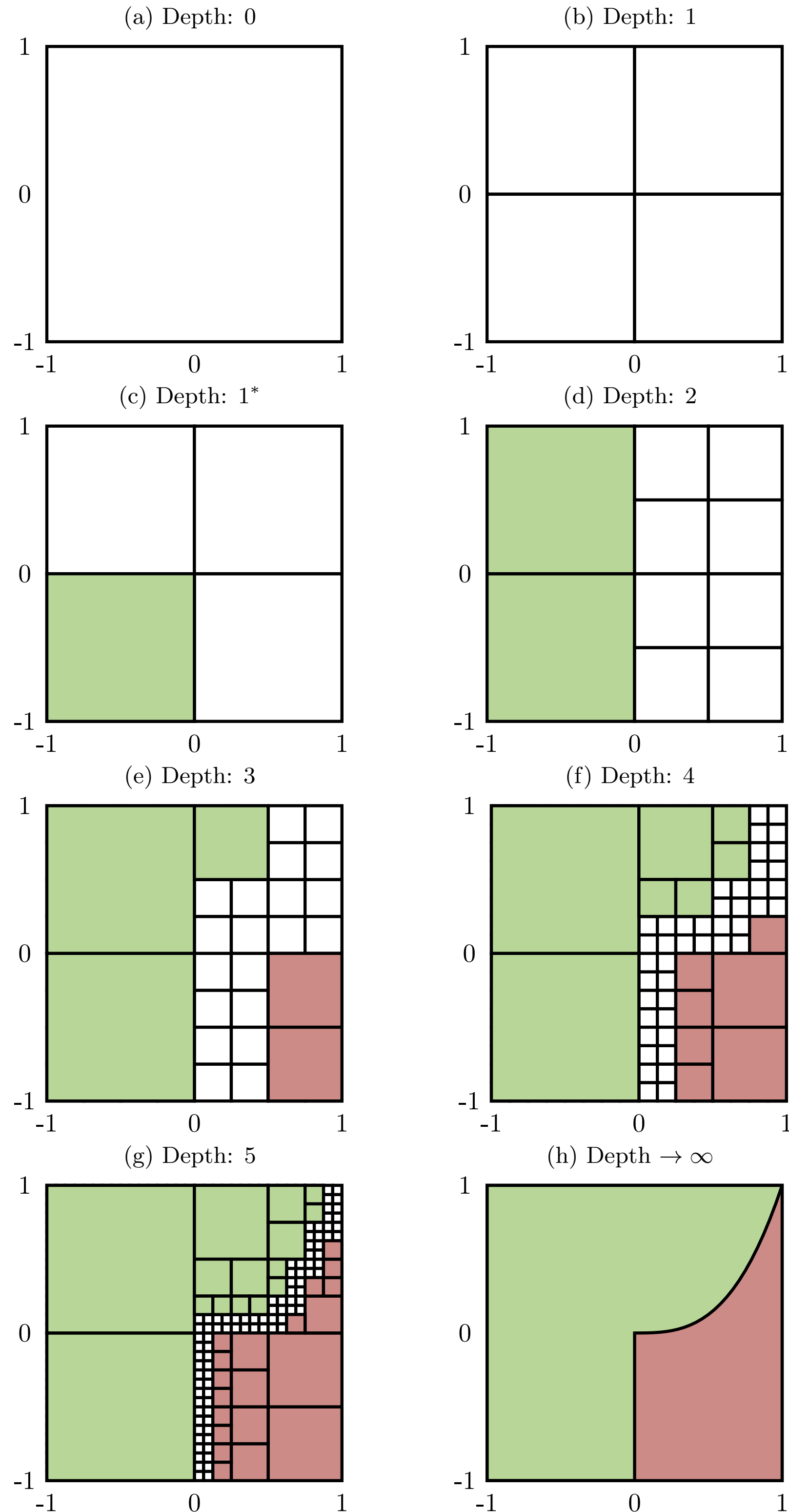

Figure 3.2: Visualization of Algorithm 3.1, executed on the input formula $\varphi(x, y) := (x \leq 0) \vee (y \geq x^3)$ and initial box $B_{\text{init}} := [-1, 1] \times [-1, 1]$. The x-axis is depicted horizontally, the y-axis vertically. The process is depicted up to depth 5. Figures 3.2a-b and d-g represent the snapshot when for the first time all boxes in $Q_?$ have the specified depth. Figure 3.2c corresponds to an intermediate stage further explained in the text. Figure 3.2h shows the actual solution of the equation, which is reached for depth $\to \infty$. Satisfying boxes are depicted in green (●), unsatisfying boxes are depicted in red (●), other, unknown boxes are depicted in white.

is a coordinate x that satisfies $\varphi(x)$ or $\neg\varphi(x)$, respectively. We can now store this coordinate and whether it is satisfying or not. When splitting the respective box in line 23, we assign the coordinate to the appropriate child box. For the child box, we then may be able to skip solver calls. Although this method has the advantage of getting samples without additional overhead, we have to carry these samples when splitting the box (option two in the paragraph below) which does introduce overhead.

Sample Splitting When splitting a box in line 23, we have different options on what to do with its samples. In this thesis, we implemented and analyzed the following.

1. One option is to just omit all samples of the box. This has the advantage that we do not need to check for each sample in which of the resulting boxes it belongs. On the downside, we loose information and need to sample in each iteration of the loop.
2. An opposing alternative is to iterate over all samples in a box and check for each in which of the resulting boxes it belongs. We do not loose information in this case, but therefore introduce significant overhead.
3. An option that has advantages of both previous points is to sample split dependent or split sample dependent. This means that, when splitting, we already know which samples will end up in which resulting box. To clarify this, assume the following setting: We have a 2D box which is always going to be bisected in both dimensions resulting in four new boxes. If we sample in the top left corner of the initial box, we know that the sample is valid for the top left box after splitting as well.

Note that in the first case it is not even necessary to store the exact coordinates of the sample. It is sufficient to store the information whether the sample satisfies φ or its negation. In the third case we also do not need to store the exact location, however we need a method to derive which sample will end up in which box.

Clever Sampling As described in Section 3.1, the current implementation is only able to handle closed intervals. However, together with sample splitting option three from above, we can make use of this fact: If we sample x in the center of a box B and use a splitting method that bisects B in an arbitrary number of dimensions, we know for sure, that this sample x is a valid sample for all child boxes B_c of B. Let $B := [-1,1] \times [-1,1]$. We now sample in the center of B at $x = 0, y = 0$. Now we bisect B w.r.t. all dimensions, resulting in child boxes $[-1,0] \times [-1,0]$, $[-1,0] \times [0,1]$, $[0,1] \times [-1,0]$, and $[0,1] \times [0,1]$. The point $x = 0, y = 0$ is part of all child boxes, making the sample taken for B a valid sample for all its children.

3.3 Splitting

When it comes to quickly finding better solutions according to Definition 2.2.2, one option is to optimize the splitting of boxes: If we are able to find large classifiable boxes, we need less solver calls to classify the same volume of the initial box. Or, the other way around: Large boxes lead to more classified volume in the same time. It is

therefore important, to set the splits at good positions in order to prevent unnecessary solver calls.

In the base algorithm, we simply bisect a box in every dimension, which results in $2^{\text{dimension}}$ child boxes and makes scaling difficult.

One idea to improve this method is to not bisect in all dimensions but to only split in one dimension. Starting with the first dimension in the first iteration of the while loop, we increase the dimension in every iteration until we have split in every dimension once and use the first one again.

Both of the last two methods were zero-knowledge approaches, meaning that we did not use any information to cleverly split the boxes. A different approach could use samples to cleverly split boxes: given a set of points we would then try to split in a way that gives us a relatively large box containing only SAT or UNSAT samples. A condition for this method is having two or more samples. The more samples we have, the better are the chances of finding a good split. However, as we are going to see in Chapter 5, taking samples takes relatively much time. Taking one sample for each box already increased running time substantially, which resulted in us not researching further into more sophisticated splitting heuristics.

In contrast, Junges et al. could impactfully improve their running times with more sophisticated splitting heuristics [Jun+19].

3.4 Incremental Solving

In Section 2.1, we explained why incremental solving and therefore context preservation may be beneficial in SMT solving. We made use of this technique in this thesis in two different ways.

In general, we made use of two different solvers and their contexts. One solver for the satisfiability-checks in line 16 and one for the satisfiability-checks in line 19. Before even checking satisfiability within boxes, we checked the satisfiability once for φ and $\neg\varphi$ during the initialization phase to set meaningful contexts for both solvers.

To carry this idea forward, we also made use of the fact that a box B always contains all of its child boxes. Let B_c be a child box of B. Then, checking $\varphi(x) \wedge B(c) \wedge B_c(x)$ will have the same result as $\varphi(x) \wedge B_c(x)$, as $B_c(x) \to B(x)$. As we have already checked $\varphi(x) \wedge B(x)$ previously, we can thus reuse this context when checking $\varphi \wedge B_c$.

Theoretically, we could use this idea to always reuse the context of a box for its child. As a result, the solver handling box B at an arbitrary depth would always solve in the context (or extensions of these contexts) of all its predecessors. However, this is not necessarily helpful as larger contexts also contain more implications that are not needed anymore.

In our code, we have thus implemented an option to always use one context for two layers. A context created for a box B with even depth is used for all its children. To prevent overhead from storing more than one context at a time, we altered the base algorithm in the following way: For a box B_{even} with even depth, instead of inserting its child boxes at the end of $Q_?$, we insert them at the beginning. Their respective child boxes are then again "normally" inserted at the end. Once all child boxes of B_{even} were handled, B_{even}'s context is deleted.

3.5 Resulting Features

The concepts implemented during this thesis have been explained above. Here, we summarize the heuristical options we consider.

- Sampling heuristics (see Section 3.2)

  1. no sampling
  2. center (one sample is taken in the center of the current box)
  3. clever sampling
  4. additional features: model saving, sample splitting (The three options above mutually exclude each other. In contrast, the additional features can be enabled additionally)

- Splitting heuristics (see Section 3.3)

  1. bisect in all dimensions
  2. bisect in one dimension

- Incremental solving (see Section 3.4)

3.6 Handling Quantifiers

The algorithm described above would work for arbitrary φ, including arbitrarily quantified φ. However, in lines 16 and 19, we rely on an external solver and, as mentioned in Section 2.1, to the best of our knowledge, there does not exist any SMT solver capable of finding solutions for arbitrarily quantified SMT formulas. Thus, the plain implementation of our algorithm is only able to support quantifiers to the extent the underlying solver is able to support quantifiers.

Despite this shortcoming, we were able to find ways in which it is possible to support quantified formulas.

3.6.1 Native Support

Using the current algorithm for quantifier-free formulas, it is already possible to handle quantified formulas to a limited extent.

Universal Quantification Let ψ be quantifier-free with the semantical meaning of being purely existentially quantified and

$$\varphi(x) := \forall a\ \psi(x, a).$$

Considering Definition 2.2.3 for a satisfying box, then (similar to the proof of Theorem 3.1.1):

$$\begin{aligned}
\forall x(B(x) \rightarrow \varphi(x)) &\equiv \forall x\ (B(x) \rightarrow \forall a\ \psi(x,a)) \\
&\equiv \forall x\ (\neg B(x) \vee \forall a\ \psi(x,a)) \\
&\equiv \forall x \forall a\ (\neg B(x) \vee \psi(x,a)) \\
&\equiv \neg\neg\forall x \forall a\ (\neg B(x) \vee \psi(x,a)) \\
&\equiv \neg\exists x \exists a\ \neg(\neg B(x) \vee \psi(x,a)) \\
&\equiv \neg\exists x \exists a\ (B(x) \wedge \neg\psi(x,a)) \\
&\equiv B(x) \wedge \neg\psi(x,a) \text{ is UNSAT.}
\end{aligned}$$

As ψ is quantifier free, it is thus sufficient to let the solver check $B(x) \wedge \neg\psi(x,a)$ in order to label a box satisfying, given a universally quantified formula. Unfortunately, we can not do the same in case of unsatisfying boxes: considering Definition 2.2.4 for an unsatisfying box, then:

$$\begin{aligned}
\forall x(B(x) \rightarrow \neg\varphi(x)) &\equiv \forall x\ (B(x) \rightarrow \neg\forall a\ \psi(x,a)) \\
&\equiv \forall x\ (\neg B(x) \vee \neg\forall a\ \psi(x,a)) \\
&\equiv \forall x\ (\neg B(x) \vee \exists a\ \neg\psi(x,a)) \\
&\equiv \forall x \exists a\ (\neg B(x) \vee \neg\psi(x,a)).
\end{aligned}$$

The transformation results in alternating quantifiers for which there is no way of handling it using a solver incapable of handling quantifiers. As a result, for universally quantified formulas we are able to find satisfying regions but not unsatisfying regions. As we will see next, this is inverse for existentially quantified formulas.

Existential Quantification Let ψ be quantifier-free and

$$\varphi(x) := \exists e\ \psi(x,e).$$

Considering the Definition 2.2.4 for an unsatisfying box, then (similar to the proof of Theorem 3.1.2):

$$\begin{aligned}
\forall x(B(x) \rightarrow \neg\varphi(x)) &\equiv \forall x\ (B(x) \rightarrow \neg\exists e\ \psi(x,e)) \\
&\equiv \forall x\ (\neg B(x) \vee \neg\exists e\ \psi(x,e)) \\
&\equiv \forall x\ (\neg B(x) \vee \forall e\ \neg\psi(x,e)) \\
&\equiv \forall x \forall e\ (\neg B(x) \vee \neg\psi(x,e)) \\
&\equiv \neg\neg\forall x \forall e\ (\neg B(x) \vee \neg\psi(x,e)) \\
&\equiv \neg\exists x \exists e\ \neg(\neg B(x) \vee \neg\psi(x,e)) \\
&\equiv \neg\exists x \exists e\ (B(x) \wedge \psi(x,e)) \\
&\equiv B(x) \wedge \psi(x,e) \text{ is UNSAT.}
\end{aligned}$$

As ψ is quantifier free, it is thus sufficient to let the solver check $B(x) \wedge \psi(x,a)$ in order to label a box unsatisfying, given an existentially quantified formula. Analogously to the case for universal quantifiers from above, trying the same for satisfying boxes according to Definition 2.2.3 would result in alternating quantifiers which the underlying solver and thus the base algorithm may not be able to handle.

In conclusion, we are able to find unsatisfying regions for existentially quantified formulas but are not able to provide information on satisfying regions.

```
1  synthesize_quantifiers(B_init, ψ)
2  {
3      // transform ψ into prenex normal form
4      ψ' = pnf(ψ);
5
6      // get prefix and matrix of ψ
7      pre = prefix(ψ');
8      φ = matrix(ψ');
9
10     // add new dimensions to inital box B_init.
11     B'_init = extend(B_init, pre);
12
13     // perform synthesis
14     S'_+, S'_-, Q'_? = synthesize(φ, B'_init);
15
16     // derive actual result
17     S_+, S_- = derive_result(S'_+, S'_-, Q'_?, pre);
18
19     return(S_+, S_-);
20 }
```

Algorithm 3.2: Parameter synthesis algorithm for handling quantifiers. A flow-chart diagram for the this algorithm is depicted in Figure 3.3

3.6.2 General Support

The native support from above has clear limitations. It is nonetheless possible to support arbitrarily quantified formulas through a slight adjustment of the algorithm. In the rest of this section, we give a sketch on how to do so. Algorithm 3.2 provides pseudocode of the adjustment, Figure 3.3 an equivalent flow-chart diagram.

The main idea is to treat initially quantified variables in the same way as free variables through introducing new dimensions to the inital box. Each new dimension corresponds to one quantifier. The boundaries of the modified initial box in the new dimensions will be $-\infty$ and ∞. Then, we use the base algorithm on the formula with removed quantifiers and the modified initial box. In the end, we contract the added dimensions of the base algorithm's output to receive the actual result.

Let ψ be the input formula and B_{init} the initial box (line 1). First, in line 4, we transform ψ into an equivalent PNF formula ψ'. ψ' has free variables $x = (x_1,...,x_n)$ and bounded variables $y = (y_1,...,y_m)$ quantified by $Q_1, ..., Q_m$ in this order. ψ' consists of a prefix `pre` (line 7) and a matrix φ (line 8) with free variables x and y.

The formula φ will later be used as an input for the base algorithm. Since the dimension of the initial box input of the base algorithm is equal to the number of free variables of the input formula φ, we need to extend the initial box B_{init}. Since quantified variables are not actually restricted, we add $-\infty$ and ∞ as new boundaries for every quantified variable y_j. Doing so, we get the extended initial box B'_{init}. This procedure is represented by the `extend` method in line 11.

In line 14, we then simply use the existing **synthesize** algorithm on φ and B'_{init} to get intermediate results S'_+, S'_- and $Q'_?$.

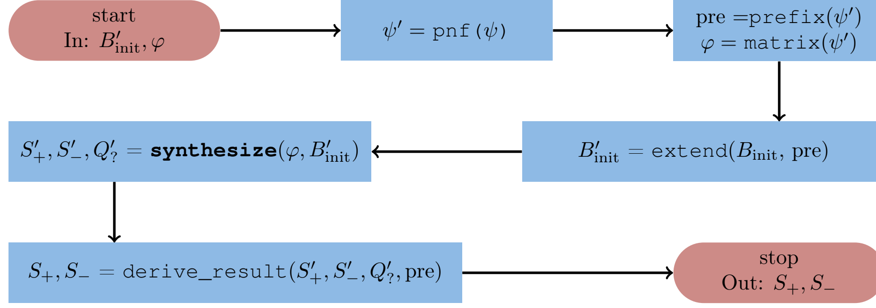

Figure 3.3: Flow-chart diagram for algorithm 3.2.

All boxes in S'_+, S'_- and $Q'_?$ now also have the additional dimensions from the initially quantified variables. The last step is to derive the actual result through removing these dimensions. This step is not trivial and we only sketch it in this thesis.

The central idea is to begin with $(n+m)$-dimensional boxes $B = I_1 \times ... \times I_{n+m}$ and project out the additional dimensions, resulting in $B^k = I_1 \times ... \times B_{n+k}$ for any $0 \leq k \leq m$. We iteratively handle the quantifiers Q_k in the order $k = m,...,1$ as follows.

For Q_k being $\exists$, we define $S_+^{k-1} := \{B^{k-1} | B^k \in S_+^k\}$ and $S_-^{k-1} := \{B^{k-1} | \exists i \in \mathbb{N}. \exists B_1^k,...,B_i^k \in S_-^k.(B^{k-1} = \bigcap_{j=1}^i B_j^{k-1}) \wedge (B^{k-1} \times \mathbb{R} \subseteq \bigcup_{j=1}^i B_i^k)\}$. Intuitively, an $(n+k-1)$-dimensional box B^{k-1} is satisfying if there exists an $(n+k)$-dimensional satisfying box in its cylinder $B^{k-1} \times \mathbb{R}$, and it is unsatisfying if the whole cylinder is covered by unsatisfying $(n+k)$-dimensional boxes.

Analogously, for Q_k being $\forall$, we define $S_-^{k-1} := \{B^{k-1} | B^k \in S_-^k\}$ and $S_+^{k-1} := \{B^{k-1} | \exists i \in \mathbb{N}. \exists B_1^k,...,B_i^k \in S_+^k.(B^{k-1} = \bigcap_{j=1}^i B_j^{k-1}) \wedge (B^{k-1} \times \mathbb{R} \subseteq \bigcup_{j=1}^i B_i^k)\}$. Intuitively, an $(n+k-1)$-dimensional box B^{k-1} is unsatisfying if there exists an $(n+k)$-dimensional unsatisfying box in its cylinder $B^{k-1} \times \mathbb{R}$, and it is satisfying if the whole cylinder is covered by satisfying $(n+k)$-dimensional boxes.

These computations are relatively easy to implement algorithmically if we assume a cylindrical arrangement of the boxes, i.e. that for each pair of boxes $B_1^k, B_2^k \in S_+^k \cup S_-^k$ we have that either $B_1^{k-1} = B_2^{k-1}$ or $B_1^{k-1} \cap B_2^{k-1} = \emptyset$ hold true. Otherwise, the sets S_+^{k-1} and S_+^{k-1} might contain overlapping boxes and their computation might require box splitting; see notes below. (Note that we used "$\cap$" in this paragraph in a way that only considers the interior of the boxes. This is necessary, because all boxes have closed intervals, which might overlap only on their borders.)

Once we have projected out all m dimensions $n+m$ down to $n+1$, we end up with $S_+^0 = S_+$ and $S_-^0 = S_-$. This iterative reduction of dimensions to receive the final result is represented by the method `derive_result` in line 17.

Finally, we return S_+ and S_- in line 19.

Correctness By construction of B^0 and therefore B, we know that a box is satisfying iff $\forall x(B(x) \rightarrow \psi'(x))$ and unsatisfying iff $\forall x(B(x) \rightarrow \neg\psi'(x))$. The algorithm is therefore correct.

Notes Without the cylindricity condition, it is necessary to split boxes in the process. As the cylindricity condition is not ensured by our implementation because it would be inefficient, an implementation would need to take care of this circumstance.

It would probably be beneficial to use distinct splitting heuristics for the treatment of quantified dimensions and free dimensions: If, for example, a dimension would be universally quantified and an unsatisfying box B_- is found, it is not necessary to further split other boxes that only differ from B_- in this universally quantified dimension.

Due to the scope of this thesis, we were not able to implement the general support for quantifiers. As this would be a useful feature, it may be a possible subject for future work on this topic.

3.7 GUI Projection

As part of this thesis, we implemented a graphical user interface (GUI) to depict the outcome of the parameter synthesis. For details, read Section 4.4. A challenge of the GUI is the number of dimensions: while the problem dimension can be arbitrarily large in theory, a representation of the output on a screen can only have two dimensions. Thus, information is necessarily going to be lost during the visualization process. We address this issue in the same way as Wiegel does [Wie21]: trough an existential projection.

Assume an input formula φ with $d > 2$ free variables. Let S_+, S_- and $Q_?$ further be the output of a parameter synthesis performed on φ. Without loss of generality, we assume that the first two dimensions are chosen to be depicted. We depict a box $B = [l_1,u_1] \times [l_2,u_2]$ as satisfying, if there exists a box $B' \in S_+$ with $B \subseteq B'|_{1,2}$. If such a box B does not exist in S_+, but there exists one in $Q_?$, then B is depicted as unknown. If such a box B' only exists in S_-, the box is depicted as unsatisfying.

As a consequence, a point (a_1,a_2) lies in an area depicted as satisfying if the parameter synthesis found an extension $(a_3,...,a_d)$ such that $\varphi(a_1, a_2, a_3, ..., a_d)$ evaluates to true. If (a_1,a_2) lies in an area depicted as unknown, there might exist an extension $(a_3,...,a_d)$ such that $\varphi(a_1, a_2, a_3, ..., a_d)$ evaluates to true, however, the synthesis did not find one. If (a_1,a_2) lies in an area depicted as unsatisfying, for all extension $(a_3,...,a_d)$ within the initial box, $\varphi(a_1, a_2, a_3, ..., a_d)$ evaluates to false.

Technically, this is realized using the following process: first, each box of the solution is projected to the two chosen dimensions through omitting all other dimensions. Then, unsatisfying, unknown and satisfying boxes are drawn in this order (where a later drawing overwrites previous ones at the same position).

Correctness Through plotting the satisfying and unknown areas after the unsatisfying ones, we ensure that only those regions are actually depicted as unsatisfying, for which it is guaranteed that there does not exist a satisfying extension within the initial box. Plotting satisfying areas last ensures that all regions for which there does exist an extension are marked satisfying. Consequently, for all remaining regions the following holds: (1) No box has been found that satisfies φ. (2) There exist boxes for which it is unknown whether they satisfy the formula or not. It is thus justified to mark them unknown.

Implications Taking this projection into account, the GUI implementation is actually able to handle bounded existential quantifiers. Bounded means that the quantification only considers a restricted interval. So the implementation is still not capable of handling a formula $\exists x \varphi(a,x)$. However, it can handle $\varphi' := \exists x(x \in [l,u] \rightarrow \varphi(a,x))$. To do so, parameter synthesis is performed on $\varphi(a,x)$ as usual. The initial box is extended by an additional dimension with a lower bound l and an upper bound u. If x is not chosen to be depicted, the existential projection will ensure that x is existentially quantified within the given boundaries.

Chapter 4

Implementation

In this chapter, we provide details about the implementation. We present the program that resulted from this thesis, justify selected design choices and describe some issues encountered during the implementation process. The implementation is uploaded on GitHub [Rad].

4.1 Overview

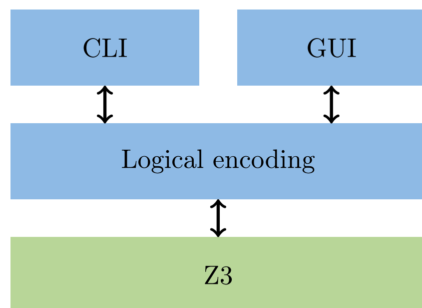

Figure 4.1: Visualization of the code structure. The logic of the code is implemented using the Z3 solver. On top of the logical encoding, our implementation provides two user interfaces: a command line interface (CLI) and a graphical user interface (GUI). Code implemented by us is blue (●), existing code green (●).

Programming Language Choice This thesis is based on the work done by Wiegel [Wie21]. Consequently, one option was to use their implementation, PaSyPy, which is written in Python. However, a main issue of this implementation is its lack of supporting exact arithmetic. Instead, it uses floating point arithmetic. Since fixing this issue requires rewriting a significant part of the code anyway, we therefore decided to switch programming language and use C++ instead, aiming for performance gains. A short comparison in Section 5.3.2 justifies the switch.

SMT Solver Regarding the SMT solver used, we initially planned to write the code in such a way that we would be able to use different solvers, including Z3 [MB08], which was used by Wiegel [Wie21]. To make the code compatible with arbitrary solvers, it would have been necessary to use a library providing an independent arbitrary-precision datatype, such as GMP [Gra]. Z3, however, uses its own datatype, `z3::expr`. This makes conversions between the independent datatype and `z3::expr` necessary, but Z3 only supports conversions from a

z3::expr to a char-array. As a result, we get the following conversion sequence: z3::expr $\leftrightarrow$ char-array $\leftrightarrow$ GMP-datatype. These conversions would be necessary when the boundaries for a box are added to the solver, during sampling, and when using model saving. As the support of multiple solvers was not the main goal of this thesis, we thus chose to only support Z3. This enabled us to use z3::expr as the native datatype and thereby prevent conversions.

SMT Language Support Z3 supports the SMT-LIB language standard [MB08; BST10]. As this standard is very suitable for parameter synthesis and to minimize parsing efforts, our implementation uses the standard too. The standard also specifies an input file standard for SMT solvers. It requires the extension .smt2.

Code Structure The algorithm, datastructures needed, and a programming interface in order to run a parameter synthesis are all written in of the logical encoding part of the code. We offer two different user-interfaces: First, a very thin command-line interface (CLI) for evaluation and development purposes. Second, a graphical user interface similar to the one in [Wie21]. An overview of this structure is depicted in Figure 4.1.

4.2 Logical Encoding

The very base of the implementation is the class polytope. An instance of this class represents a single polytope. The current implementation only supports boxes (orthotopes), meaning a subset of all polytopes. Consequently, boxes are implemented as a subclass of polytope. polytope is then used by the programming interface class parameter_synthesis, which implements the algorithms presented in Chapter 3. It provides a simple interface for creating, executing or resuming a parameter synthesis. During the instantiation, it is possible to pass the formula, define an initial box, set the maximal depth, and choose between implemented features.

Because the interface class parameter_synthesis does only use polytope, the code is ready for an extension to polytopes other than boxes, possibly even arbitrary polytopes.

One detail that needed a sophisticated approach was evaluating the formula during the sampling process. An easy way to evaluate a formula φ with one free variable at point x is to replace every occurrence of the free variable with x and then evaluate the resulting formula, which has no free variables anymore. Z3 provides a substitute function for the replacing process. The only way to now safely get the evaluation result is to call a solver on the substituted formula. However, calling the solver just to sample the formula introduces overhead. In fact, this overhead is too large to implement a sampling feature that reduces running time. Another way to evaluate a formula without free variables is using the simplify function Z3 provides. Our own tests have shown that calling a solver on simple instances is slower than calling simplify on the same instance by a factor of around 1000. The problem with the simplify function is that it does not give any guarantees. A formula $\sqrt{4} = 2$ may be simplified to the Boolean value *true*, but it may also, theoretically, be simplified to the formula $2 = 2$. To ensure an error-free result, we therefore implemented a sanity check to prevent that formulas have not been completely simplified.

4.3 CLI

The command line interface (CLI) has been built using the Boost library Program Options [Boo]. It provides an extensive set of functions for building CLIs. Our CLI has one required parameter: an SMT-LIB file containing the function the parameter synthesis should be performed on. The example from Section 3.1.2 could be reproduced with a file `formula.smt2` containing the following lines:

```
(declare-const x Real)
(declare-const y Real)
(assert (or (>= 0 x ) (>= y (* x (* x x)))))
```

All other parameters are optional and can be viewed through executing with the `help` option.

```
$ ./build/cli --help
Allowed options:
  -h [ --help ]
                      produce help message
  --boundaries-file arg
                      Text file containing a list of all
                      variables and their boundaries. The
                      file should contain lines of the from
                      '<variable-name> <lower-bound>
                      <upper-bound>'.
  --default-boundaries arg (=10)
                      Set default 'radius' of the inital
                      orthotope.
  --splitting-heuristic arg (=bisect_all)
                      Select a splitting heuristic. Options
                      are 'bisect_all' and 'bisect_single'
  --sampling-heuristic arg (=no_sampling)
                      Select a sampling heuristic. Options
                      are 'no_sampling', 'center', and
                      'clever'.
  --max-depth arg (=10)
                      Set maximal depth.
  --save-model
                      Save models found by solver. Only
                      useful if 'split-samples' enabled.
  --incremental
                      Enable incremental solving.
  --split-samples
                      Also carry samples when splitting
                      orthotopes.
  --splits-needed
                      Returns true if splits are needed to
                      process this formula.
```

```
  --print-orthotopes
                    Prints all (SAFE, UNSAFE and UNKNOWN)
                    resulting orthotopes.
```

The rest of this section will explain the optional arguments.

Meta Parameters

Initial Box In order to provide the parameters of the initial box, it is possible to provide an additional file containing the initial boundaries using the `boundaries-file` option. Each line of this file needs to have the following structure:

```
<variable-name> <lower-bound> <upper-bound>
```

If the file is not specified correctly (variable missing, invalid path, ...), an error message will help finding the issue. In order to specify $[-1,1]^2$ as initial box like in the example from Section 3.1.2, execute with `--boundaries-file=boundaries.txt`, where the file `boundaries.txt` contains

```
x -1 1
y -1 1
```

If no file is specified, the initial box is defaulted with $[-10,10]^n$, where n is automatically determined from the formula passed. It is also possible to change this default value with the `--default-boundaries` option; `--default-boundaries=`x, will lead to an initial box $[-x,x]^n$.

Maximal Depth The maximal depth until which the synthesis should run can be specified by the `max-depth` argument, which defaults to 10. A reproduction of the example in Section 3.1.2 would require `--max-depth=5`.

Heuristics

Most optional arguments concern the heuristics listed in Section 3.5.

The sampling heuristic can be chosen via the `sampling-heuristic` option. Possible choices are `no_sampling`, `center`, and `clever`, which have been explained in detail in Section 3.2.

Further, it is possible to enable model saving and sample splitting as described in Section 3.2 through setting the flags `save-model` and `split-samples`, respectively.

Equivalently it is possible to specify a splitting heuristic through the option `splitting-heuristic`.

Incremental solving can be enabled through `incremental`.

Other Arguments

The remaining two arguments are mainly for development and evaluation purposes. When `splits-needed` is set, a parameter synthesis is not actually being performed. Instead, it is only checked whether there exist satisfying *and* unsatisfying coordinates

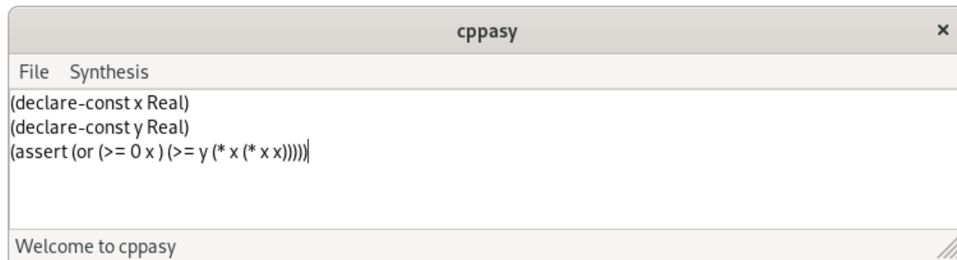

Figure 4.2: Screenshot of the main window. It can be used to enter and SMT-LIB formula on which the parameter synthesis can be performed. Using the "Synthesis" tab of the navigation bar, it is possible to open the preferences window or to execute the synthesis.

in the initial box, which implies that splits are needed to further analyze the given region. The return value is `1`, iff splits are needed. This feature was added because a notable number of benchmarks used to evaluate the algorithms are not satisfiable at all. These benchmarks are not suitable for evaluating purposes, as the calculation will be finished after the first iteration in the main loop. Using this feature we eliminated these "uninteresting" benchmarks.

The option `print-orthotopes` is almost self-explanatory as it simply prints all resulting boxes (their dimension and whether they are classified as safe, unsafe or unknown) in the console. As, depending on the depth, the number of these orthotopes quickly exceeds 100, the output may not be very helpful for anything else then development purposes.

4.4 GUI

One focus of this thesis was to provide a suitable visualization of the results similar to the one by Wiegel [Wie21]. Our GUI has been built using wxWidgets, a cross-platform GUI library [wxW].

The GUI consists of three windows. First, a main window to enter the formula (Figure 4.2). Second, a window to set parameters concerning the synthesis (Figure 4.3). Third, a window presenting the plot (Figure 4.4).

Main Window In the main window, a user can enter the formula on which the parameter synthesis should be performed. The input must comply with the SMT-LIB standard. From the main window, the user is able to (1) open the preferences window or (2) start the parameter synthesis. Both is possible via the "Synthesis" tab in the navigation bar. If a user executes the parameter synthesis, the plot window is opened automatically after the execution has been performed. The file tab provides an option to close the window. Figure 4.2 is a screenshot of the main window.

Preferences Using the preferences window, the user is able to specify the initial box, the maximal depth and which dimensions to plot. They can further decide which of the features listed in Section 3.5 are used during the executing of the parameter synthesis. The preferences dialog is generated dynamically in dependence of the

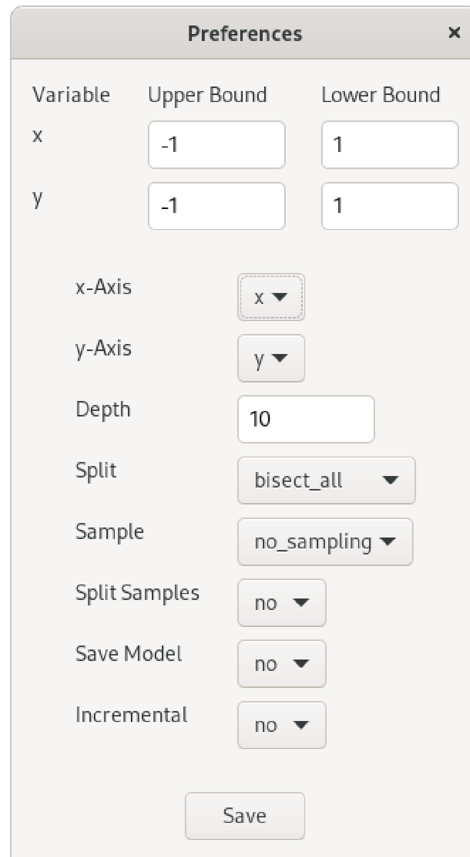

Figure 4.3: Screenshot of the preferences window.

formula entered in the main window. This is necessary, because the names and number of variables may change. Invalid values (upper bound > lower bound, ...) are rejected and the user is requested to change them through a pop-up dialog. If no preferences are specified before the execution, all options are defaulted with the ones appearing when opening the preferences window. Figure 4.3 is a screenshot of the preferences window.

Plot As mentioned above, the plot window opens automatically after performing the parameter synthesis. The window is resizable and the axis labeling changes dynamically with the window size. It is further possible to resume the parameter synthesis using the "Resume"-button. For each click on the button, the maximal depth is incremented by one and the execution is resumed, reusing previous execution results. Figure 4.4 is a screenshot of the plot window.

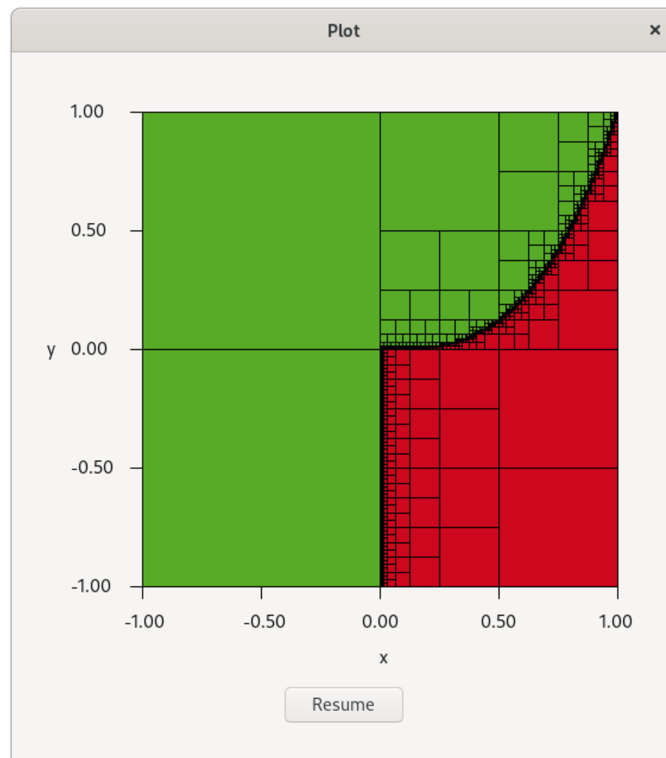

Figure 4.4: Screenshot of the plot window. It is used to depict the result of the parameter synthesis performed on the formula entered in the main window.

Chapter 5

Experimental Evaluation

In this chapter, we experimentally evaluate the implementation described in Chapter 4. The evaluation is done only using the CLI for two main reasons: First, using the GUI, we noticed that functions provided by the GUI, in particular the plotting, are done almost instantly once the parameter synthesis has been performed. Evaluating running times of the GUI would thus not be meaningful. Second, evaluating the CLI alone is sufficient for a comparison of the proposed features.

In Section 5.1, we describe the setup of the experiments. We shortly explain the evaluation strategy in Section 5.2 before presenting and analyzing the results in Section 5.3.

5.1 Setup

The evaluation was performed on the SMT-LIB QF-NRA benchmark collection of 11552 quantifier free NRA formulas [SMT]. The running time analysis has been performed on a cluster with 4x 2.1 GHz AMD Opteron processors, each consisting of 12 cores. The cluster has 192 GB of RAM. However, the current implementation does not have a parallelization option. As a consequence, only a single of all 48 cores was used at a time.

In Section 4.2, we described why it was theoretically necessary to implement a sanity check when using Z3's `simplify` function. The sanity check results in a slowdown of factor 2 in the evaluation process of formulas. However, we observed that the sanity check was not necessary for any benchmark used. For running time measurements, we thus disabled sanity checks. Note that this does not affect the soundness of the approach.

5.2 Strategy

Benchmark Filtering

Not all of the 11552 benchmarks are suitable for evaluation: Some of the formulas may not be satisfiable. Running the synthesis on them would not be valuable for the evaluation process as the synthesis would be finished after a single iteration of the main algorithm described in Section 3.1. Other formulas are just too hard to

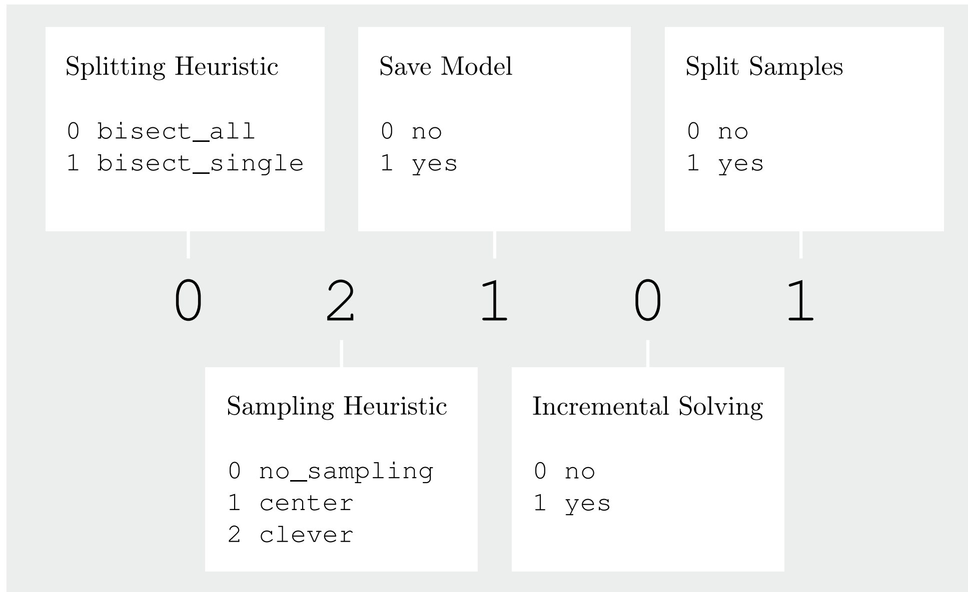

Figure 5.1: Description of the 5 digits notation used to express a specific setting of a parameter synthesis.

solve. To measure the influence of the implemented features, a certain depth has to be reached. If this depth is not reached in a feasible amount of time, the benchmark is not suitable for evaluating the implementation too.

Furthermore, choosing a depth that is high enough to give meaningful results on the one hand, but not too deep to eliminate too many benchmarks through timeouts on the other hand, is not a clear process. Choosing boundaries for an initial box gives a similar challenge.

To address these challenges, we proceeded in the following way. Through taking a look at multiple benchmarks, we found that if a formula was satisfiable, the satisfying area was in nearly all cases within the box $B = [-4,4]^n$. We thus chose $B = [-4,4]$ as the initial box for the whole evaluation process. As a maximal depth that is high enough to give meaningful results we chose 6. Due to the incremental solving (Section 3.4), it was sensible to chose an even depth and choosing 8 would have eliminated too many benchmarks. We additionally chose 15 seconds as a timeout value due to time constraints on the whole evaluation process.

Out of the 11552 benchmarks, we eliminated 9179 using the above parameters, leaving 2373 benchmarks for the evaluation. Around 90% were eliminated through timeouts because they were not able to calculate up to depth 6 within 15 seconds. Another around 10% were eliminated because they did not have satisfying *and* unsatisfying coordinates within the initial box. Interestingly, only 13 benchmarks are not part of the meti-tarski directory in the SMT-LIB benchmark collection.

Choosing Setting Combinations

To measure the impact of one specific feature from the list in Section 3.5, each of them needs to be evaluated by itself, meaning all other features need to be disabled. Additionally, combinations of features are compared in the evaluation.

Figure 5.1 introduces a notation that allows abbreviating chosen settings in a convenient way. The base setting is `00000`, meaning that `bisect_all` is chosen as the splitting heuristic and all other features are disabled. Using this notation, we can immediately see that there are $2 \cdot 3 \cdot 2 \cdot 2 \cdot 2 = 48$ different settings. However, not all of these settings are sensible. The right most digit is often implied by the other ones. Under these restrictions, 32 sensible combinations are left. As this is still too much to evaluate all of them on all benchmarks, we proceeded in the following way: First, we picked 10 out of the 2373 benchmarks to run all 32 sensible combinations. We then preselected a smaller set of suitable combinations to run all benchmarks on them.

Section 5.3 will give more details on this process through presenting and analyzing its results.

Comparability of Runs

In general, it is not possible to compare the running time of two runs with different settings: Assume two runs have used a different splitting heuristic. Then, for a given depth, the combined volume of the boxes which are still labeled unknown may be larger in one of the runs than for the other. Just comparing running times would thus not be fair, as a solution in which this volume is smaller is considered better. Fixing one of the two splitting heuristics, running times become comparable, as all other features do not lead to differences in the area covered at a certain depth. They are only able to influence the time needed to get to this depth.

As a consequence, when only comparing running times of different settings, we handle the two different splitting heuristics separately. To compare them we need diagrams with running time on one axis and the share of classified area on the other.

Compensating Depth Differences The splitting heuristic `bisect_single` only splits a box w.r.t. one dimension per iteration. This means that the size of the smallest boxes in a specific run is larger than if `bisect_all` would have been used. This has multiple effects: On the one hand, the unknown area can not be smaller than when `bisect_all` would have been used because the resolution is not as fine. On the other hand, far less solver calls are made when using `bisect_single` because far less splits and thus far less boxes are checked.

To compensate for this, before the execution, we multiply the maximal depth of a run using `bisect_single` with the dimension of the respective formula. With other words, we allow `bisect_single` to iteratively split boxes until the minimum box size achievable under `bisect_all` is reached. As a result, the size of the smallest boxes which are still labeled unknown is then the same for `bisect_single` and `bisect_all` for all depths. When naming a specific depth in context of a `bisect_single` run in the following, we actually used this depth scaled with the dimension of the respective formula in order to ensure similar running times for the same depth.

	Depth		
	4	5	6
02101	0.76	0.90	0.80
02000	0.88	0.94	0.96
02111	0.79	-	0.97
00000	1.00	1.00	1.00
01111	0.99	-	1.03
01101	0.96	1.05	1.06
00111	0.98	-	1.06
01000	0.98	0.98	1.08
00101	0.95	1.11	1.11
01001	1.12	1.11	1.12
01100	1.20	1.17	1.23
02010	1.96	-	1.29
00010	2.11	-	1.36
01011	2.27	-	1.46
01010	2.13	-	1.46
01110	2.30	-	1.62

	Depth		
	4	5	6
12101	0.68	0.63	0.58
12111	0.77	-	0.65
11101	0.85	0.83	0.80
10111	1.04	-	0.81
12000	1.11	0.93	0.82
12010	1.21	-	0.84
11000	0.91	0.89	0.87
11111	0.92	-	0.87
10101	1.14	1.03	0.89
11010	1.00	-	0.89
10010	1.07	-	0.91
10000	1.00	1.00	1.00
11001	1.08	1.06	1.02
11011	1.18	-	1.07
11100	1.16	1.12	1.09
11110	1.26	-	1.12

Table 5.1: Average relative running times in the preselection runs. The columns represent different depths. As described in Section 5.3.1, the two splitting heuristics are handled separately. The rows of both tables are sorted ascending by the values in the column for depth 6. The relative running times are calculated in comparison to the respective base-settings `00000` and `10000`, which are marked blue (●). For each splitting heuristic, the setting with the best average relative running time is marked green (●). Note that incremental solving is applicable at even depths; we denote non-applicability by "-".

5.3 Experimental Results and Running Time Analysis

5.3.1 Preselection Process

We first evaluated all sensible setting combinations on 10 selected benchmarks. As explained in Section 5.2, we handle the different splitting heuristics separately. To ensure comparability, results of all benchmarks that have not reached depth 6 for all settings are discarded. For `bisect_all` we did not have to discard results, for `bisect_single`, however, we discarded results from two benchmarks. Runs with depth 0-3 are discarded as well because the running times were regularly below five milliseconds and thus too short to give meaningful results. Due to the low number of benchmarks which reached depth 7 or higher, we discarded these depths too.

Table 5.1 depicts the results of the preselection runs. There are two interesting phenomena to observe.

First, the standard setting for `bisect_all` performs relatively good (fourth best) compared to other settings using the `bisect_all` splitting heuristic. In contrast,

	Depth								
	4	5	6	7	8	9	10	11	12
	1910	1910	1910	1843	1319	922	732	448	306
02000	0.85	0.96	0.91	0.98	0.91	1.04	0.95	1.03	0.97
00000	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
02101	1.57	1.33	1.01	1.03	0.93	1.04	0.91	0.88	0.78
01000	1.02	1.01	1.03	1.03	1.05	1.10	1.10	1.09	1.09
01001	1.12	1.12	1.17	1.15	1.17	1.23	1.23	1.17	1.17
00101	1.92	1.39	1.22	1.19	1.16	1.13	1.09	0.92	0.89
00010	1.39	-	1.48	-	1.1	-	1.07	-	1.03

Table 5.2: Average relative running times in the final evaluation runs. The columns represent different depths. The rows are sorted ascending by the values in the column for depth 6. The relative running times are calculated in comparison to the base-setting `00000`, which is marked blue (●). The setting with the best average relative running time is marked green (●). Note that incremental solving is applicable at even depths; we denote non-applicability by "-". The number below each depth indicates the number of benchmarks which have reached this depth for all combinations and are thus used to calculate the respective relative running time.

the standard setting for `bisect_single` performs quite bad compared to other settings using the `bisect_single` splitting heuristic. Consequently, the implemented features have a much more positive impact on the running times when using `bisect_single` compared to `bisect_all`.

Second, independent of the splitting heuristic, the `x2101` setting performed best. As explained in Figure 5.1, `x2101` means that the `clever` sampling heuristic was chosen, and model saving and sample splitting were enabled. We thus decided to further evaluate settings `02101` and `12101` in addition to the settings evaluating a single feature. As a result, we evaluate the following settings on all benchmarks:

- `00000`
- `00010`
- `00101`
- `01000`
- `01001`
- `02000`
- `02101`
- `10000`
- `12101`

5.3.2 Extensive Evaluation

In this section, we evaluate the results from running all 2373 benchmarks on the preselected combinations. However, to improve comparability, we further removed all benchmarks which did not reach depth 6 for all combinations, which left 1910 benchmarks.

General Running Time

Equivalently to Table 5.1, we calculated the average relative running times for all 1910 benchmarks. Tables 5.2 and 5.3 depict the results. As in Table 5.1, runs with depth 0-3 are discarded due to very short running times.

	Depth								
	4	5	6	7	8	9	10	11	12
	1910	1910	1910	1578	1008	763	568	290	201
12101	0.79	0.77	0.83	0.71	0.64	0.61	0.59	0.53	0.44
10000	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 5.3: Average relative running times in the final evaluation runs. The columns represent different depths. The rows are sorted ascending by the values in the column for depth 6. The relative running times are calculated in comparison to the base-setting 10000, which is marked blue (●). The setting with the best average relative running time is marked green (●). The number below each depth indicates the number of benchmarks which have reached this depth for all combinations and are thus used to calculate the respective relative running time.

In general, the implemented features performed better in the preselection runs than in the evaluation runs on all benchmarks. For the bisect_all splitting heuristic, only combination 02000 performed better than the base setting on average. All other combinations performed comparable to or worse than the base setting. For the splitting heuristic bisect_single, combination 12101 did outperform the base setting. Also, the relative running times for combination 12101 are significantly better than all observed relative running times for the bisect_all splitting heuristic.

For a specific combination, relative running times do vary over the depth, however, these variations are mostly negligible. Additionally, from depth 6 onward, relative running times were calculated using a decreasing number of benchmarks, as slow benchmarks did not reach this depth. Comparison of different depths is thus only possible to a limited extent.

Additionally to the relative running times, Figures 5.2 and 5.3 show box-plots of the relative running times for depth 6. Remarkably, the relative running times have a wide range for a given combination. For over 50% of the benchmarks, combinations 00010, 02000, 02101, and 12101 outperform their respective base setting.

It is unjustified to simply say that the implemented features did not improve running times in general. Whether the features improve running time or not is very dependent on the input formula.

In the following sections, the different heuristics and their performance are evaluated in detail.

Sampling Heuristics

To further investigate running times of different combinations, Figures 5.4 and 5.5 depict the running time distribution of all evaluated combinations. Additionally, Tables 5.4 and 5.5 list the share of solver calls that are potentially preventable and the share of solver calls that have actually been prevented (through a sample).

In general, Figures 5.4 and 5.5 clearly show that independent of the combination, the time spent on solving accounts for the majority of the running time.

In the base case, additionally to the solving time, a relatively small share is spent splitting the boxes. When using bisect_single as a splitting heuristic, a box is split in less dimensions. As a consequence, the share of time spent splitting boxes is lower for the base case in Figure 5.5 than for the base case of Figure 5.4.

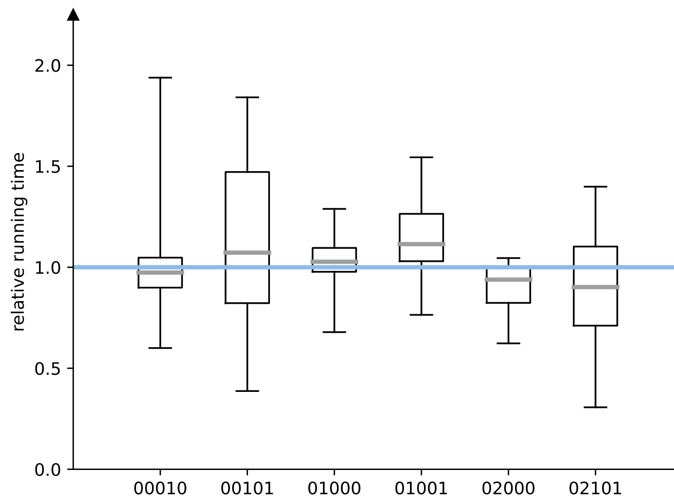

Figure 5.2: Box-plots of the relative running time for depth 6 and splitting heuristic `bisect_all`. The whiskers mark the 5th and 95th percentile. The base setting is indicated by the blue (●) line.

	00000	00010	00101	01000	01001	02000	02101
Preventable	0.55	0.55	0.55	0.55	0.55	0.55	0.55
Prevented	0.00	0.00	0.17	0.29	0.31	0.29	0.39

Table 5.4: Solver call prevention statistics for depth 6. The table lists the share of solver calls that are potentially preventable and the share of solver calls that have actually been prevented (through a sample). The base-setting is marked blue (●), the setting which was able to prevent the the largest share of solver calls is marked green (●).

The unspecified time includes initializations, if-cases, switches and more, which are inevitable overhead introduced through the algorithm itself. As a consequence, this share of time is similar for all tested feature combinations.

Tables 5.4 and 5.5 show another notable phenomenon: When using `bisect_all`, on average, only 55% of the solver calls were preventable; using `bisect_single` made 83% of the calls preventable. A call is not preventable, if it classifies a box as satisfying or unsatisfying, as this classification is impossible using only samples.

Center Taking a closer look at the sampling heuristic center (combinations `01000` and `01001`), we can immediately see in Figure 5.4, that sampling introduces a significant amount of overhead. However, Table 5.4 shows the benefit of doing so: Around 30% of all solver calls are prevented, which leads to a decrease in solving time. However, it is also clearly visible that this decrease is not the 30% one might have hoped for.

Dividing the samples between child boxes when splitting a box (sample splitting)

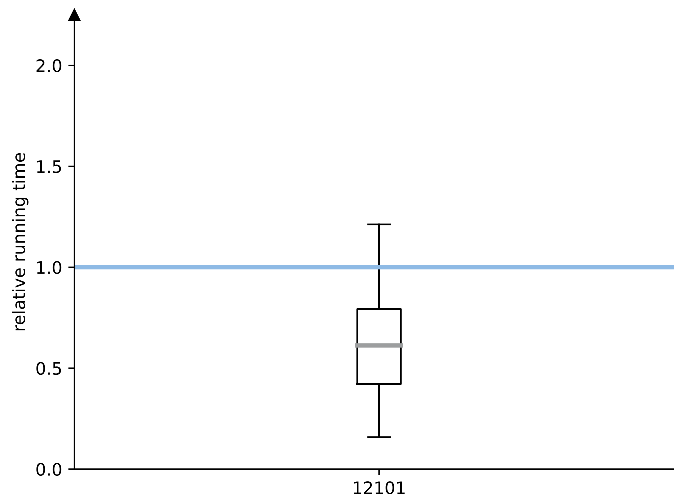

Figure 5.3: Box-plot of the relative running time for depth 6 and splitting heuristic `bisect_single`. The whiskers mark the 5th and 95th percentile. The base setting is indicated by the blue (●) line

	10000	12101
Preventable	0.83	0.83
Prevented	0.00	0.72

Table 5.5: Solver call prevention statistics for depth 6. The table lists the share of solver calls that are potentially preventable and the share of solver calls that have actually been prevented (through a sample). The base-setting is marked blue (●), the setting which was able to prevent the the largest share of solver calls is marked green (●).

does only prevent 2 additional percent of solver calls (Table 5.4), but introduces significant overhead when splitting.

Clever Clever sampling (combination `02000`) introduces much less overhead than center sampling through a simple trick: Because the cuts made to split a box are all going through the sample, we know that all resulting boxes contain this sample. For these resulting boxes, we thus do not introduce additional splitting time. Additionally, we already know one sample for every child box. We make use of this through taking samples only every other depth. For d given dimensions, this means that instead of taking 1 sample for a box and 2^d for its child boxes, we only take 1 for this box *and* its child boxes. Table 5.4 shows that this is enough to prevent almost 30% of the solver calls.

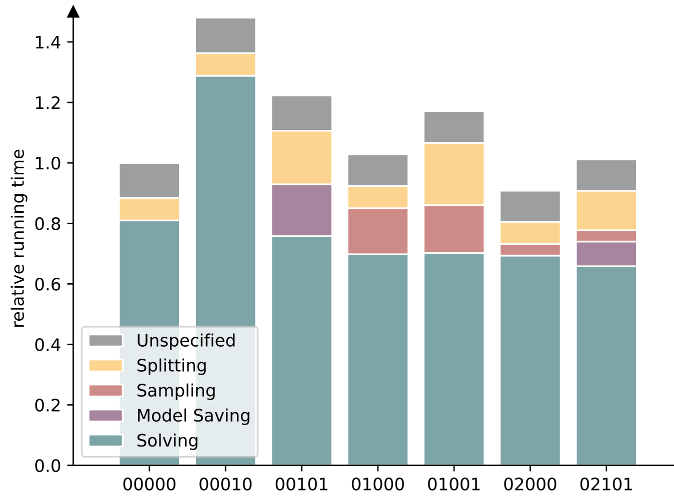

Figure 5.4: Relative running time distributions for depth 6 and splitting heuristic `bisect_all`.

Model Saving Table 5.4 shows that, on average, 17% of the solver calls can be prevented through saving the models of all solver calls returning "satisfiable" when using `bisect_all` (combination `00101`). However, model saving does introduce a substantial amount of overhead through the model saving itself but also through the then necessary sample splitting (Figure 5.4). Although, on average, this leads to a substantial increase in running time, Figure 5.2 shows that model saving does improve running time for a notable number of benchmarks.

Combination `02101` also uses model saving. As (compared to `00101`) less solver calls are made through clever sampling (Figure 5.4), the introduced splitting and model saving overhead is lower than in `00101`. On average, model saving is nonetheless not beneficial when using clever sampling (Table 5.2); however, Figure 5.2 also indicates a lower mean relative running time for `02101` than for `02000`.

Solver Calls and Solving Time Dependency Throughout the evaluation of all features, one very notable phenomenon can be observed: Tables 5.4 and 5.5 also show that the features were able to prevent a substantial share of all solver calls, in the case of `12101` even over 70%. However, Figures 5.4 and 5.5 clearly show that although the number of solver calls was reduced substantially, the total solving time is not reduced by the same portion.

In the scope of this thesis, we were not able to find the reason for this phenomenon with absolute certainty. Nonetheless, the following paragraph provides a possible explanation.

The solving times of the solver calls that have been prevented could be much smaller than the ones which could not be prevented. As mentioned above, a solver call is preventable only if the call is preventable through a sample, meaning if the formula is satisfiable. Consequently, only these solver calls are prevented. All solver

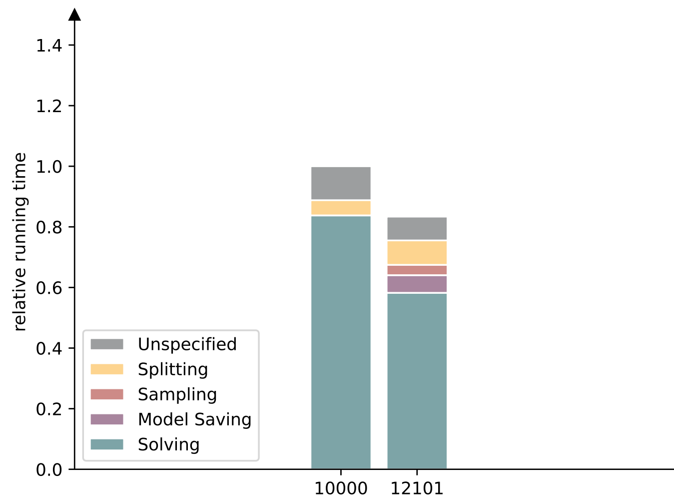

Figure 5.5: Relative running time distributions for depth 6 and splitting heuristic `bisect_single`.

calls for which the result is "unsatisfiable", can not be prevented. Assuming a solver is terminating faster on satisfiable instances, the above phenomenon becomes explainable.

Splitting Heuristics

As described in Section 5.2, it is impossible to directly compare running times of combinations with different splitting heuristics, as, even for the same depth, the classified area can differ. Therefore it is necessary to use a diagram that considers both, running time and the classified area. Figure 5.6 does exactly that.

When looking at the base settings, `bisect_single` seems to be outperformed by `bisect_all`. However, as already observed in Section 5.3.2, the relative running time for `12101` is significantly better than all observed relative running times for the `bisect_all` splitting heuristic. This is also clearly visible in Figure 5.6. In general, setting `12101` does seem to be the best performing setting overall.

As a consequence, it is not possible to conclude that one splitting heuristic is better than the other. Although the base setting of `bisect_single` is performing worse in the base setting, it is responding better to other implemented features than `bisect_all`. In the end, it is dependent on these other features which splitting heuristic is performing better.

The fact that the `bisect_single` is reacting better to other implemented features, may be explained by comparing Table 5.4 with Table 5.5: The share of preventable solver calls is much higher using `bisect_single`. In case these solver calls are prevented, also the solving time and thus running time is able to drop by a higher amount compared to using `bisect_all`.

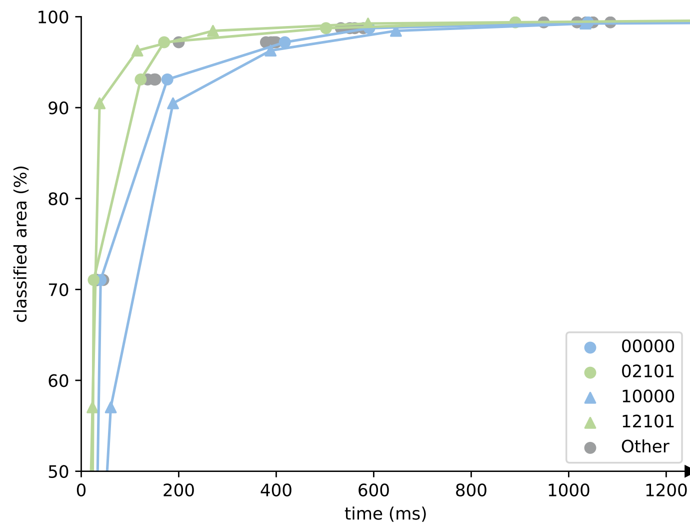

Figure 5.6: Development of the classified area over time. The classified area is the share of the combined area of all boxes which are not marked unknown. One data point is calculated through averaging running time and classified area for a specific depth and feature combination.

Incremental Solving

Table 5.2 shows that the average running time does not benefit from using incremental solving, in particular for depth 6. Although the average slowdown decreases for higher depths, incremental solving did not improve the average running time for any depth. In contrast, Figure 5.2 shows, that in over 50% of the cases the running time did decrease using incremental solving. It is thus very dependent on the benchmark, whether using incremental solving is sensible.

Comparison to PaSyPy

Unfortunately, the evaluation of PaSyPy has only been done through a case study. Additionally, the computer use by Wiegel to perform the case study has much better specifications than the one used in this work. Comparing both implementations is thus no really possible without significant effort. Nonetheless, on very simple examples, our implementation was up to 50 times faster. On more complex examples, where less boxes were created and checked for satisfiability, this factor dropped, because the pure Z3 solving time is probably comparable in both implementations. This implies that the Python overhead in PaSyPy is large and the switch to C++ was a sensible decision.

Other Notable Findings

A small case study indicated that, the number of free variables and thus the dimension has an enormous impact on the resulting running time. In Section 3.6.2, we explained

that general quantifier handling would be achievable through removing the quantifier and treat the variable as free variable. As this would result in an increase of the dimension, it should be noted that the idea described in Section 3.6.2 would thus most probably result in a serious increase in running time and is thus only feasible for a very limited number quantifiers. An approach that might be more scalable is an extended support of quantifiers by the underlying solver.

Chapter 6

Conclusion

In this chapter, we conclude the work done in this thesis. First, in Section 6.1, we summarize the work done before we shortly discuss it in Section 6.2. In Section 6.3, we finally give an outlook on open problems and possible topics for future work.

6.1 Summary

In Chapter 1, we gave a brief introduction to the topic, explained how this theses was built on the work of Wiegel, and shortly sketched related work [Wie21]. In Chapter 2, we then provided the prior knowledge necessary to understand the theoretical part of this thesis from Chapter 3. In this chapter, we explained in detail the base algorithm, implemented heuristics to improve the running time of this algorithm, and also solved the issue of quantifier handling on a theoretical level. In Chapter 4, we gave an overview of the implementation, which we evaluated experimentally in Chapter 5.

6.2 Discussion

Through the reimplementation of the base algorithm, this thesis was able to resolve the major flaws of PaSyPy. In particular, the implementation described in this thesis uses exact arithmetic. For formulas on which Z3 has a good performance, this thesis presented a well performing program for parameter synthesis, whose results are accessible through a GUI.

The experimental evaluation in Chapter 5 made very clear, that the performance of the implemented heuristics is very benchmark dependent. While some of the implemented heuristics introduce too much overhead to be useful in most cases, other heuristics are improving the running time for the majority of benchmarks. Combining different heuristics did further decrease the running time for most benchmarks. A brief running time comparison to PaSyPy justified the switch from Python to C++.

Unfortunately, it was not possible to also implement quantifier handling in the scope of this thesis.

6.3 Future Work

As the quantifier handling presented in Section 3.6 has not been implemented, one possible topic for future work is implementing the suggested method.

Also, as explained in Section 4.1, only Z3 is supported as underlying SMT solver by the current implementation. It may also be interesting to test other solvers. However, this would require altering a substantial part of the code.

In Section 5.3.2, the experimental evaluation showed that the reduction of solver calls is not correlating linearly with the reduction of solving time. This can set a focus for future work on this topic: First, is it possible to confirm the assumption that in our case the prevented solver calls on satisfiable instances were taking less time than the unprevented ones on unsatisfiable instances? If so, is this a property of the solver, the benchmarks, both, or something completely different? If it turns out, that satisfying instances generally perform better, the focus on more sophisticated sampling heuristics can only help to a very limited extent. Our implementation was already able to prevent over 85% of the preventable solver calls (Table 5.5). Additionally, the sampling or model saving necessary to do so introduced a non-negligible amount of running time overhead. Future work would thus have to focus on reducing the number of non-preventable solver calls. This directly implies optimizing the splitting heuristic. In order to find a reasonable cutting axis, it is necessary to take samples. We have seen in Figure 5.4, that taking one sample per box already introduces significant overhead. A necessary condition for finding good splitting heuristics would thus be that they are only using a very low number of samples, at best only two or three. Although these observations clarify that further research into this topic needs complex approaches, it is most probably worth the effort.

Bibliography

[Ábr20] Erika Ábrahám. *Lecture Notes in Satisfiability Checking.* Theory of Hybrid Systems Group, RWTH Aachen University, https://ths.rwth-aachen.de/teaching/ws19/lecture-satisfiability-checking/. 2020.

[Boo] Boost. *Boost C++ Libraries.* Retrieved Febuary 20, 2022 from http://www.boost.org/.

[BST10] Clark Barrett, Aaron Stump, and Cesare Tinelli. "The SMT-LIB Standard: Version 2.0". In: *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories.* Vol. 13. 2010, p. 14.

[Deh+15] Christian Dehnert et al. "PROPhESY: A PRObabilistic ParamEter SYnthesis Tool". In: *International Conference on Computer Aided Verification (CAV'15).* Springer. 2015, pp. 214–231.

[Gra] Torbjörn Granlund. *The GNU Multiple Precision Arithmetic Library.* Retrieved Febuary 20, 2022 from http://gmplib.org/.

[Grä21] Erich Grädel. *Lecture Script in Mathematische Logik.* Lehr- und Forschungsgebiet Mathematische Grundlagen der Informatik (Logik und Komplexität), RWTH Aachen University, https://logic.rwth-aachen.de/Teaching/MaLo-SS21/index.html.de. 2021.

[Jun+19] Sebastian Junges et al. *Parameter Synthesis for Markov Models.* arXiv preprint arXiv:1903.07993. 2019.

[MB08] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08).* Springer. 2008, pp. 337–340.

[Rad] Nicolai Radke. *cppasy.* https://github.com/nicolai9135/cppasy.

[SMT] SMT-LIB. *SMT-LIB QF-NRA Benchmarks.* https://clc-gitlab.cs.uiowa.edu:2443/SMT-LIB-benchmarks/QF_NRA.

[Wie21] Alexander Wiegel. *A python-based tool using parameter synthesis to find safe and unsafe regions of the parameter space.* RWTH Aachen University, https://ths.rwth-aachen.de/wp-content/uploads/sites/4/wiegel_bachelor.pdf. Master's Thesis. 2021.

[wxW] wxWidgets. *wxWidgets.* https://github.com/wxWidgets/wxWidgets.