

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

BACHELOR OF SCIENCE THESIS

**A PYTHON-BASED TOOL USING PARAMETER
SYNTHESIS TO FIND SAFE AND UNSAFE REGIONS OF
THE PARAMETER SPACE**

Alexander Wiegel

Examiners:

Prof. Dr. Erika Ábrahám

Prof. Dr. Thomas Noll

Aachen, 18.09.2021

Abstract

Parameter synthesis raises the question of which parameter values lead to the satisfaction of a logical formula. To directly address this problem we do not want to know if a logical formula is satisfying or unsatisfying but instead, we are interested in which combination of parameters satisfy or does not satisfy the formula. This is especially interesting for *satisfiability modulo theories (SMT) solving* when considering specific intervals for different parameters. The problem with nonlinear real arithmetic is to find solutions or counterexamples in systems of equalities or inequalities. Combining both problems leads to a satisfiability problem of high theoretical interest. This thesis presents a Python-based tool to find and visualize satisfying (safe) and unsatisfying (unsafe) regions of the parameter space on nonlinear real arithmetic (NRA).

Acknowledgements

First of all, I want to thank Prof. Dr. Erika Ábrahám for giving me the opportunity to write this thesis. Especially I am thankful for her continuous feedback, motivation and huge expertise which helped me developing the corresponding tool *PaSyPy* [1]. I also want to thank Prof. Dr. Thomas Noll for being my second supervisor.

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.

Alexander Wiegel
Aachen, den 18. September 2021

Contents

1	Introduction	9
1.1	Related Work	9
1.2	Outline	10
2	Preliminaries	11
2.1	Real Arithmetic	11
2.2	Satisfiability Modulo Theories	13
2.3	Parameter Synthesis	15
2.4	SMT-LIB	15
2.5	Python	16
3	The Synthesis Algorithm	17
3.1	Find Safe and Unsafe Regions	17
3.2	Splitting	19
3.3	Sampling	22
4	PaSyPy	27
4.1	Functionality	28
4.2	Supported Solvers	33
4.3	Validation	34
4.4	Installation and Usage	34
4.5	Known Challenges	34
5	Other Tools	35
5.1	PROPhESY	35
5.2	pySMT	35
5.3	rise4fun	35
5.4	Similarities and Differences	36
6	Case Study	37
7	Conclusion	49
7.1	Summary	49
7.2	Future Work	49
	References	51
	Appendix	53

A	Further Tool Explanation	53
A.1	Buttons	53
A.2	Information Fields	55
A.3	Settings	56
A.4	Miscellaneous	57

Chapter 1

Introduction

The tool developed during this thesis, namely *PaSyPy*, uses parameter synthesis to find satisfying (safe) and unsatisfying (unsafe) regions of the parameter space for a logical formula. This is done with satisfiability modulo theories (**SMT**) solving on nonlinear real arithmetic (**NRA**) formulas, with addition of a given domain for every parameter.

This tool goes through three basic steps:

- Satisfiability modulo theories (SMT) solving on nonlinear real arithmetic (NRA).
- Adding boundaries to the parameters tackling the parameter synthesis problem.
- Visualizing the found safe (green) and unsafe (red) regions.

The combination of all these steps is already what makes this tool unique and stand out from other tools and is discussed more in Section 5.4.

1.1 Related Work

While there are several tools and techniques available for satisfiability modulo theories (SMT) solving, only a small part of them is capable of proving whole regions as safe or unsafe. To realize this, we need to add boundaries to our parameters represented by a box, being the Cartesian product of intervals, one for each parameter. Often not taken into consideration is the parameter synthesis problem which is described in Section 2.3. An approach on parameter synthesis for Markov models was done in [2][3], where a tool was introduced, namely *PROPhESY* [4], for analyzing parametric Markov chains. As an input *PROPhESY* expects a rational function which is a fraction of two polynomials representing model parameters. This rational function is then computed to find satisfying (safe) and unsatisfying (unsafe) regions which can then be visualized with the help of a featured web front-end. Different to *PROPhESY* instead of a rational function, our tool expects a logical formula with parameters on nonlinear real arithmetic.

Because quantifier elimination is automatically performed by the underlying solver we used, i.e., the *z3 theorem prover* [5] and its Python interface, we can not guarantee its correctness nor its efficiency and will mostly stick to quantifier-free nonlinear real arithmetic (QF NRA), even though our tool was able to solve formulas containing multiple quantifiers. The goal of this thesis is to develop and implement a pure Python-based tool, which uses satisfiability modulo theories (SMT) solving for nonlinear real arithmetic to find safe and unsafe regions with consideration of the parameter synthesis problem.

1.2 Outline

In this thesis, a Python-based tool named *PaSyPy* was developed, which uses parameter synthesis to find safe and unsafe regions of the parameter space.

Chapter 2 introduces the underlying theoretical and technical background of this tool, necessary for an understanding of how this tool works and what it can be used for.

Chapter 3 summarizes the underlying problems and explains the methodology especially the algorithms used by this tool to solve these problems.

Chapter 4 is dedicated to this tool, specifically all functionalities, the graphical user interface (GUI), the currently supported solvers, how this tool is validated, a manual for installation and usage and known challenges that appeared during development with problems and potential solutions.

In Chapter 5 other tools are considered and compared with their functionality and performance.

In Chapter 6 this tool is tested on various benchmarks. We will use different logic and theories to illustrate the tool's capabilities and to verify that this tool provides the correct results. We will analyze the time needed by the solver to find the safe and unsafe regions and the time needed to visualize the graph. Different parts of this tool will be considered here. Also at some point, all currently implemented heuristics are used and compared to each other. To complete the case study we will use our tool on different benchmarks from SMT-LIB [6].

The last Chapter 7 summarizes the perception obtained after developing this tool and also gives a quick outlook on how this tool can be further extended and improved.

Chapter 2

Preliminaries

This chapter provides the basic knowledge and background necessary to understand how this tool works.

2.1 Real Arithmetic

Real arithmetic, often also known as nonlinear real arithmetic (NRA) is a first-order logic theory $(\mathbb{R}, +, \cdot, 0, 1, <)$ over the real numbers (\mathbb{R}) together with addition (+), multiplication (\cdot) and the comparison operator smaller ($<$). [7, Ch. 2]

The syntax of real algebra is defined as follows:

Terms:	t	::=	0		1		x		$t + t$		$t \cdot t$
Constraints:	c	::=	$t < t$								
Formulas:	φ	::=	c		$\neg\varphi$		$\varphi \wedge \varphi$		$\exists x\varphi$		

where $x \in \text{Var}(\varphi)$ is a variable.

As syntactic sugar, additional relations and boolean functions can be derived based on the already defined operators, such as $(t_1 \leq t_2)$, $(t_1 = t_2)$, $(t_1 \neq t_2)$, $(t_1 \vee t_2)$, etc.

Nonlinear real arithmetic (NRA) is a theory that may contain nonlinear constraints, i.e., constraints or more specifically polynomials p with degrees larger than one. We are most interested in quantifier-free nonlinear real arithmetic (QF NRA), which are all formulas without quantifiers. Note that our tool also allows using quantification and was able to solve such formulas.

A term is a coefficient $a \in \mathbb{R}$ times a monomial. A monomial is a product of variables $\{x_1, \dots, x_m\}$ with non-negative exponents e . A sum of terms is then called a **polynomial** p , which we can rewrite in quantifier-free nonlinear real arithmetic (QF NRA) as follows:

$$p := \sum_{i=0}^n a_i \cdot \prod_{j=1}^m x_j^{e_{ij}}$$

Free and Bound Variables

Free variables are all variables that are not bound by any quantifier. In contrast, a variable can be bound by a quantifier to a specific domain or universe.

$$\begin{aligned} \text{Free} &: (x > 0.25) \wedge (x < 0.75) \\ \text{Bound} &: \exists x((x > 0.25) \wedge (x < 0.75)) \end{aligned}$$

If the quantifier is in the outer scope it bounds the inner scope variables. Quantifiers in an inner scope cannot bound variables from an outer scope. Note that the same variable in a formula can be both bound and free, e.g., $(x > 0.25) \wedge \exists x(x < 0.75)$ where x is both bound and free. Our parameters are exactly all free variables. Bound variables are internally considered in the solving process but we are not interested in their values.

Satisfaction Relation

The problem with nonlinear real arithmetic is to find solutions or counterexamples in systems of equalities or inequalities. This means that to satisfy a formula in nonlinear real arithmetic we have to find a combination of all parameters that satisfy the system of equalities or inequalities. Such system can consist of multiple equalities or inequalities that are connected by logical connectors:

$$\begin{aligned} \text{And} &: \wedge \\ \text{Or} &: \vee \end{aligned}$$

The formula satisfies if and only if we find a combination of values for our parameters so that all clauses connected by \wedge hold and at least one clause connected by \vee holds.

Hyperrectangle

Because we want to address the parameter synthesis problem (see Section 2.3), we have to restrict all parameters which are all free variables. For this, every parameter is restricted by a closed interval which includes the borders:

$$[a, b] = \{x \in \mathbb{R} \mid a \leq x \leq b\}$$

where $a, b \in \mathbb{R}$. A hyperrectangle or a box is the Cartesian product of intervals where each interval is assigned to a different parameter. We then get a multidimensional rectangle with one interval for each unique parameter.

2.2 Satisfiability Modulo Theories

The boolean satisfiability problem or propositional satisfiability problem (SAT) is the most basic form of satisfiability problems and checks a single boolean formula φ for satisfiability. SAT is the first problem proven to be NP-complete by Cook [8] and Levin [9]. That implies several decision and optimization problems being at most as difficult to solve as SAT. Therefore this problem is of theoretical importance and provides a central starting point for most problems.

An extension to the basic boolean satisfiability problem (SAT) is the satisfiability modulo theories problem (SMT) which is a decision problem based on first-order logic with respect to a given theory, e.g., NRA as defined in Section 2.1. This means that single constraints are connected by logical operators, i.e., the operators defined in Section 2.1. SMT-LIB (see Section 2.4) describes a large part of those theories. Different from the satisfiability modulo theories problem (SMT) where exact values are checked for different variables, *PaSyPy* checks a whole interval of values for all variables yielding areas classified into three different categories which are described in Section 3.2.

Based on the theory and logic used there are two different approaches for solving satisfiability modulo theories (SMT) problems. [10, Ch. 3]

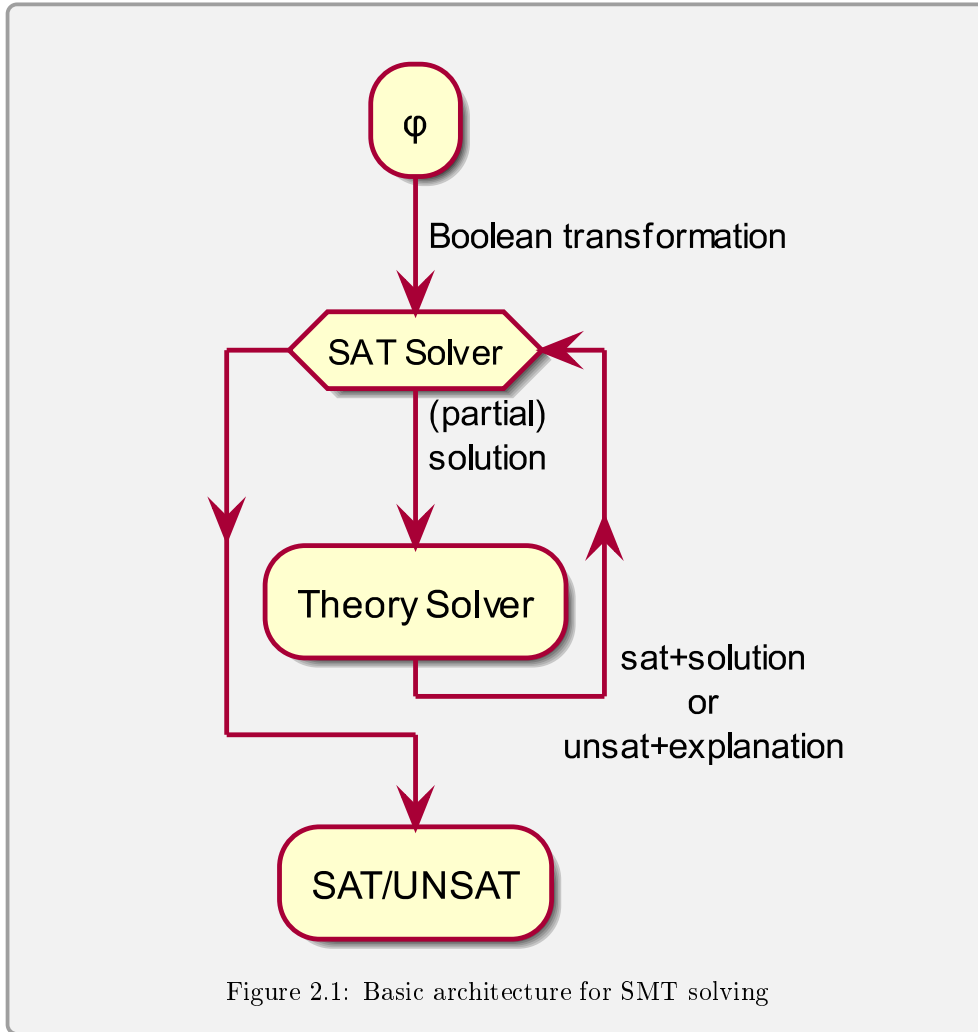
Eager SMT Solving

The *eager* approach is commonly used for theories whose formulas can be naturally encoded in propositional logic. It is well-suited for the theories of quantifier-free linear integer arithmetic (QF LIA), bit-vector arithmetic and uninterpreted functions. The *eager* SMT solving approach is similar to the polynomial reduction where a problem gets reduced to a problem with an already known solution. For this, all formulas from a given logic are transformed to satisfiability-equivalent propositional logic formulas which are then checked by basic SAT-solvers for satisfiability.

Because we are mostly interested in nonlinear real arithmetic (NRA) logic where the *eager* approach is not very effective, we need to follow another approach for SMT solving.

Lazy SMT Solving

The *lazy* approach is more powerful than the *eager* approach and combines a basic SAT-solver with a theory-solver. The basic architecture needed for SMT solving is shown in Figure 2.1. The input formula φ over some logic is abstracted to a boolean formula which is processed by a SAT-solver. Here every theory constraint is represented by a boolean variable. The SAT-solver provides a partial solution which is then further processed by a special theory solver that is suited for the used logic. The theory solver checks the given Boolean solution for consistency in the theory and returns the answer to the SAT-solver which is either *sat* with a solution or *unsat* with an explanation. Before returning an *unsat* solution the theory solver tries to refine its abstraction to return a proper explanation which the SAT-solver uses to search for further solutions. The SAT-solver finally returns whether the given formula is *sat* or *unsat*. Our tool adds a box B to the original input formula φ which assigns boundaries for each parameter.



Incremental Approach

An incremental approach in satisfiability modulo theories (SMT) solving tries to reduce overall solving cost by grouping similar constraints together. Also on incremental solving the solver can take advantage of previous queries. The *z3 theorem prover* [5] can automatically apply incremental solving. Our tool gives the formula to the solver and the negated formula to the other solver once at the beginning and then only changes the interval borders for our parameters. Resetting the whole formula on every step yields a much worse performance compared to the incremental approach of only changing the boundaries.

2.3 Parameter Synthesis

Parameter synthesis raises the question of which parameter values lead to the satisfaction of a logical formula. To directly address this problem we do not want to know if a logical formula is satisfying or unsatisfying but instead, we are interested in which combination of parameters satisfy or does not satisfy the formula. This means that for the parameter synthesis problem all parameters with all their possible values have to be taken into consideration. Each parameter represents a property inside the overall system. Often the parameter synthesis problem tries to find optimal solutions which we are not interested in here. The feasible area shows the combination of values for all parameters which satisfy the given formula. In contrast, the infeasible area shows the combination of values for all the parameters which do not satisfy the given formula.

Solving parameter synthesis is hard to realize since this problem is not trivial by having a potentially infinite number of different combinations with capable dependencies. There are different approaches to solving the parameter synthesis problem. Most approaches have been developed for Markov models as explained in [2].

One iterative approach which we will use here to solve the approximate synthesis problem is parameter space partitioning. [2, Ch. VIII] An exact synthesis problem would mean that we get a complete solution including all possible values for all parameters which is not possible for most nonlinear real arithmetic formulas because of having a potentially infinite number of different combinations. Therefore, instead of giving all possible values, we approximate the solution space. The approximation happens on the borders between satisfying (safe) and unsatisfying (red) regions where each iteration refines closer to the borders. We start with the complete region which is represented by an interval for each parameter. In case this region contains both safe (green) and unsafe (red) regions, it is split into one or more smaller sub-regions such that all sub-regions together cover the whole domain. This is recursively done in a way that each new sub-region is processed inside a queue using the FIFO principle (First In - First Out) to ensure that all regions are split to an equal depth. A region is complete and not further split when it either contains only safe (green) or unsafe (red) areas.

The problem and the approach are described in more detail in Chapter 3.

2.4 SMT-LIB

*Satisfiability Modulo Theories LIB*rary is an interface language or file format for describing *SMT* problems. The main intention with SMT-LIB was to assist the development and research in the field of SMT solving. SMT-LIB supports many different logics over many different theories. We are mainly interested in *nonlinear real arithmetic (NRA)*, which *PaSyPy* operates on. Our tool is able to read constraints from *.smt2* files and also save constraints back to *.smt2* files. SMT-LIB includes over 100.000 benchmarks classified into all different logics. All examples contained in this paper are either parsed from self-created SMT-LIB files which can be found on the project's page or were found on the official SMT-LIB benchmark repository. Figure 2.2 shows an example *.smt2* file. [6]

```
(set-logic NRA)
(set-info :source | Produced by Alexander Wiegel |)
(set-info :smt-lib-version 2.0)
(declare-fun x () Real)
(declare-fun y () Real)
(assert (exists ((z Real))
  (and
    (>= x 0.5)
    (>= y 0.5)
    (> z x)
    (< z y)
  )
))
(check-sat)
```

Figure 2.2: Example *.smt2* file

2.5 Python

PaSyPy is written in Python, a high-level general-purpose programming language. Python was first released in 1991 and is today one of the most popular programming languages. The main characteristics of Python are easy syntax and code readability which both increase the maintainability of code. One big advantage of Python is the several libraries available which this tool also takes advantage of. The used packages are:

- **z3** [5] and its Python interface *Z3Py* for every solver related action.
- **tkinter** [11] for the complete graphical user interface (gui).
- **matplotlib** [12] for plotting satisfying (safe) and unsatisfying (unsafe) regions.
- **scikit-learn** [13] which uses **numpy** [14] arrays filled with all satisfying (safe) and unsatisfying (unsafe) regions to create an approximation curve known as *Large Margin Classifier* by regression to separate the regions.

We use Python 3 [15] which is the most recent version and an extension to the original Python.

Chapter 3

The Synthesis Algorithm

With this tool, we try to tackle the parameter synthesis problem on nonlinear real arithmetic (NRA). More precisely we want to find and prove as much of the whole parameter space as feasible or infeasible as possible in a relatively short and efficient time. *PaSyPy* tries to solve this problem by separating all satisfying (safe) and unsatisfying (unsafe) regions for all possible combinations of parameters. We will separate the implementation into three parts:

- The general procedure on how safe and unsafe regions are found and proven.
- Splitting regions into smaller sub-regions.
- Optimize performance by sampling.

This method is specified in Algorithm 1. First an optional *pre-sampling* is performed. Then the queue Q which initially contains the complete region represented by intervals for every parameter is processed. The first element of the queue Q is analyzed whether it is a safe (green), an unsafe (red) or an unknown (white) region. In case it is an unknown (white) region it is split into smaller sub-regions which are then appended to the queue Q . An optional *sampling* before every split is possible to increase the efficiency of splitting. This procedure is repeated until the queue Q is either empty or a pre-defined depth is reached.

3.1 Find Safe and Unsafe Regions

When we look at single points inside a box B , every point is either satisfying or unsatisfying. For the parameter synthesis problem, we do not want to analyze single points but whole regions to get the feasibility area of the parameter space. To realize this, we replace the single point which is represented by a coordinate consisting of a number for every parameter, with an interval for every parameter. Each region then consists of an infinite amount of points based on the accuracy of the numbers, e.g., if we have the border 0.1 for x and the previous candidates 0.09 and 0.11 found by our solver, we then approach this border by refining the approximation on each step $0.09 > \dots > 0.1 < \dots < 0.11$ where the dots represent a more refined number between the border 0.1 and the previous candidates. Because we are in the range of real numbers \mathbb{R} , the refinement of the approximation yields an infinite number of new candidates.

The formula has one or multiple clear borders based on all contained constraints which divide the feasible and infeasible area. If the original region which is our starting box B contains both feasible and infeasible areas it is impossible to get a solution for the whole area with every combination of possible parameter values, but we can get a solution for a major part of it (approximately 90%+) in a matter of seconds.

Algorithm 1 General procedure for finding safe and unsafe regions

Input: φ	▷ φ is the logical formula
Input: $Q := [B]$	▷ B is a box with an interval for each parameter and a depth that represents the number of splits
Output: G, R	▷ Set of all safe (green) and unsafe (red) regions
$G := []$	▷ The satisfying (safe) regions
$R := []$	▷ The unsatisfying (unsafe) regions
(optional: pre-sample)	
while (Q is not empty) AND ($Q[0].\text{depth} \leq \text{DEPTH_LIMIT}$) do	▷ $Q[0]$ is the next box inside the queue, here we check if the global depth limit was reached
if $\varphi \wedge B \text{ sat}$ then	
if $\neg\varphi \wedge B \text{ sat}$ then	
(optional: sample)	
Split B and append to Q	▷ Depth on B is multiplied by the number of new boxes
else	
$G.\text{append}(B)$	▷ B is safe (green)
end if	
else	
$R.\text{append}(B)$	▷ B is unsafe (red)
end if	
$Q.\text{pop}(0)$	
end while	

The algorithm to find safe and unsafe regions of the parameter space uses a satisfiability modulo theories (SMT) solver to not only analyze single points for satisfiability but whole regions. For this as input, the algorithm gets the formula together with a box B which is represented by an interval for each parameter and a depth. The result either yields *sat* or *unsat* for the whole region. If the result is *unsat*, the whole region does not contain a single solution. This region is called unsafe and is represented by a red color. If the result is *sat*, the whole region contains at least one feasible solution but it is unclear if the region also contains infeasible solutions. To differentiate if the area also contains infeasible solutions this time the original formula from the input is replaced by a counterexample which is the negated version of this formula. If the result is *unsat*, the whole region exclusively contains feasible solutions. This region is called safe and is represented by a green color. Should both steps result in *sat*, the region contains both safe and unsafe areas which means there is both a feasible and infeasible solution inside this region. This region is called unknown and stays white just

as in the beginning. To extract the safe (green) and unsafe (red) areas of an unknown (white) region another algorithm is used which we call *Splitting* (see Section 3.2).

Correctness

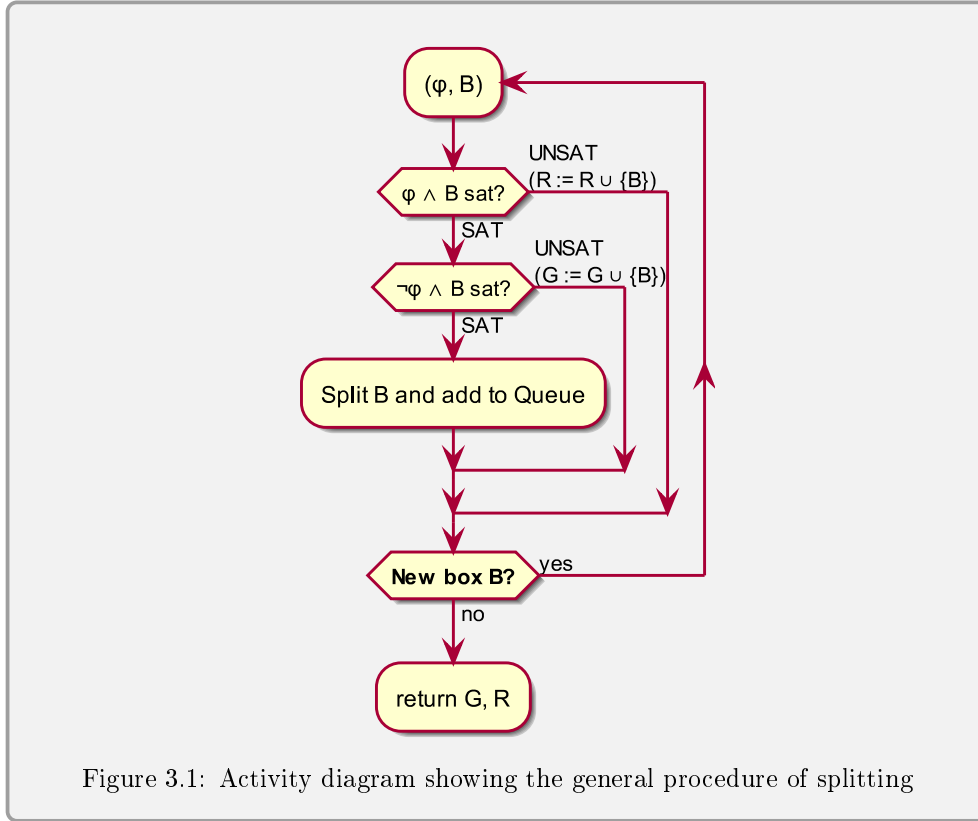
On the first step, the underlying solver internally proves if a solution for the formula within the given region exists which can be written as $\varphi \wedge B$. If not then this is already evidence for this region to be unsafe (red) which means that all counterexamples or the negated formula hold in a given region B . On the other hand, if a solution exists, this does not mean that the formula on a given region holds for all values. Therefore we try to find a counterexample by checking the satisfiability of $\neg\varphi \wedge B$. If no counterexample exists this is evidence for this region to be safe (green) which means that the formula holds on all values in a given region B . In the third case, the formula on the given box B contains both a solution and also a counterexample where we have to split the region into smaller sub-regions. This proof is done recursively on all sub-regions. \square

3.2 Splitting

When trying to find safe and unsafe regions of the parameter space there are three different outcomes:

- If the formula on viewed area is **sat** and the negated formula is **not sat**, the area is considered *safe (green)*.
- If the formula on viewed area is **not sat**, the area is considered *unsafe (red)*.
- If the formula on viewed area is both **sat** and the negated formula is **sat**, the area contains both *safe and unsafe regions (white)*.

In the last case, the viewed area has to be split into smaller regions. Splitting is really important in terms of performance, especially when regarding multiple parameters. Unfortunately because of the nature of constraints with *real arithmetic logic*, finding a good splitting heuristic is not trivial. Therefore this tool provides different heuristics for splitting unknown (white) regions. Figure 3.1 shows the general procedure when splitting is applied. In the activity diagram, B is the current box, φ is the formula, G are the safe (green) areas and R are the unsafe (red) areas. The initial box B is the whole parameter space which can be set individually for each parameter. By default this is set to $[0.0, 1.0] \in \mathbb{R}$ for every parameter. All satisfying (safe) regions are appended to the solution set G and all unsatisfying (unsafe) regions are appended to the solution set R . If the area is unknown meaning it both contains satisfying (safe) and unsatisfying (unsafe) regions, this area is split by the pre-selected split heuristic. This splits the original area into smaller sub-areas which are then added to the queue. The queue is traversed until it either is empty or the selected accuracy or depth limit is reached.



This tool uses an incremental approach which means that the main constraints from the formula are inserted once at the beginning and then only the interval boundaries are updated at each iteration. The incremental approach yields a huge performance lead over the non-incremental approach.

While all heuristics differ at some point, they all have two characteristics in common:

1. Always at the beginning the complete parameter space is taken into consideration.
2. All heuristics can use *sampling* (see Section 3.3) to determine in which dimension and where to split.

Currently *PaSyPy* provides four different heuristics:

Default

The *Default* heuristic splits every box in exactly $2^{\text{dimensions}}$ boxes. I.e., for 2 dimensions $2^2 = 4$ boxes, for 3 dimensions $2^3 = 8$ boxes and for 4 Dimensions $2^4 = 16$ boxes on every split.

Without sampling, it uses the middle point in every dimension as the point for splitting, yielding equally large boxes.

This heuristic was also used in [2, Ch. VIII: Equal splitting]

Simple

The *Simple* heuristic splits every box into two boxes, starting with the first dimension and iterating through all. I.e., the first cut is on the x -axis, the second cut on the y -axis, the third cut on the z -axis. To know which dimension we have to split next we use a counter *modulo* `NUMBER_OF_PARAMETERS`.

Without sampling it splits every dimension in the middle, yielding two equally large halves.

Extended

The *Extended* heuristic first gets a model for a white box. This is usually the first matching point found by the underlying solver. Then this heuristic checks, if splitting on the found point is possible, i.e., the point must not lie on the border. Otherwise the *Default* heuristic has to be used. In general, this heuristic operates similar to the *Default* heuristic with the difference, that no fixed point is used but the underlying solver is exploited to find an appropriate point. This is also the main problem of this heuristic as we do not have any influence on which point is found by the underlying solver. We tried to convert this into an optimization problem where we take the minimum or maximum found value for our parameters but this turned out to be only possible on formulas containing really easy constraints.

Random

The *Random* heuristic operates like the *Default* heuristic but chooses a random point between the interval on every dimension.

With sampling, it chooses a random point for splitting on the sampled area.

Other Heuristics

Because we examine on the whole region if the original formula is satisfying and also if the negated formula is satisfying, we get a model for both a satisfying point and also for an unsatisfying point. We tried to use those points as starting points to separate the satisfying and unsatisfying regions. The problem was again the underlying solver which could not find good points reliably. Also, the solver did not find the same exact points on every program execution for the exact same formula.

Correctness

The correctness of splitting is trivial and fulfilled if the new split sub-regions all together form back the original region. Therefore splitting can only be incorrect if it either adds a new area or dismisses the area from the original region which is not the case for this tool. \square

3.3 Sampling

Combining the general logic for finding safe and unsafe regions of the parameter space together with different splitting heuristics is already enough and leads to a solution where it finds satisfying and unsatisfying regions. Based on how the formula fits the used splitting heuristic, this solution and the time taken for computing might not be optimal or very efficient. For this we will use a method called *sampling* to get better candidates for *splitting*.

Because the complexity of a constraint matters we can separate the problem of *sampling* into two smaller problems, which we will call *sampling on easy constraints* and *sampling on composed constraints*.

Sampling on Easy Constraints

A formula can have a possibly infinite number of dimensions where each dimension represents one parameter. Each parameter introduces a new edge which is represented by an interval in \mathbb{R} . An easy constraint is a constraint of the form **parameter** $\{\geq, \leq, >, <, =, \neq\}$ **a** and $a \in \mathbb{R}$. We name those constraints easy because such constraints separate the feasible and infeasible regions exactly on the edge of this parameter. This can easily be *sampled* as soon as the formula is given. We call this method *Pre-Sampling*. For this, we first extract all constraints classified as easy from the formula.

We have three different classes which sample slightly different:

- $\{\geq, <\}$, which results in three intervals $[\dots, a - \epsilon]$, $[a - \epsilon, a]$, $[a, \dots]$.
- $\{>, \leq\}$, which results in three intervals $[\dots, a]$, $[a, a + \epsilon]$, $[a + \epsilon, \dots]$.
- $\{=, \neq\}$, which results in three intervals $[\dots, a - \epsilon]$, $[a - \epsilon, a + \epsilon]$, $[a + \epsilon, \dots]$.

where $\epsilon \in \mathbb{R}$ is a pre-defined low number, $a \in \mathbb{R}$ the number from the constraint and the mid-interval is a relatively small gap between accepting and rejecting regions.

Having such constraint, the affected interval can be split into three new intervals with two bigger intervals and one very small interval in between those intervals behaving like a gap. E.g., if we have the constraint $x \geq 0.5$ for x in the interval $[0.0, 1.0]$, with *Pre-Sampling* we get the new intervals $[0.0, 0.5-\epsilon]$, $[0.5-\epsilon, 0.5]$ and $[0.5, 1.0]$ for the parameter x and a low number ϵ (e.g., 0.0001). Usually, we would split the interval inside our box into two intervals $[0.0, 0.5]$ and $[0.5, 1.0]$ on this dimension. While $[0.5, 1.0]$ is an accepting region for the constraint $x \geq 0.5$, $[0.0, 0.5]$ contains both accepting and rejecting regions because the interval contains the border 0.5. Therefore, we introduced a small gap where $[0.5, 1.0]$ is an accepting region, $[0.0, 0.5-\epsilon]$ is a rejecting region and $[0.5-\epsilon, 0.5]$ is an unknown region that can then be further processed and split. With this method, we can extract most accepting and rejecting regions with a relatively small remaining unknown region. The algorithm is defined in Algorithm 2.

Algorithm 2 Pre-Sampling

Input: SIMPLE := $[[Var, [[Op, Num], \dots], \dots]$ \triangleright An array of arrays sorted by parameters (*Var*) with all inserted easy constraints filtered by operator and number of the constraints

Input: $Q := [B]$ \triangleright B is a box with an interval for each parameter and a depth that represents the number of splits

Output: Q \triangleright The queue containing all newly sampled elements

$\epsilon = 0.001$ \triangleright Smallest factor to use since solver crashes on lower numbers

for index, parameter **in** enumerate(SIMPLE) **do** \triangleright Iterate through all parameters

for candidate **in** parameter[1] **do** \triangleright Iterate through all easy constraints of a parameter

 delete = 0

for queue_index, element **in** enumerate(Q[:]) **do** \triangleright Iterate through all queue elements where the queue is copied so we do not get any problems in the process

if \triangleright Check if the border of the easy constraint is inside the element of the queue

 (candidate[1] \geq queue[index][0]) and (candidate[1] \leq queue[index][1]) **then**

if candidate[0] in $\{\geq, <\}$ **then** \triangleright [..., $a - \epsilon$], [$a - \epsilon$, a], [a , ...]

 pre_sampling.append([queue[index][0], candidate- ϵ], [candidate- ϵ , candidate], [candidate, queue[index][1]])

else if candidate in $\{\leq, >\}$ **then** \triangleright [..., a], [a , $a + \epsilon$], [$a + \epsilon$, ...]

 pre_sampling.append([queue[index][0], candidate], [candidate, candidate+ ϵ], [candidate+ ϵ , queue[index][1]])

else \triangleright [..., $a - \epsilon$], [$a - \epsilon$, $a + \epsilon$], [$a + \epsilon$, ...]

 pre_sampling.append([queue[index][0], candidate- ϵ], [candidate- ϵ , candidate+ ϵ], [candidate+ ϵ , queue[index][1]])

end if

$Q.pop(queue_index-delete)$ \triangleright Track the already deleted elements to stay on the correct index

 delete += 1

end if

end for

end for

end for

Sampling on Composed Constraints

Complex constraints are all other constraints not classified as easy constraints. This is due to those constraints not separating the feasible and infeasible regions exactly on the edge of a parameter but diagonally or curvy based on the degree of the polynomial. Here, we will follow another approach where we *sample* before every split to find a more suitable candidate for splitting. For this, when we get an unknown region, meaning a region which we have to split into smaller sub-regions, we calculate satisfiability for the exact middle point of the region. We then iterate through all axes or parameters and compare the borders of the interval corresponding to the axis with the middle point in terms of satisfiability. If they differ we know that there must be a more suitable candidate for splitting than the middle point. We then recursively approach new candidates on this segment until we find a well-suited one that we use for splitting on this axis.

We first have to say that we could not get an efficient and bug-free implementation of the sampling algorithm and disabled it in the most recent build. Nevertheless, we will present both of our approaches. In our first approach on *sampling* defined in Algorithm 3 we started on the left border of an axis and iterated by a factor of 0.1% of the whole interval size of this parameter. We stopped at the exact mid of the interval. If we could not find a suitable candidate we tried the same from the right border. A suitable candidate is one with a different satisfiability status than the exact middle point. This approach was of course really inefficient and resource-heavy. Most of the time we could not find a more suitable candidate than the middle point and therefore we wasted time going through the whole interval. In our second approach on *sampling* defined in Algorithm 4 we replaced the iteration and the factor by always taking the mid of the considered interval. We started by taking the mid between the left border and the exact middle point of the interval. If both points differ in their satisfiability status we found a more suitable candidate. If not we again took the mid of the left border and the last checked candidate. If we could not find a candidate after some time we repeated this process with the right border.

Correctness

Since *sampling* is an optimization problem there is no need to prove the correctness. The only important condition is that the new sampled candidate has to be inside the given region otherwise we would create new areas or dismiss old ones. \square

Algorithm 3 Sampling with Incrementation

Input: B $\triangleright B$ is the given box with an interval for each parameter where the status of the box B is unknown
Output: borders \triangleright The new borders either with new candidate or with just the middle point as without sampling

```

borders = []
 $\epsilon = (B[index][1] - B[index][0]) * 0.001$   $\triangleright$  steps to explore new candidates
for index, parameter in enumerate(NUMBER_OF_PARAMETERS) do
     $\triangleright$  Iterate through all parameters
    first_point =  $(B[index][0] + B[index][1]) / 2$   $\triangleright$  Middle point of the interval of the current parameter

    solver.push()
    solver.add(parameter == first_point)
    status1 = check_satisfiability()
    solver.pop()
    found = False
    counter = 0
    test_point =  $B[index][0]$ 
    while (counter < 499) and (not found) do  $\triangleright \epsilon$  is exactly 0.1% and we iterate to the exact mid
        test_point +=  $\epsilon$ 
        solver.push()
        solver.add(parameter == test_point)
        status2 = check_satisfiability()
        solver.pop()
        if status1 != status2 then  $\triangleright$  Stop if status of middle point and new candidate differs
            found = True
        else
            counter += 1
        end if
    end while
    if not found then
        counter = 0
        test_point =  $B[index][1]$ 
        while (counter < 499) and (not found) do  $\triangleright \epsilon$  is exactly 0.1% and we iterate to the exact mid
            test_point -=  $\epsilon$ 
            solver.push()
            solver.add(parameter == test_point)
            status2 = check_satisfiability()
            solver.pop()
            if status1 != status2 then  $\triangleright$  Stop if status of middle point and new candidate differs
                found = True
            else
                counter += 1
            end if
        end while
    end if
    if found then  $\triangleright$  Take new found candidate otherwise take the original middle
        borders.append([ $B[index][0]$ , test_point], [test_point,  $B[index][1]$ ])
    else
        borders.append([ $B[index][0]$ , first_point], [first_point,  $B[index][1]$ ])
    end if
end for

```

Algorithm 4 Sampling with Division of the Mid

Input: B $\triangleright B$ is the given box with an interval for each parameter where the status of the box B is unknown
Output: borders \triangleright The new borders either with new candidate or with just the middle point as without sampling

```

borders = [ ]
for index, parameter in enumerate(NUMBER_OF_PARAMETERS) do
     $\triangleright$  Iterate through all parameters
    first_point = ( $B[index][0]$  +  $B[index][1]$ ) / 2  $\triangleright$  Middle point of the interval of the current parameter

    solver.push()
    solver.add(parameter == first_point)
    status1 = check_satisfiability()
    solver.pop()
    found = False
    new_mid = ( $B[index][0]$  + first_point) / 2
    while not found do  $\triangleright$  Check the mid between left border and the middle point
        solver.push()
        solver.add(parameter == new_mid)
        status2 = check_satisfiability()
        solver.pop()
        if status1 != status2 then  $\triangleright$  Stop if status of middle point and new candidate differs
            found = True
        else
            new_mid = ( $B[index][0]$  + new_mid) / 2  $\triangleright$  New candidate is between checked point and the left border
        end if
    end while
    if not found then
        new_mid = ( $B[index][1]$  + first_point) / 2
        while not found do  $\triangleright$  Check the mid between right border and the middle
            solver.push()
            solver.add(parameter == new_mid)
            status2 = check_satisfiability()
            solver.pop()
            if status1 != status2 then  $\triangleright$  Stop if status of middle point and new candidate differs
                found = True
            else
                new_mid = ( $B[index][1]$  + new_mid) / 2  $\triangleright$  New candidate is between checked point and the right border
            end if
        end while
    end if
    if found then  $\triangleright$  Take new found candidate otherwise take the original middle
        borders.append([[ $B[index][0]$ , new_mid], [new_mid,  $B[index][1]$ ]])
    else
        borders.append([[ $B[index][0]$ , first_point], [first_point,  $B[index][1]$ ]])
    end if
end for

```

Chapter 4

PaSyPy

PaSyPy is a Python-based tool using parameter synthesis to find safe and unsafe regions of the parameter space. This chapter will explain the implementation of this tool and also how to use it. Extensive documentation, source code, tests and benchmarks can be found on the projects page of *PaSyPy* hosted on GitHub [1]. Further explanation can be found in Appendix A.

Figure 4.1 shows the full graphical user interface (GUI) of this tool. All different parts of it are explained in this chapter.

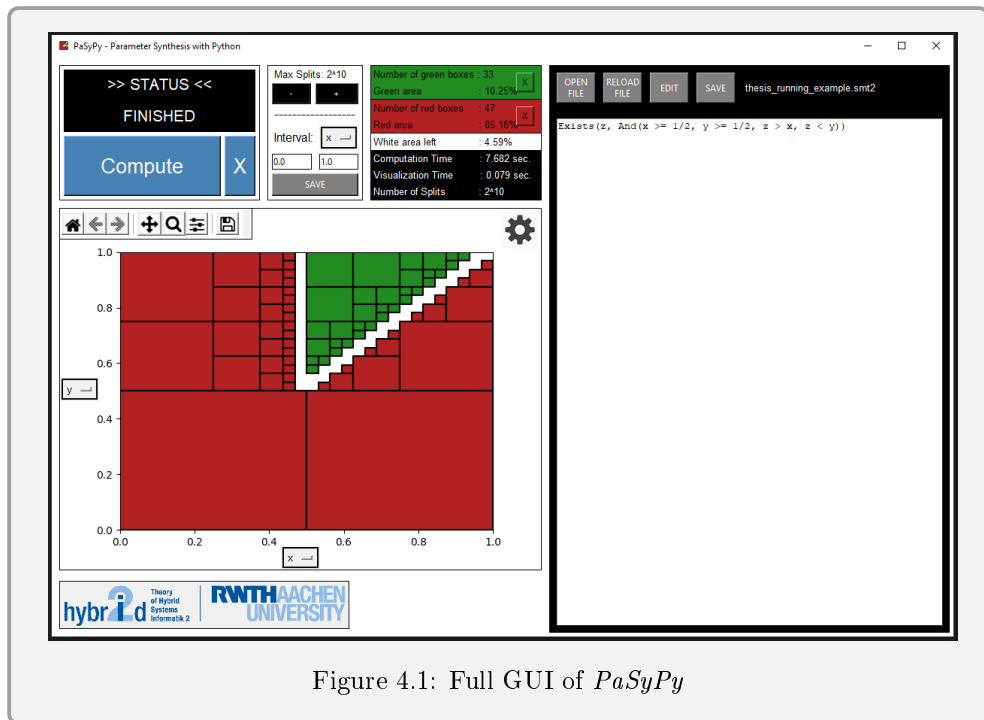


Figure 4.1: Full GUI of *PaSyPy*

4.1 Functionality

PaSyPy provides different functionalities, some parts that are pretty obvious but also parts that might be hidden. This section will cover every feature that this tool has to offer. Very roughly said, this tool computes and plots safe (green) and unsafe (red) regions for an input formula on nonlinear real arithmetic (NRA) combined with a box containing intervals for each parameter. The steps before computing and receiving a result are:

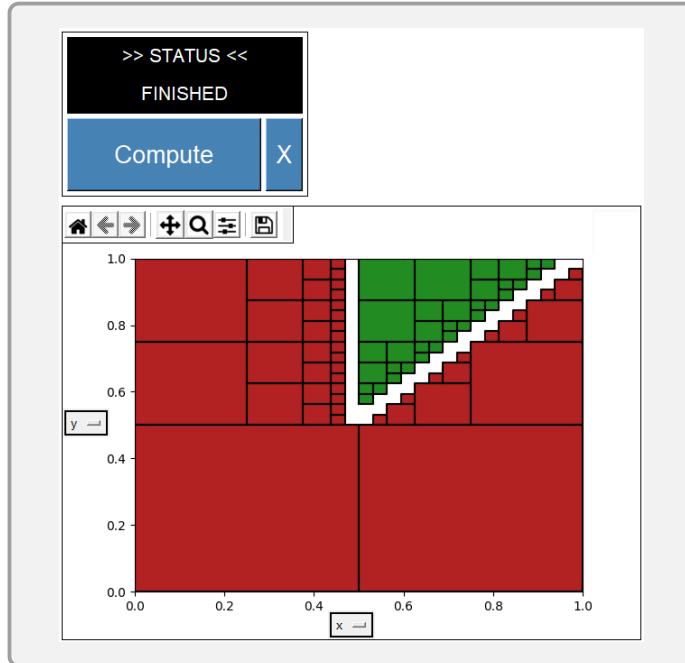
1. Input a logical formula.
2. Choose the splitting heuristic (see Section 3.2)
3. Adjust the maximum number of splits for the computation.
4. Select the interval for all parameters.
5. Select a parameter on each axis for plotting.

When all these tasks are completed, this tool will compute the result in the form of visualizing the safe (green) and unsafe (red) areas for the selected parameters on the given interval.

4.1.1 Computing and Visualizing

This subsection describes the main purpose of this tool. A given formula in nonlinear real arithmetic (NRA) combined with a box with pre-defined intervals for each parameter is computed. This means that at first, the full area for a given formula and its negation is checked for satisfiability. Then areas are either flagged as safe (green), unsafe (red) or unknown (white). The unknown (white) areas are split into smaller

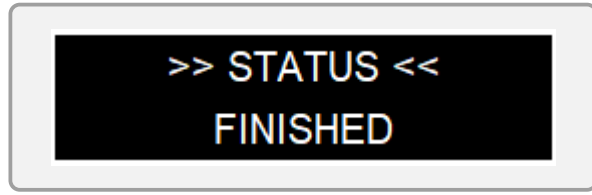
regions where the procedure is repeated. As a result, we will get a list of all found safe (green) and unsafe (red) regions which are then visualized for two chosen parameters. Note that all parameters are considered and the area is calculated for all of them, but due to the nature of 2D plots, only two parameters can be visualized at once. It is also possible to showcase single parameters in the form of a bar.



Status

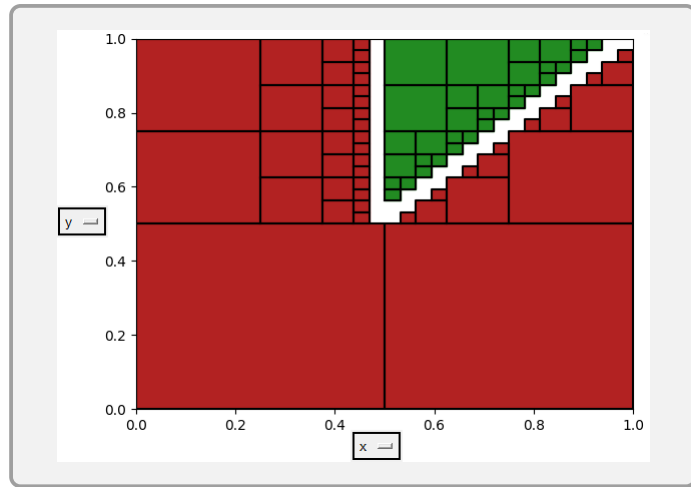
The status field will indicate the current state of the program. Initially, the state is *WAITING*.

Then based on if correct pre-conditions are met, the state is switched to *ERROR* or *READY*. When the computation is started the state will be *COMPUTING...* for the actual computation and *VISUALIZING...* for creating the graph. Finally, the state is switched to *FINISHED* to indicate the end of the computation and visualization.



Visualization field

One key part of this tool, the field where the found safe (green) and unsafe (red) regions are displayed. On the edge, the visualized parameters and the interval can be found. The main box represents the full viewed area. Inside it will contain safe (green) and unsafe (red) boxes, white left area and also a potential approximation curve which is called regression here. The boxes are those that are proven by the underlying solver. Red boxes are those that are proven to be unsatisfiable, more specifically this area only contains **unsat** solutions. Green boxes are those that are proven to be satisfiable, more specifically this area only contains **sat** solutions. The remaining white areas are those that contain both safe (green) and unsafe (red) regions. Regression, which is represented by a blue curve is an approximation of the exact border between safe (green) and unsafe (red) regions.




4.1.2 Information

This part contains useful information about the already computed and visualized process. Note that all information refers to the whole result and not to the currently shown dimensions which means all dimensions are considered.

Number of green boxes	: 33	X
Green area	: 10.25%	X
Number of red boxes	: 47	X
Red area	: 85.16%	X
White area left	: 4.59%	
Computation Time	: 7.682 sec.	
Visualization Time	: 0.079 sec.	
Number of Splits	: 2*10	

Number of Splits

This field shows the currently computed number of splits. The number of splits indicates the depth. A number of splits of one or 2^0 mean that there is one box, a number of splits of four or 2^2 will indicate that there are four boxes and a number of splits of 256 or 2^8 will indicate that there are 256 boxes. Regarding our *split heuristics* the *Default* tactic splits every box into $2^{\text{dimensions}}$ new boxes, which means if we have two parameters a number of splits of four or 2^2 allow us to split the box into four new boxes and will stop computing any further. The *Simple* tactic on the other hand splits every box into two new boxes, where our original box is split once and the two new boxes are also split, having four boxes at the end.



Number of Splits : 2¹⁰

4.1.3 Formula Handling

There are two choices for inputting a formula. Either the formula is parsed from a *.smt2* file or written manually. Independently from the chosen input the formula inside the textfield may be edited freely with compliance to the syntax. The expected input is a formula from nonlinear real arithmetic (NRA) logic.



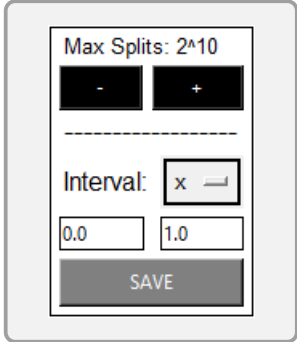
OPEN FILE RELOAD FILE EDIT SAVE thesis_running_example.smt2
Exists(z, And(x >= 1/2, y >= 1/2, z > x, z < y))

Exists(z, And(x >= 1/2, y >= 1/2, z > x, z < y))

Figure 4.2: Example syntax for a logical formula

4.1.4 Settings

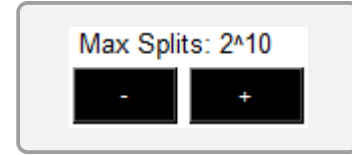
There are several settings the user can adjust. Most settings will initially have a default value set.



Max Splits: 2¹⁰
- +
Interval: x
0.0 1.0
SAVE

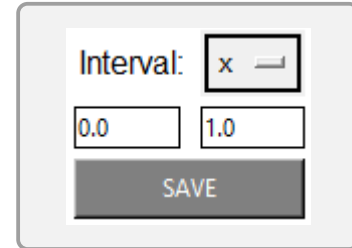
Increase/Decrease Maximum Number of Splits

The Number of splits were already explained. This option will set the grade of refinement with which the computation is performed. Increasing the maximum number of splits will reduce white area and approach the constraint boundaries with much smaller found regions. Initially, the maximum number of splits is 2^8 or 256 which for two parameters means it will split a box for a maximum of four times. One key feature of changing the max splits between computations is, that the results of the previous computation are always saved and increasing or decreasing the maximum number of splits will not lead to recalculation of already calculated regions. Therefore, increasing the maximum number of splits will only calculate the safe (green) and unsafe (red) regions of the additional maximum number of splits level. Decreasing the maximum number of splits will only update the graph and do not require new calculations.



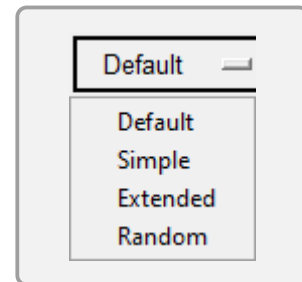
Adjust Parameter Boundaries

This option is really important considering the parameter synthesis problem. Here the boundaries for our parameters can be adjusted. Each parameter has its own interval which can be modified separately or we can modify all values at once. Most satisfiability modulo theories (SMT) solving tools can check for satisfiability of a given formula but cannot give whole areas proven sat or unsat. This tool however will gather safe (green) and unsafe (red) areas inside the chosen boundaries for our parameters. Initially, the interval is set to $[0.0, 1.0] \in \mathbb{R}$ for each parameter.



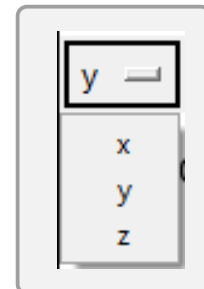
Choosing the Split Heuristic

This option allows the user to choose from all currently implemented split heuristics, i.e., *Default*, *Simple*, *Extended* and *Random*. (see Section 3.2) Initially, the split heuristic is set to *Default*.



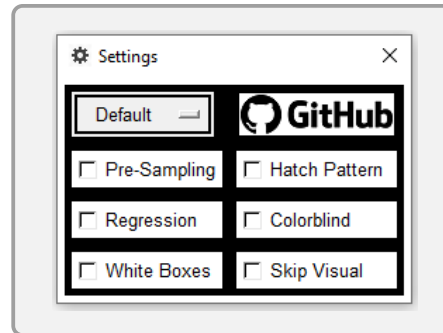
Select shown parameter

While all involved parameters are computed, only two parameters can be visualized at once. The user can select which parameters should be visualized on x -axis and y -axis respectively. Initially, the first parsed parameters are set for the x -axis and y -axis. In case of one parameter only the x -axis is set.



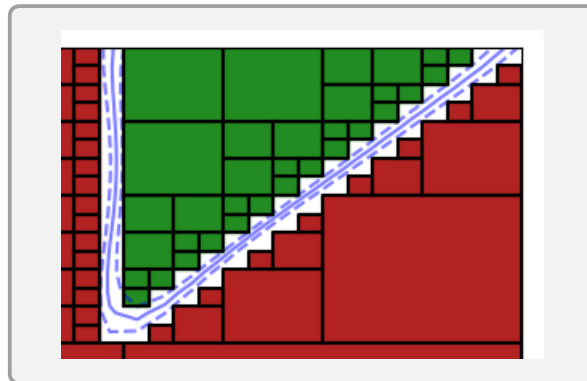
More Settings

Clicking on the settings wheel will open up a new window with further settings for customization.



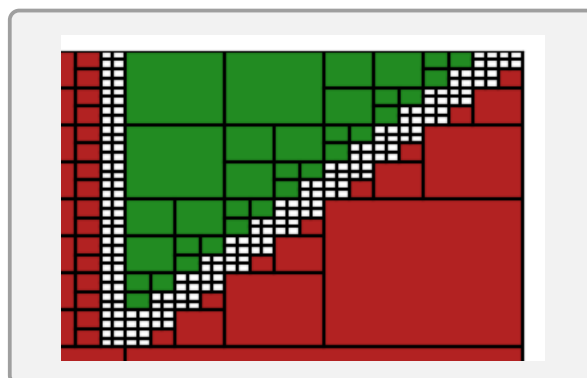
Regression

This tool computes an approximation of the function considering all safe (green) and unsafe (red) areas which is done by a *Large Margin Classifier* and regression. This option toggles the visibility of said regression on and off. Initially, regression is deactivated.



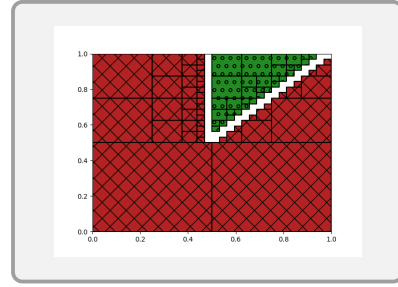
White Boxes

Usually only safe (green) and unsafe (red) boxes are drawn, while the outline of white boxes is left out. This option toggles the visibility of the outline of white boxes on and off. Initially, the outline for white boxes is deactivated.



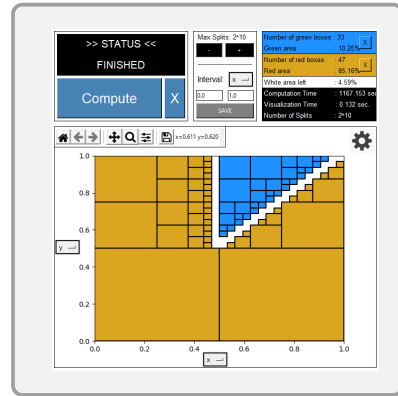
Hatch Pattern

This option toggles a hatch pattern for better differentiation between safe (green) and unsafe (red) areas. Initially, the hatch pattern is deactivated.



Colorblind Mode

The colorblind option will support those people who have difficulties differentiate green and red colors. The colorblind mode will display the safe area (green) with blue color and the unsafe area (red) with yellow color.



4.2 Supported Solvers

To provide satisfiability modulo theories (SMT) solving capabilities we need to embed a solver. Most parts or modules inside *PaSyPy* are solver independent, i.e., *gui.py*, *visualize.py*, *timer.py*, *logger.py*, *color.py* and *area_calculation.py*. The remaining parts or modules are in need of a solver. At the moment the only supported solver is the *z3 theorem prover*, but integrating additional solvers is planned in the future.

4.2.1 Z3 Theorem Prover

The *z3 theorem prover* [5] is a satisfiability modulo theories (SMT) solver by Microsoft Research. Its main purpose is to solve problems in software verification and software analysis. The default input format for the *z3 theorem prover* is *SMT-LIB*, more specifically *SMT-LIB2* which is described in Section 2.4.

While the Microsoft *z3 theorem prover* is originally written in *C++*, a Python API is provided with *Z3Py*. Our tool uses the *z3* Python API for defining solvers, reading constraints from *.smt2* files, and checking the formula for satisfiability and unsatisfiability. Most solvers provide the capability to define solvers, adding constraints and checking for satisfiability and unsatisfiability. But the function we use to read constraints from *.smt2* files is *z3* specific. This has to be considered when replacing the *z3 theorem prover* with any other solver. One solution is to let *z3* parse the formula and then convert the formula to the needed data type by the new solver.

Further information on how to use the *z3* Python API can be found in [16].

4.3 Validation

To validate that all modules and underlying functions are working properly the *unit testing framework* of Python was utilized to write unit tests [17]. Unit tests allow mocking a program sequence and give the possibility to verify exact variable values with assertions. The coverage rate of the written unit tests was then measured with the *Coverage.py* package, which tells exactly what lines of code were invoked by providing a detailed HTML report.

To check for some basic programming errors and help to enforce a coding standard the source code got checked with the static code analysis tool *pylint* [18].

4.4 Installation and Usage

To install this tool, clone the directory from GitHub [1] and install all dependencies:

- z3-solver
- matplotlib
- scikit-learn
- numpy

All dependencies are also included in **requirements.txt** and can be installed by **pip install -r requirements.txt**.

To start this tool, simply execute **__main__.py**.

4.5 Known Challenges

In general, the Python interface of the *z3 theorem prover* sometimes has difficulties automatically select the correct theory solver needed for the specific problem. This often results in a timeout even on relatively easy formulas. As a solution multiple theory solvers are used in parallel in case the default one fails to process the given formula. This does not have any effect on the performance or time needed to solve the problem.

The performance of Python in comparison to hardware-related programming languages (f.e. C and C++) is very weak. This problem can potentially be counteracted by integrating C into Python which is talked about in Section 7.2.

Performance is also highly dependent on the splitting heuristic and sampling regarding the computation step and array processing regarding the visualization step. While *Pre-Sampling* is really efficient and often leads to a huge performance boost, *Sampling* lacks efficiency and needs a rework. This can be further worked on to achieve optimal results.

Chapter 5

Other Tools

This chapter will introduce other tools that might have a relation to *PaSyPy*.

5.1 PROPhESY

PROPhESY [4] is probably the most similar tool to our tool *PaSyPy*, due to its nature of tackling the parameter synthesis problem together with satisfiability modulo theories (SMT) solving. *PROPhESY* is also written in Python and provides methods for parameter synthesis. As an input PROPhESY expects a rational function which is a fraction of two polynomials representing model parameters. It serves more as a library with interfaces to model checkers, i.e., *PRISM* [19] and *Storm* [20] and SMT solvers, i.e., *SMT-RAT* [21] and *z3* [5]. Therefore the full power of *PROPhESY* can only be utilized by using other packages.

5.2 pySMT

pySMT [22] is a Python API for satisfiability modulo theories (SMT) solving. It is not a direct tool for satisfiability modulo theories (SMT) solving, moreover, it acts as an interface for using the simple Python syntax on different originally non-Python solvers.

5.3 rise4fun

rise4fun [23] provides a web front end for software engineering tools and is hosted by Microsoft. It is a collection of various tools with the advantage of not having to worry about the configuration of the used machine. *rise4fun* also supports the *z3 theorem prover*. There an *.smt2* file can be parsed and checked for satisfiability. This can tell if the original formula combined with a box of intervals for each parameter is **sat**, but cannot split the box into smaller boxes and return safe and unsafe regions.

5.4 Similarities and Differences

As already teased at the beginning of Chapter 1, this tool combines satisfiability modulo theories (SMT) solving with the parameter synthesis problem and visualizes it. While there are several tools available for performing satisfiability modulo theories (SMT) solving, only a few can tackle the parameter synthesis problem and even fewer tools are able to visualize the results, especially for nonlinear real arithmetic (NRA). *PROPhESY* is more complex than *PaSyPy* and also provides more functionality. While *PROPhESY* is used to tackle the parameter synthesis problem on Markov models and expects a rational function as an input, *PaSyPy* expects other logical formulas and operates mostly on nonlinear real arithmetic (NRA). Both *PaSyPy* and *PROPhESY* use the *z3 Theorem Prover* and its Python interface, but *PaSyPy* is currently limited to it due to the missing implementation for other solvers.

Chapter 6

Case Study

In this chapter, we will analyze this tool regarding solutions and performance. Also we will compare all available split heuristics, particularly *Default*, *Simple*, *Extended* and *Random*. All tests were done on a computer with an **AMD Ryzen 5 2600X Six-Core processor and 16 Gigabytes of RAM**. We will first use our own examples to investigate different properties, i.e., number of parameters, degree of polynomial, quantification and sampling. In the end, we will use some heavier examples found on the benchmarks from the SMT-LIB [6].

Number of Parameters

At first, we will compare all split heuristics on very basic formulas for one, two and three parameters. The table displays time in seconds and total area found, meaning either safe (green) or unsafe (red) area, for different accuracies. Also, the table contains the underlying formula and considered interval.

Number of Splits	2^2		2^8		2^{16}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
Default	0.094	75.00	0.272	99.61	0.511	100.00
Simple	0.094	75.00	0.272	99.61	0.511	100.00
Extended	0.075	100.00	-	-	-	-
Random	0.096	69.09	0.309	99.52	0.581	100.00
Formula	$x \geq 1/2$					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.1: Simple formula with one parameter

Number of Splits	2^4		2^{10}		2^{14}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
Default	0.381	56.25	3.259	95.21	13.012	98.82
Simple	0.494	56.25	4.905	95.21	14.205	98.82
Extended	0.371	87.50	2.441	98.44	11.331	99.61
Random	0.242	58.83	3.071	91.58	13.379	98.09
Formula	$And(x \geq 1/2, y \geq 1/2, x \geq y)$					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.2: Formula with two parameters

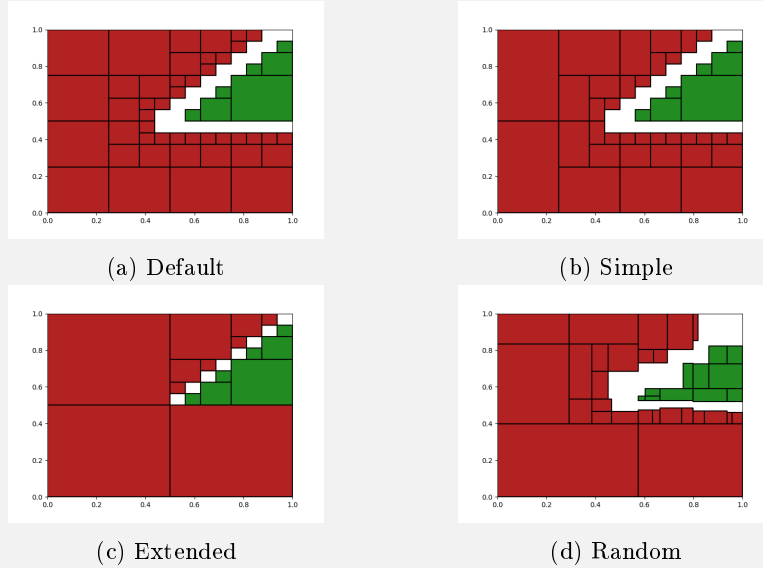
Number of Splits	2^9		2^{12}		2^{15}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
Default	3.951	88.48	10.841	95.19	35.945	97.82
Simple	5.327	88.48	16.764	95.19	56.601	97.82
Extended	2.439	96.48	8.241	98.05	32.195	98.95
Random	1.146	87.32	9.909	94.63	34.421	97.75
Formula	$And(x \geq 1/2, y \geq 1/2, z \geq 1/2, x \geq y, y \geq z)$					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.3: Simple formula with three parameters

Looking at the results we can already have an interim conclusion. The *Simple* split heuristic does yield the exact same results as the *Default* split heuristic. The difference is that on the *Default* tactic every split happens on every dimension at the same time, while on the *Simple* tactic the dimensions get cut split by split. This makes the *Simple* tactic loosing time compared to the *Default* tactic.

The *Random* split heuristic has the potential to yield very good and fast results. The problem is the unreliability of the randomness factor. Every execution of the *Random* tactic gives other results. On one iteration the safe (green) and unsafe (red) areas are nearly found instantly, while on the next iteration this tactic cannot find any valuable area. Therefore we will not consider the *Simple* and *Random* split heuristics on the next tests, but we will focus on the more interesting *Default* and *Extended* split heuristics.

Figure 6.1: Formula with two parameters.



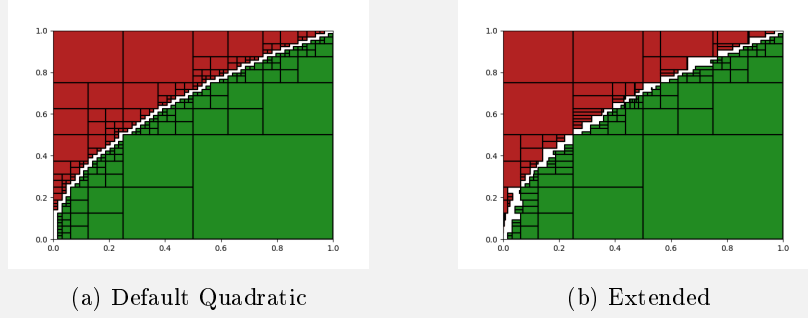
Degree of Polynom

Next we will use nonlinear formulas, i.e., with polynomial degree greater than one. For the sake of simplicity all nonlinear formulas will consist of two parameters.

Number of Splits	2^8		2^{12}		2^{16}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
Default	2.251	87.89	9.380	96.90	38.845	99.22
Extended	3.273	88.67	12.649	96.47	57.009	98.90
Formula	$x \geq y^2$					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.4: Formula with polynomial degree of two

Figure 6.2: Quadratic Formula

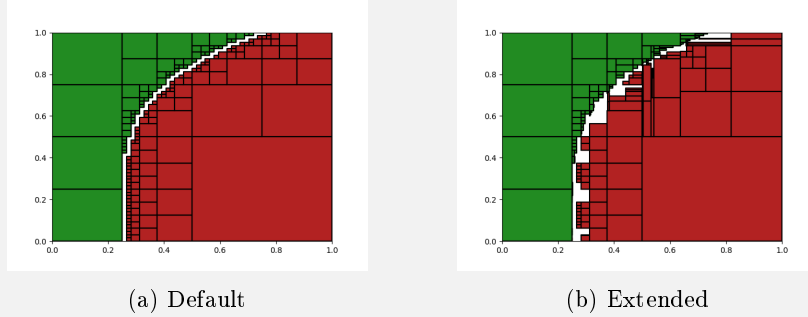


We can already see, that the *Extended* split heuristic is struggling on nonlinear formulas from the time in Table 6.4 and also from the found are in Figure 6.2b. It is slower than the *Default* tactic and also more unreliable. This is due to the underlying solver having difficulties with finding reliable solution points for splitting. If the *Extended* tactic cannot find a splitting candidate, it switches to the *Default* tactic. This means that even though the *Default* tactic is applied, the calculation from the *Extended* tactic is still performed prior which results in a loss of time.

Number of Splits	2^8		2^{12}		2^{16}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
Default	1.706	90.62	7.077	97.66	29.377	99.41
Extended	3.118	88.66	16.301	89.47	69.188	97.44
Formula	$2x \leq y^4 + 0.5$					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.5: Formula with polynomial degree of four

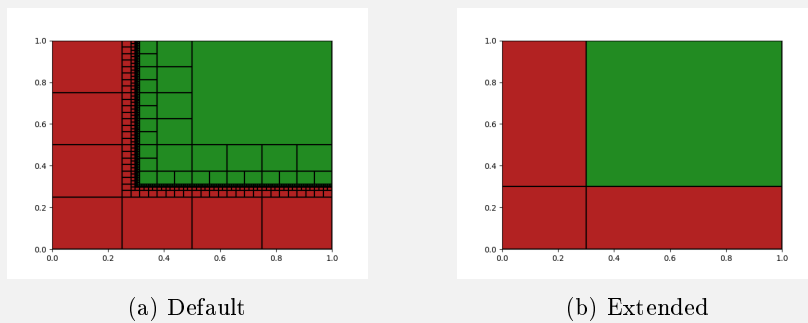
Figure 6.3: Quartic Formula



As a conclusion to the different tactics, we can say that the most reliable split heuristic is the *Default* tactic as it yields good performance and also provides reliable solutions even on multi polynomial formulas. The *Extended* tactic struggles on non-linear formulas but outperforms the *Default* tactic on linear formulas as it finds more safe (green) or unsafe (red) areas in less amount of time.

The difference between the *Extended* tactic and the *Default* tactic in finding solutions do not only depend on the number of parameters or the degree of the polynomials, but also on the formula itself. While the *Default* tactic always exactly splits in the middle meaning that it cannot take any shortcut even if the solution for the formula is really easy to see. The *Extended* tactic on the other hand has the potential to find the complete area nearly instantly as shown in Figure 6.4.

Figure 6.4: Extended tactic beats Default tactic



Number of Splits	2^2		2^{12}		2^{22}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
Default	0.109	25.00	5.894	97.83	219.228	99.93
Extended	0.159	100.00	-	-	-	-
Formula	$And(x \geq 0.3, y \geq 0.3)$					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.6: Extended tactic beats Default tactic

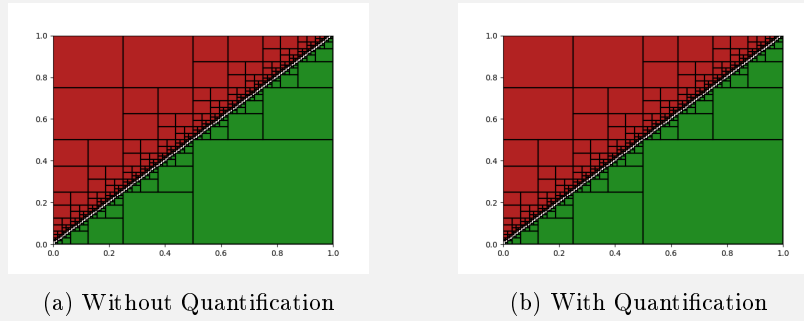
Existential Quantification vs. No Quantification

We will compare a technically equal formula in real arithmetic and in quantifier-free real arithmetic.

Number of Splits	2^8		2^{14}		2^{20}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
without quantifier	1.842	87.89	16.484	98.44	138.366	99.80
with quantifier	1.952	87.89	17.568	98.44	147.415	99.93
Formula	<i>Without Quantifier: $x \geq y$</i>					
Formula	<i>With Quantifier: $Exists(z, And(x \geq z, z \geq y))$</i>					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.7: Existential Quantification vs. No Quantification

Figure 6.5: Existential Quantification vs. No Quantification



As shown in Figure 6.5, both formulas output the exact same graph regardless of quantification used or not. There are only some minor differences in performance as shown in Table 6.7, where the formula without quantification is computed slightly faster.

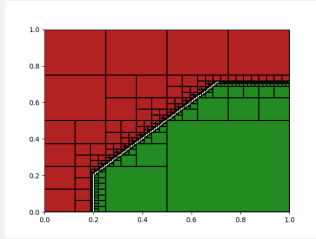
Pre-Sampling

On some formulas that are considered easy, the solver has difficulties separating safe (green) and unsafe (red) regions. For this, we use *pre-sampling* which can increase performance by filtering the initial region to get more favorable regions.

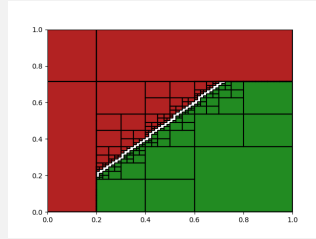
Number of Splits	2^8		2^{14}		2^{20}	
Split Heuristic	Time/s	Area%	Time/s	Area%	Time/s	Area%
without Sampling	1.501	90.62	13.131	98.82	107.994	99.85
with Sampling	1.136	90.18	9.428	98.77	67.729	99.85
Formula	$And(x \geq 1/5, y \leq 5/7, x \geq y)$					
Interval	$[0.0, 1.0] \in \mathbb{R}$					

Table 6.8: Pre-Sampling

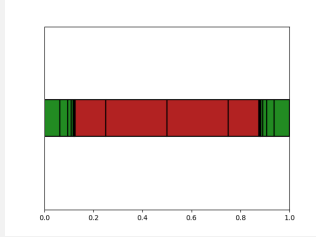
Figure 6.6: No Sampling vs. Sampling



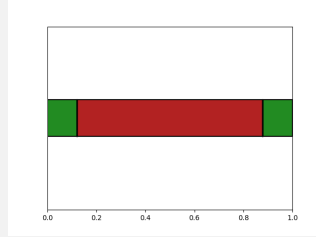
(a) No Sampling - Multiple Parameters



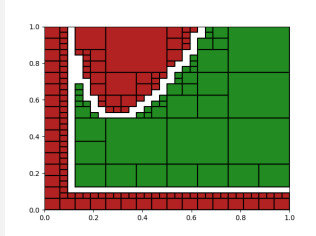
(b) Sampling - Multiple Parameters



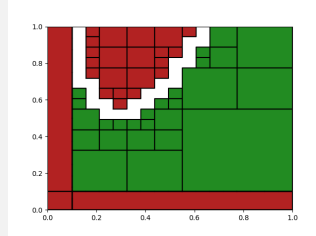
(c) No Sampling - Single Parameter



(d) Sampling - Single Parameter



(e) No Sampling - Complex Polynomial



(f) Sampling - Complex Polynomial

As shown in Figure 6.6, pre-sampling can give an initial advantage on the borders. This advantage scales with the number of parameters. On ten different parameters, pre-sampling founds over 99.99% in a relatively short amount of time. Without pre-sampling finding any safe (green) or unsafe (red) region on this high number of parameters is not possible in a feasible time.

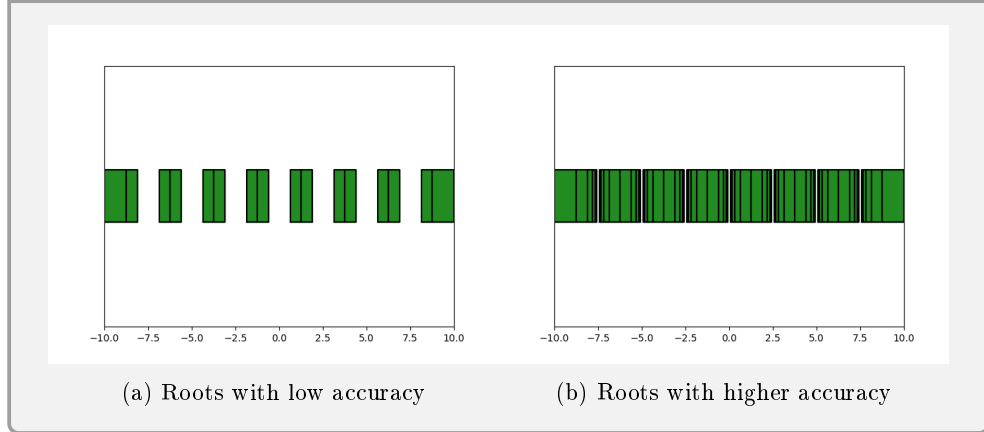
Split Heuristic	Time/s	Area%
without Pre-Sampling	1000.000	0.10
with Pre-Sampling	20.000	99.99
Formula	And($x1 \geq 1/5, x2 \geq 1/5, \dots, x10 \geq 1/5$)	
Interval	$[0.0, 1.0] \in \mathbb{R}$	

Table 6.9: Pre-Sampling with multiple parameters

Correctness

It is not easy to verify the correctness of our visualization. For this, we will abuse the nature of polynomials of the form $(x - 1) \cdot (x + 2) \cdot \dots$ as we exactly know the roots there with one and minus two on this example. Also, we can see how this tool computes formulas with only one parameter and also on different intervals other than $[0.0, 1.0] \in \mathbb{R}$.

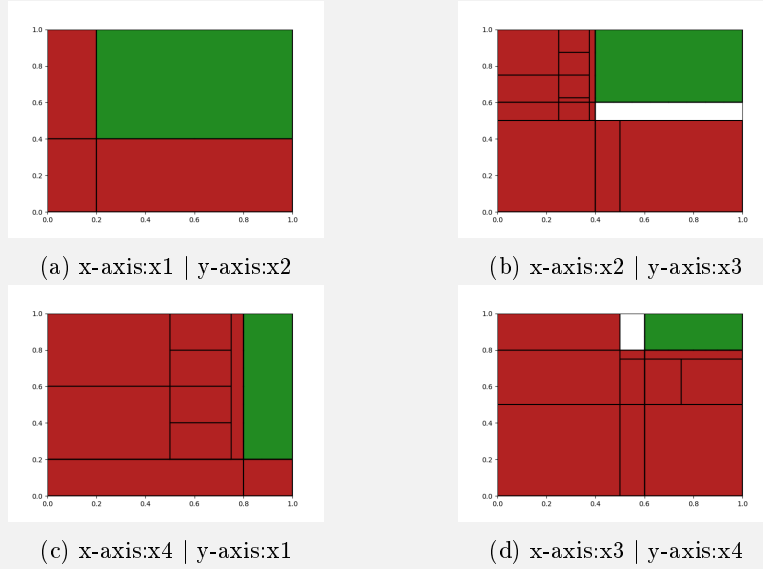
Figure 6.7: Correctness of Roots for $(x - 7.5) \cdot (x - 5) \cdot (x - 2.5) \cdot x \cdot (x + 2.5) \cdot (x + 5) \cdot (x + 7.5) \neq 0$ for interval $[-10.0, 10.0] \in \mathbb{R}$



As we can see in Figure 6.7 this tool yields us the correct roots for our formula, i.e., the expected roots $\{-7.5, -5.0, -2.5, 0, 2.5, 5.0, 7.5\}$ for our polynomial $(x - 7.5) \cdot (x - 5) \cdot (x - 2.5) \cdot x \cdot (x + 2.5) \cdot (x + 5) \cdot (x + 7.5)$. One big topic we did not cover yet is the solution as it is only an **approximation** and it gets more accurate with increasing accuracy. Therefore, we will usually do not get to 100% area coverage but only close to it, except if the formula lies favorably as in Figure 6.4b.

We will also take a simple formula with four parameters to show that all axes are visualized correctly. Figure 6.8 shows that all axes are visualized as expected. The scale from all graphs is in the range from 0 to 1.

Figure 6.8: Formula: $\text{And}(x1 \geq 0.2, x2 \geq 0.4, x3 \geq 0.6, x4 \geq 0.8)$



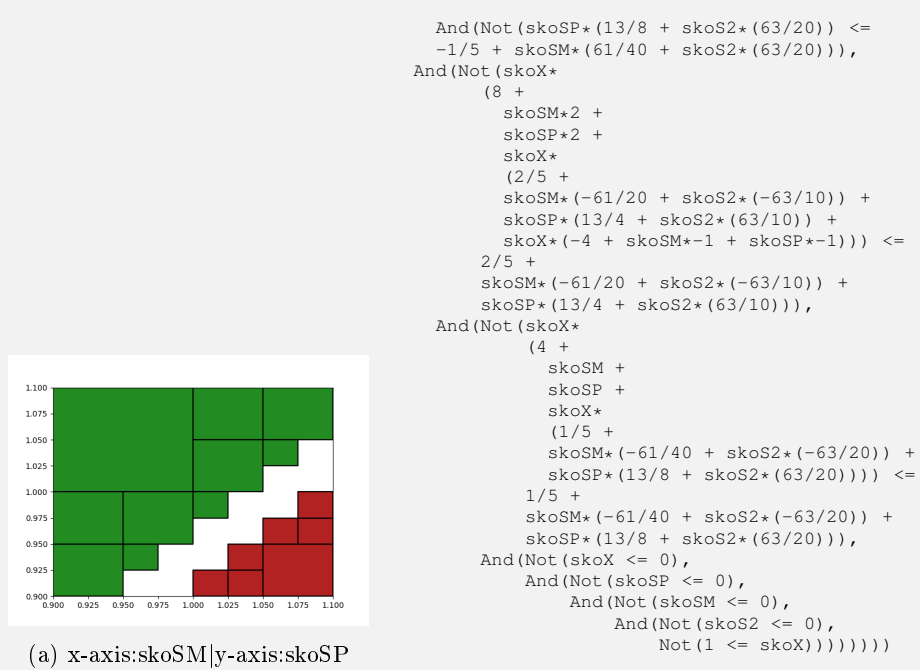
Time Analysis

Analyzing all parts inside the computation leads to the following breakdown of time:

- Under 1% for initializing the solvers, resetting previous constraints, checking conditions like current zoom-level and creating the logfile.
- 5% for assigning boundaries of the current region.
- The remaining time (90%+) is used by the underlying solver for checking the region inside the assigned boundaries.

SMT-LIB Benchmarks

Now we will use examples we found on the benchmarks from SMT-LIB. For this, we first searched for benchmarks that have a model. We then got the model by inputting the benchmark in the *Z3 Online Demonstrator* [24] and added a (**get-model**) line at the end. If the *Z3 Online Demonstrator* found a model we noted down the satisfiable value for every parameter and set our region around those values. We also used some unsatisfiable formulas where we slowly increased the region to check if we also get an unsatisfiable result.

Figure 6.9: *QF_NRA/meti-tarski/asin/8/vars4/asin-8-vars4-chunk-0056.smt2*

Green Area%	Red Area%	White Area Left%	Time/s
67.19	15.62	17.19	57.911
<i>Interval</i>	<i>skoSP</i> := $[0.9, 1.1]$, <i>skoS2</i> := $[0.9, 1.1]$, <i>skoSM</i> := $[0.9, 1.1]$, <i>skoSX</i> := $[0.4, 0.6]$		

Table 6.10: SMT-LIB - in-8-vars4-chunk-0056.smt2

As we can see from Figure 6.9 and Table 6.10 we parsed the formula correctly and could compute it on our region.

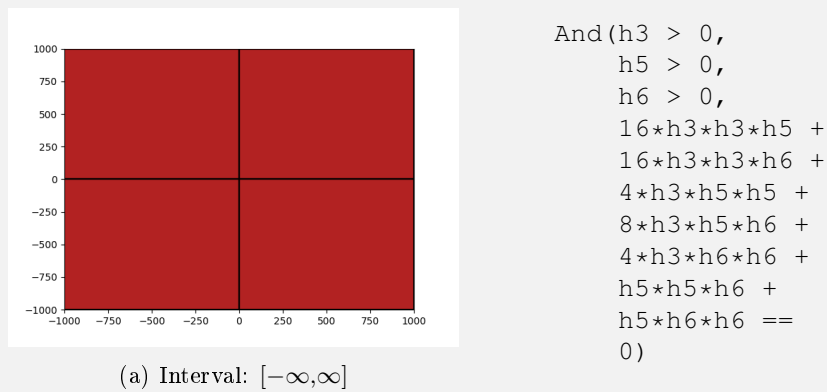
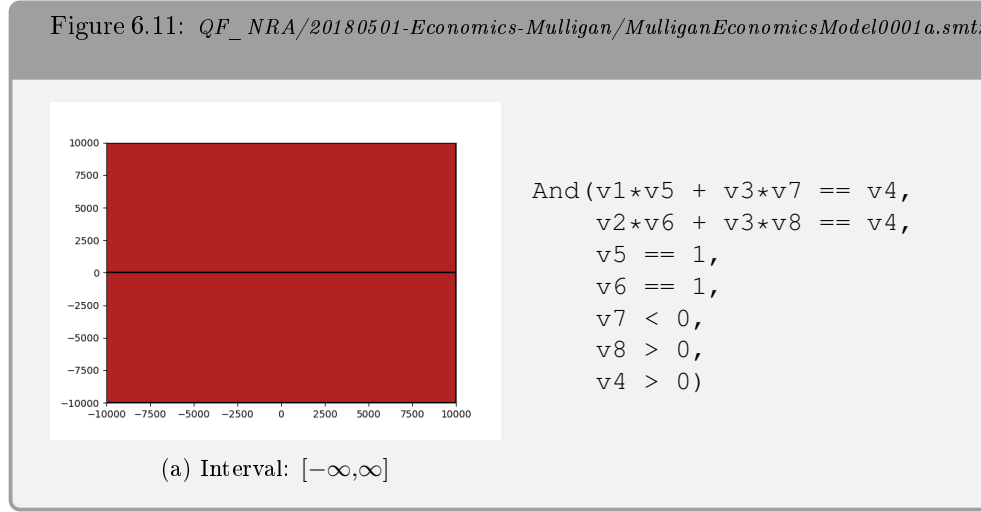
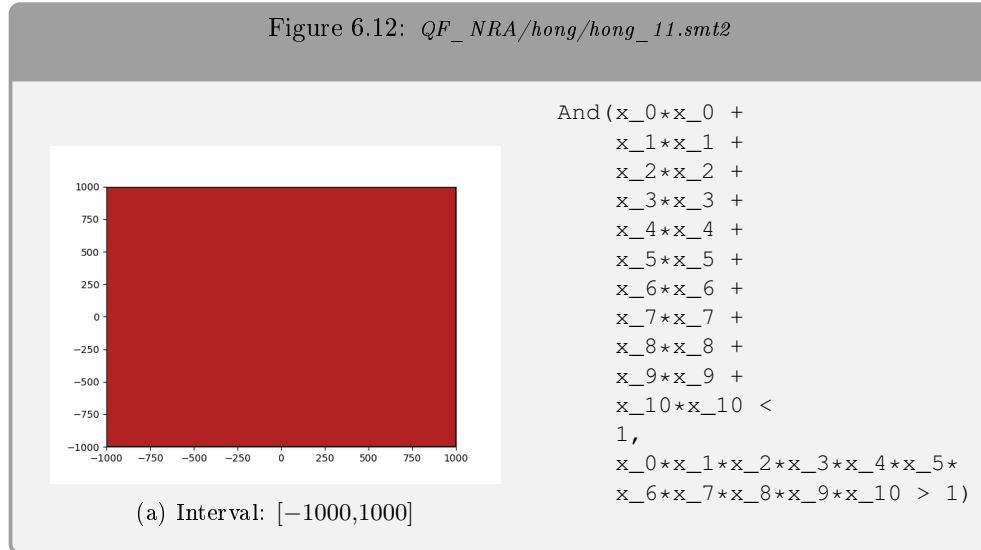
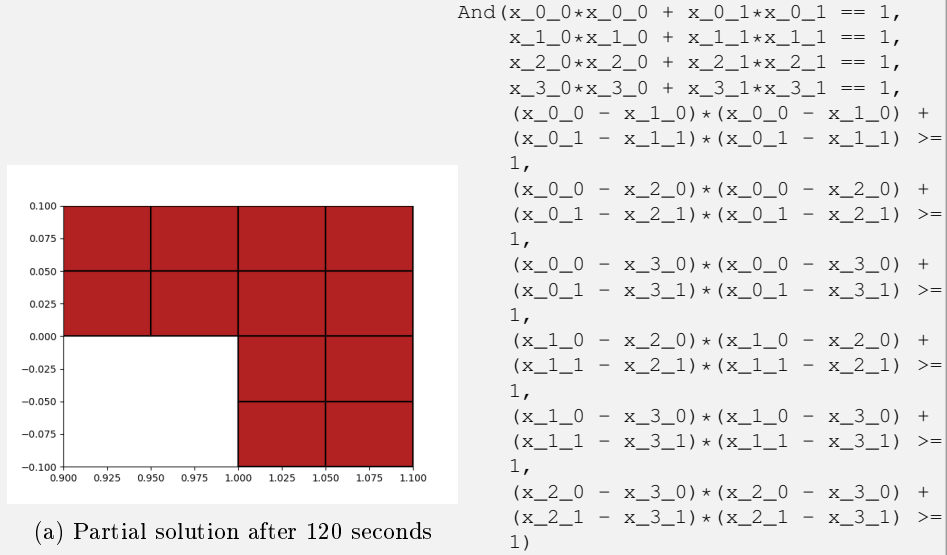
Figure 6.10: *QF_NRA/20161105-Sturm-MBO/mbo_E28.smt2*

Figure 6.11: *QF_NRA/20180501-Economics-Mulligan/MulliganEconomicsModel0001a.smt2*

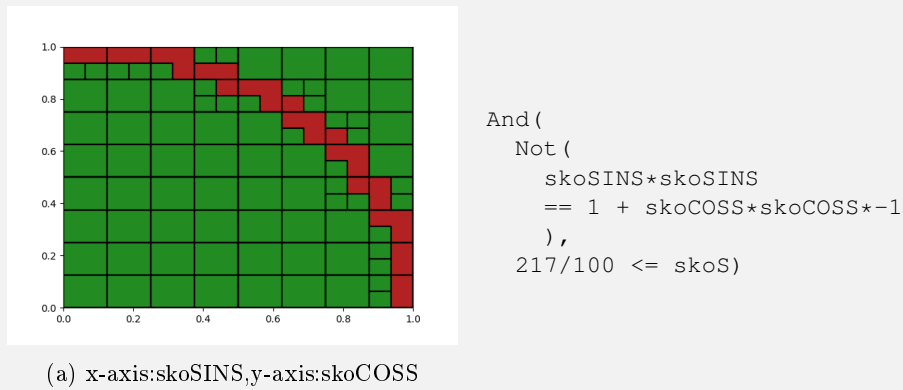
In Figure 6.10 and Figure 6.11 we started on our default region with interval borders $[0.0, 1.0]$ for every parameter and slowly increased the interval borders until we reached $[-\infty, \infty]$ to prove that this formula is unsatisfiable. After only a few seconds we already got our results and covered the whole parameter space.

Figure 6.12: *QF_NRA/hong/hong_11.smt2*

The formula from Figure 6.12 took a bit more time for finding unsatisfying values for our parameters. On an interval of $[-1000, 1000]$ for every parameter, we got the results after around 60 seconds.

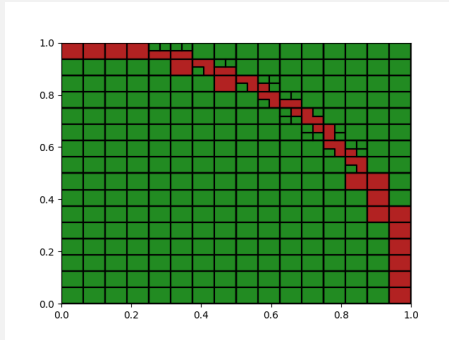
Figure 6.13: *QF_NRA/kissing/kissing_2_4.smt2*

On the formula from Figure 6.13 which has a satisfying solution, we could not get it in time. But we could get unsatisfying regions that slowly got discovered the longer the program was running. Refining the region, i.e., setting each interval for every parameter, would eventually find a satisfying (safe) region. After around 300 seconds we found 0% safe area but over 20% unsafe area.

Figure 6.14: *QF_NRA/meti-tarski/Arthan/1C/Arthan1C-chunk-0007.smt2*

The formula from Figure 6.14 returned really interesting results on the interval borders $[0.0,1.0]$ for *skoSINS* and *skoCONS* and the interval border $[2.0,3.0]$ for *sko*. We could find around 73% of safe regions and around 17% of unsafe regions in around 87 seconds. The formula from Figure 6.15 also worked very well but this time on the region with $[0.0,1.0]$ as an interval for every parameter. This time it returned around 45% for both safe and unsafe regions in around 30 seconds.

Figure 6.15: *QF_NRA/meti-tarski/Arthan/KM2/ArthanKM2-chunk-0007.smt2*



```
And (
  Not (
    skoSINS*skoSINS
    == 1 + skoCOSS*skoCOSS*-1
  ),
  9/20 <= skoS)
```

(a) x-axis:skoCOSS,y-axis:skoSINS with partial result

Conclusion on SMT-LIB Benchmarks

We could further work through every available benchmark from SMT-LIB but we will stop here and make a conclusion. First, we have to annotate that most benchmarks available are not suitable for our tool. Some benchmarks even timeout on finding a single model. Proving whole regions as satisfying (safe) or unsatisfying (unsafe) is close to impossible there. Our tool always starts with a default interval for every parameter. If a formula contains multiple parameters often all parameters have different accepting and rejecting ranges. E.g., a formula with three parameters x , y and z expects x to be in $[-100,-50]$, y to be in $[275,975]$ and z to be in $[0.001,0.003]$. If the expected intervals are known prior this is no problem but otherwise, it is nearly impossible to find a solution in a feasible time. This problem gets worse the more parameters a formula has. The second problem is constraints having either the equal-to operator $==$ or the not-equal-to operator $!=$. Our tool cannot find an exact solution for such constraints but can only approximate to its borders. E.g., if we have the formula $x == 2$ we will never get a satisfiable value for x since our tool works on regions and not exact points. Nevertheless, we could get interesting results on the used benchmarks.

Chapter 7

Conclusion

The goal of this thesis, mainly developing and implementing a Python-based tool to find safe and unsafe regions of the parameter space using parameter synthesis, was successfully accomplished. *PaSyPy* is a useful, easy-to-use tool for finding those regions with consideration of all parameters tackling the parameter synthesis problem on nonlinear real arithmetic logic. It is especially useful in visualizing the interesting areas.

7.1 Summary

The problem of parameter synthesis in satisfiability modulo theories (SMT) solving is of high theoretical and also practical interest, due to its optimization potential in terms of optimal system composition. Unfortunately, this problem is not trivial. The tool developed during this thesis, namely *PaSyPy*, provides promising results regarding satisfiability modulo theories (SMT) solving with parameter synthesis on nonlinear real arithmetic.

7.2 Future Work

Even though *PaSyPy* is already a working and complete tool, there are still possibilities for extension and improvement. At this moment, the only solver supported by *PaSyPy* is the *z3 Theorem Prover* specifically its Python interface. Adding more solvers e.g. *SMT-RAT* would diversify this tool and allow comparing different solvers inside *PaSyPy* concerning performance.

To find safe and unsafe regions of the parameter space, different splitting heuristics are available for selection. Because of how well-structured the source code of this tool is, it is pretty easy to add further splitting heuristics. This gives a great opportunity to optimize specific problems with specific algorithms and benefit from the underlying visualization of *PaSyPy*. The same applies to *sampling* where we could not find an efficient and bug-free implementation.

In general, even though Python is a great programming language and has a lot of advantages, it does not yield the best performance. One approach to benefit from the simplicity of Python and not dispense with high performance, is to integrate Cython [25] an optimizing static compiler. This would give us the combined power of Python and the programming language C, which is well known for high performance due to hardware-related functionality to speed up the execution of our Python code.

Right now our tool only allows nonlinear real arithmetic and similar logic. This means our tool only accepts a system of equalities or inequalities connected by \wedge and \vee . The next step could be implementing support for boolean structures and trigonometric functions. For this, parsing has to be adapted.

Another improvement would apply to formulas with a lot of parameters. Because every parameter has a different feasible region which is a region where the parameter satisfies the formula, there is a problem as we always start on the default region $[0.0, 1.0] \in \mathbb{R}$ for every parameter. An approach would be to first give the formula to the solver to get a model containing a satisfying value for each parameter. This would of course only work if the formula has a satisfying solution. We then could build the region around the found model for every parameter.

References

- [1] Alexander Wiegel. PaSyPy. <https://github.com/awiegel/PaSyPy>, 2021.
- [2] Sebastian Junges, Erika Ábrahám, Christian Hensel, Nils Jansen, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. Parameter Synthesis for Markov Models. *CoRR*, abs/1903.07993, 2019.
- [3] Christian Dehnert, Sebastian Junges, Nils Jansen, Florian Corzilius, Matthias Volk, Joost-Pieter Katoen, Erika Ábrahám, and Harold Bruintjes. Parameter Synthesis for Probabilistic Systems. In Ralf Wimmer, editor, *19th GI/ITG/GMM Workshop Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, MBMV 2016, Freiburg im Breisgau, Germany, March 1-2, 2016*, pages 72–74. Albert-Ludwigs-Universität Freiburg, 2016.
- [4] Sebastian Junges and Matthias Volk. PROPhESY. <https://github.com/moves-rwth/prophesy>, 2014.
- [5] Leonardo de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963, pages 337–340. Springer Berlin Heidelberg, 04 2008.
- [6] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2016.
- [7] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. SMT-RAT: An SMT-Compliant Nonlinear Real Arithmetic Toolbox. In *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317, pages 442–448. Springer Berlin Heidelberg, 06 2012.
- [8] S. Cook. The complexity of theorem-proving procedures. *Proceedings of the third annual ACM symposium on Theory of computing*, 1971.
- [9] L. A. Levin. Universal problems of full search. *Probl. Peredachi Inf.*, 9(3):115–116, 1973.
- [10] Erika Ábrahám and Gereon Kremer. SMT Solving for Arithmetic Theories: Theory and Tool Support. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 1–8, 2017.
- [11] Fredrik Lundh. An introduction to tkinter. *URL: www.pythonware.com/library/tkinter/introduction/index.htm*, 1999.
- [12] John D Hunter. Matplotlib: A 2D graphics environment. *Computing in science & engineering*, 9(3):90–95, 2007.

- [13] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in Python. *Journal of machine learning research*, 12(Oct):2825–2830, 2011.
- [14] Charles R. Harris, K. Jarrod Millman, Stéfan J van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585:357–362, 2020.
- [15] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.
- [16] Nikolaj Bjørner, Leonardo de Moura, Lev Nachmanson, and Christoph M. Wintersteiger. *Programming Z3*, pages 148–201. Springer International Publishing, Cham, 2019.
- [17] Unit testing framework for Python. <https://docs.python.org/3/library/unittest.html>.
- [18] Pylint. <https://pylint.org/>.
- [19] Marta Kwiatkowska, Gethin Norman, and David Parker. PRISM 4.0: Verification of Probabilistic Real-Time Systems. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 585–591, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.
- [20] C Dehnert, Sebastian Junges, J.-P. Katoen, and M Volk. A storm is coming: A modern probabilistic model checker. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 592–600, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.
- [21] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. SMT-RAT : an Open Source C++ Toolbox for Strategic and Parallel SMT Solving. In *Theory and Applications of Satisfiability Testing : SAT 2015 ; 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings / Marijn Heule ; Sean Weaver [Hrsg.]*, volume 9340 of *Lecture Notes in Computer Science*, pages 360–368, Cham, Sep 2015. International Conference on Theory and Applications of Satisfiability Testing, Austin, Tex. (USA), 24 Sep 2015 - 27 Sep 2015, Springer International Publishing.
- [22] Marco Gario and Andrea Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT Workshop 2015*, 2015.
- [23] rise4fun. <https://rise4fun.com/>.
- [24] Z3 Online Demonstrator. <https://compsys-tools.ens-lyon.fr/z3/>.
- [25] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn, and Kurt Smith. Cython: The best of both worlds. *Computing in Science & Engineering*, 13(2):31–39, 2011.

Appendix A

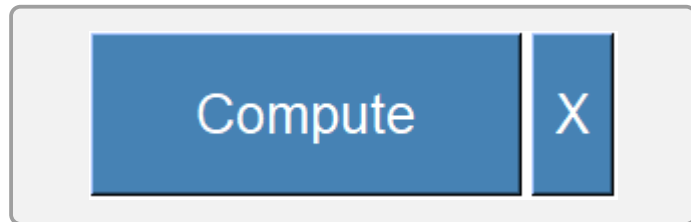
Further Tool Explanation

We give an extended overview of all parts of our tool that we have not covered yet.

A.1 Buttons

Compute button

This button will initiate the computing. This will only work if all pre-conditions are fulfilled, i.e., a correct formula. The 'X'-button immediately stops the computation and returns a partial solution.



Navigation toolbar

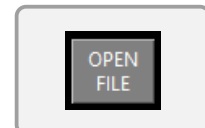
The navigation toolbar provides useful functions, i.e., zooming inside an area and also

saving the current plot in a file. If zooming is active the next computation will only consider the zoomed area which can be used to refine specific regions.



Open File

With this button, one can select an *SMT-FILE* (.smt2), which then gets parsed by the *z3* Python interface and the formula is shown in the text field.



Reload File

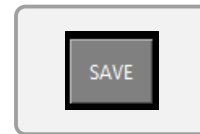
Reloads the last opened file, restoring the original formula.

**Edit**

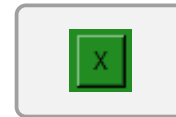
The user has the possibility to write own constraints or edit the parsed formula from a *SMT-FILE* (.smt2). To apply the made changes and give the new formula to the solver, one has to press this button.

**Save**

This button allows the user to save the formula from the text field into a *SMT-FILE* (.smt2).

**Show green area**

All safe (green) boxes containing the interval for every parameter are saved to a log-file (*safe_area.log*), which can be opened by this button.

**Show red area**

All unsafe (red) boxes containing the interval for every parameter are saved to a log-file (*unsafe_area.log*), which can be opened by this button.

**GitHub**

Clicking on the GitHub icon will redirect to the official GitHub page of *PaSyPy* [1].

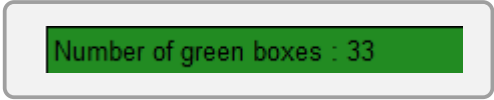
University Logo

Clicking on the RWTH Theory of Hybrid Systems icon will redirect to their official website <https://ths.rwth-aachen.de/>.

A.2 Information Fields

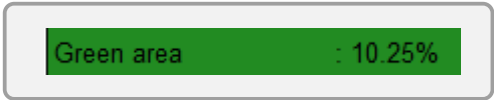
Number of green boxes

This field shows the sum of all safe (green) boxes across all dimensions.



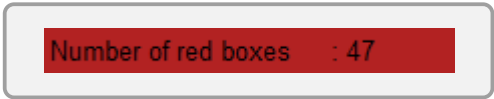
Green area

This field shows the percentage of safe (green) area in comparison to the overall area.



Number of red boxes

This field shows the sum of all unsafe (red) boxes across all dimensions.



Red area

This field shows the percentage of unsafe (red) area in comparison to the overall area.



White area left

This field shows the percentage of white area in comparison to the overall area, meaning all area that contains both safe (green) and unsafe (red) regions.



Computation Time

This field shows the required time for computing the formula and finding the safe (green) and unsafe (red) regions.



Visualization Time

This field shows the required time for calculating and visualizing the underlying graph.

Visualization Time : 0.086 sec.

Filename

This field reflects the name of the currently opened (*.smt2*) file.

thesis_running_example.smt2

Textfield

This field either represents the formula parsed from the last opened (*.smt2*) file or a self-defined formula. The formula inside this field may be edited freely and reparsed by hitting the *Edit* button.

Exists(z, And(x >= 1/2, y >= 1/2, z > x, z < y))

A.3 Settings

Pre-Sampling

This option toggles the pre-sampling mechanism when inputting a formula. Initially, pre-sampling is activated.

Sampling

This option toggles the sampling mechanism before every splitting. Initially, sampling is deactivated.

Skip Visualization

Activating this option will skip the visualization part. The safe (green) and unsafe (red) areas can still be acquired by the logs.

A.4 Miscellaneous

Keyboard Shortcuts

There are some keyboard shortcuts available to facilitate the user experience. Holding the control key (*CTRL*) and:

- *+*, for increasing the current accuracy.
- *-*, for decreasing the current accuracy.
- *o*, for opening a file.
- *r*, for reloading the currently opened file.
- *s*, for saving the current formula in a file.

Logs

This tool saves several logs:

- *logfile.log*, with the internal sequence of the normal solver.
- *logfile_neg.log*, with the internal sequence of the negated solver.
- *safe_area.log*, with all safe regions found.
- *unsafe_area.log*, with all unsafe regions found.