UNIVERSITÀ DEGLI STUDI
DI TRENTO

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE
ICT International Doctoral School

Incremental Linearization for Satisfiability and Verification Modulo Nonlinear Arithmetic and Transcendental Functions

Ahmed Irfan

Advisor

Prof. Alessandro Cimatti

Fondazione Bruno Kessler

Co-Advisors

Prof. Roberto Sebastiani **Dr. Alberto Griggio**

Università degli Studi di Trento Fondazione Bruno Kessler

April 2018

To my parents,

Shafaat Ali
&
Nasreen Akhter

Abstract

Satisfiability Modulo Theories (SMT) is the problem of deciding the satisfiability of a first-order formula with respect to some theory or combination of theories; Verification Modulo Theories (VMT) is the problem of analyzing the reachability for transition systems represented in terms of SMT formulae. In this thesis, we tackle the problems of SMT and VMT over the theories of polynomials over the reals ($\mathcal{NRA}$), over the integers ($\mathcal{NIA}$), and of $\mathcal{NRA}$ augmented with transcendental functions ($\mathcal{NTA}$). We propose a new abstraction-refinement approach called Incremental Linearization*. The idea is to abstract nonlinear multiplication and transcendental functions as uninterpreted functions in an abstract domain limited to linear arithmetic with uninterpreted functions. The uninterpreted functions are incrementally axiomatized by means of upper- and lower-bounding piecewise-linear constraints. In the case of transcendental functions, particular care is required to ensure the soundness of the abstraction. The method has been implemented in the* MathSAT *SMT solver, and in the* nuXmv *VMT model checker. An extensive experimental evaluation on a wide set of benchmarks from verification and mathematics demonstrates the generality and the effectiveness of our approach.*

Moreover, the proposed technique is an enabler for the (nonlinear) VMT problems arising in practical scenarios with design environments such as Simulink*. This capability has been achieved by integrating* nuXmv *with* Simulink *using a compilation-based approach and is evaluated on an industrial-level case study.*

Keywords

[SMT, VMT, Satisfiability, Formal Verification, Model Checking, Nonlinear Arithmetic, Transcendental Functions, Automated Reasoning]

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Hardware and software systems have become not only a fundamental component in safety- and mission-critical applications but also an essential component of human lives. Their increasing use comes with many challenges: demand for additional functionalities, lower development and maintenance cost, shorter time to market, and correct operation are few examples. In fact, ensuring correctness of systems is extremely important. These challenges have led to the use of model-based design methodologies with the support of formal verification tools to ensure correctness (or to find bugs). Nowadays, several design tools, as well as verification tools, are available. A tremendous amount of progress has been made over the years in dealing with the challenges of design tools and in improving the efficiency of verification tools. However, reducing the gap between the two remains a challenge.

Verification tools are based on mathematical logic as the calculus of computation. Logic formulae are used to describe states, transformations, and desired properties of systems. Therefore, tools that can work on logical formulae are at the core of the verification tools – a particular example is *Satisfiability Modulo Theories* (SMT) solvers which allow for the symbolic representation of and reasoning on many expressive forms of software and hardware systems. We refer to the problem of deciding the satisfiability of a first-order logic formula with respect to some theories of interest as SMT and the problem of analyzing (e.g., invariant checking) systems described using SMT formulae as *Verification Modulo Theories* (VMT).

There has been a lot of progress in the last decade, on the development of powerful and effective SMT and VMT techniques and tools for the quantifier-free theories of linear arithmetic and uninterpreted functions. This progress has enabled to solve practical problems in different application domains, due to the expressiveness offered by the linear arithmetic theories and the efficiency of the tools. Yet, many applications require even more expressive theories (e.g., railways, aerospace, control software, and cyber-physical systems), and it is a fundamental challenge to go beyond the linear case, by introducing *nonlinear polynomials* and *transcendental functions* such as exponentiation and trigonometric functions.

Modern SMT solvers combine propositional satisfiability (SAT) solvers with decision procedures for first-order theories. The focus of earlier developments in SMT solvers has been on relatively easier theories like linear arithmetic, and that has brought success in many application domains, e.g., verification, planning, scheduling, security, testing, synthesis, etc. This success in the scale of problems that SMT solvers can solve is partly due to the enormous progress in SAT solvers, and also because of the innovation in core decision procedures, efficient data structures, heuristics, and paying attention to implementation details. The progress of SMT solvers is quite evident from the annual SMT competitions [CSW15], which also serve as a driving force for the development of SMT solvers. However, the attention on the theory of nonlinear arithmetic is relatively recent. In fact, there were not many benchmarks for nonlinear arithmetic in the SMT-LIB repository – SMT-LIB [BFT16] provides the SMT standard for expressing different theories and the benchmarks – until recently. Currently, the research for nonlinear arithmetic in SMT focuses on building scalable solvers. The situation of the theory of nonlinear arithmetic extended with transcendental functions is even weaker; there is no standard (and also no benchmarks) for that theory in SMT-LIB.

VMT techniques rely on scalability and efficiency of SMT solvers for checking the satisfiability of SMT formulae. They also exploit the incremental interface of SMT solvers by asking small incremental queries. Moreover, powerful VMT approaches (like IC3ia, techniques based on interpolation) require extended features like unsatisfiable cores, interpolants, etc., from SMT solvers. The success of VMT methods for the theories of linear arithmetic and uninterpreted functions is because modern SMT solvers can fulfill such requirements. However, this is not the case when we move from the linear to the nonlinear case. In fact, there are not many verification tools that can support nonlinear arithmetic and even less for the theory of nonlinear arithmetic extended with transcendental functions.

1.1 Contributions

In this thesis, we address the challenge of dealing with the quantifier-free theory of nonlinear arithmetic, also with transcendental functions, in SMT as well as in VMT. Moreover, we also focus on the issue of integrating verification tools in two of the widely-used design tools.

We propose a practical and unifying approach, referred to as *Incremental Linearization*, that trades the use of expensive, exact solvers for nonlinear arithmetic for an abstraction-refinement loop on top of much less expensive solvers for linear arithmetic and uninterpreted functions. A key feature of the approach is that the linearization is performed incrementally and only when and where needed, driven by spurious counterexamples. Note that we do not discuss the VMT case for nonlinear arithmetic over the integers mainly because we currently do not have the infrastructure (tools to compare against, benchmarks, ...) for evaluation. However, we remark that in principle, the incremental linearization approach presented in the thesis should also work in that case, and the evaluation is left as future work.

Our contributions are the following:

1. We address the SMT problem for nonlinear arithmetic over the reals, nonlinear arithmetic over the reals extended with transcendental functions, and nonlinear arithmetic over the integers, by presenting a novel approach – incremental linearization. Incremental linearization for the reals has been presented in [CGI$^+$17a], but no experimental evaluation has been reported in the paper. The work for transcendental functions has been published in [CGI$^+$17b] and the work for integers has been accepted for publication [CGI$^+$18]. In the thesis, we present the approach in a unified way and also provide an extensive experimental evaluation. Key features of our approach include the ability to provide provably correct linear approximations for the multiplication function as well as transcendental functions, dealing with periodicity property of trigonometric functions, and proving the existence of a solution without explicitly constructing it. We experimentally evaluate our approach on all the SMT-LIB benchmarks from the quantifier-free nonlinear arithmetic categories, as well as with benchmarks from SMT-based verification queries over nonlinear transition systems, including Bounded Model Checking of hybrid automata, several mathematical properties from the MetiTarski suite and from other competitor solver distributions. We compare against various techniques and show that our approach is always competitive and often outperforms state-of-the-art techniques.

2. We address the challenge of checking invariant properties of transition systems expressed over nonlinear arithmetic and transcendental functions, by extending incremental linearization for the VMT case. We give the first IC3-based procedure for checking invariant properties of such systems. The work about VMT for nonlinear arithmetic over the reals has been published in [CGI+17a]. In the thesis, we also show the effectiveness of our approach for the case of transcendental functions. We evaluate on a set of benchmarks collected from various sources, comparing against state-of-the-art VMT techniques; we show that our approach outperforms state-of-the-art techniques.

3. We integrate the nuXmv VMT model checker with Simulink using a black-box compilation approach and with Verilog using a white-box compilation approach. This allows to verify certain classes of Simulink models with nonlinear dynamics, and apply sate-of-the-art VMT techniques on Verilog models which are mostly verified using SAT-based techniques. (Verilog to nuXmv has been presented in [ICG+16].)

4. As a side-product of this work, we have collected a large number of SMT and VMT benchmarks.

Implementation

The prototypes of incremental linearization for SMT and VMT, as presented in [CGI+17a, CGI+17b], were implemented in Python using the PySMT [GM15] library and the Python interfaces of the MathSAT SMT solver and the nuXmv VMT model checker. Now, we have a tighter integration of the approach inside MathSAT and nuXmv, and we report the experimental results using the newer implementations.

1.2 Structure of the Thesis

The thesis is structured into five parts.

In Part I, we give some necessary technical background related to the thesis and survey the state of the art in SMT and VMT over nonlinear arithmetic and transcendental functions. Chapter 2 presents background notions and introduces notations used in the thesis. In Chapter 3, we discuss the SMT problem, mention theories of interest, and overview the state of the art in nonlinear arithmetic and transcendental functions. In Chapter 4, we focus on the VMT problem and techniques, and overview the state of the art on nonlinear arithmetic and transcendental functions. Chapter 5 discusses some design tools used in industry and academia; and mentions some relevant verification tools that are developed in our group.

In Part II, we present our contributions for the SMT case. In Chapter 6, we first outline the general ideas, and then provide details on the refinement mechanisms and on the detection of satisfiable results. We also prove the correctness of the overall approach. In the end of the chapter, we discuss related work. In Chapter 7, we provide some heuristics and details of the implementation of the approach inside MathSAT. We conclude the chapter with experimental evaluation of our technique on a large number of benchmarks, comparing MathSAT against different solvers.

In Part III, we present our contributions for the VMT case. In Chapter 8, we describe the extension of incremental linearization from the SMT to the VMT case. We also give the correctness proofs and in the end we mention related work. In Chapter 9, we discuss some of the heuristics and implementation details of incremental linearization in nuXmv. Then, we present the experimental results comparing nuXmv against other VMT tools over a collection of VMT benchmarks.

Part IV is devoted for integration of verification tools with design tools.

In Chapter 10, we describe a black-box compilation approach to integrate nuXmv with Simulink based on a black-box compiler. Then, a white-box compilation approach to integrate nuXmv with a Verilog tool is presented in Chapter 11. In Chapter 12, we evaluate the Simulink to nuXmv flow on an industrial-level case study and the Verilog to nuXmv flow on a set of benchmarks.

Finally, we conclude the thesis in Chapter 13 by overviewing the contributions and highlighting some future research directions.

Part I

Background and State of the Art

Chapter 2

Technical Background

In this chapter, we introduce the basic notation and notions that are used throughout the rest of the thesis. First, we introduce a basic first-order language and notation. Then, we discuss nonlinear arithmetic and transcendental functions. Later, we also introduce concepts from differential calculus.

Preliminaries

Our setting is standard first-order logic. Let Σ be the signature containing function and predicate symbols. Let $\mathcal{V}$ be a set of variables. A 0-ary function symbol is called a *constant.* A 0-ary predicate symbol is called a *Boolean atom.* A Σ*-term* is a constant or a variable in $\mathcal{V}$ or it is built by applying function symbols in Σ to Σ-terms. If $t_1, \ldots, t_n$ are Σ-terms and R is a predicate with arity n, then $R(t_1, \ldots, t_n)$ is a Σ-atom. A Σ*-literal* is a Σ-atom l or its negation $\neg l$. A *clause* is a disjunction of Σ-literals. A Σ*-formula* is a Σ-literal, the application of the binary logical operators $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ to Σ-formulae, and the quantifiers $\exists$, $\forall$ to an individual variable and a Σ-formula. We call a Σ-formula *quantifier-free* if it does not contain quantifiers. A variable is *free* in a Σ-formula if it is not in the scope of a quantifier in the formula. A Σ*-sentence* is a Σ-formula without free variables. A formula is in *conjunctive normal form* (CNF) if it is expressed as a conjunction of clauses.

The semantics of a Σ-formula is given by a Σ-*interpretation* $\mathfrak{I}$, which is a pair $(D_{\mathfrak{I}}, \alpha_{\mathfrak{I}})$: where $D_{\mathfrak{I}}$ is a domain and $\alpha_{\mathfrak{I}}$ is an assignment. The assignment $\alpha_{\mathfrak{I}}$ maps:

- each constant and each variable to a value from $D_{\mathfrak{I}}$,
- each n-ary function symbol f to an n-ary function

  $$f_{\mathfrak{I}} : D_{\mathfrak{I}}^n \rightarrow D_{\mathfrak{I}}$$

  that maps n elements of $D_{\mathfrak{I}}$ to an element of $D_{\mathfrak{I}}$,
- each n-ary predicate symbol R to an n-ary predicate

  $$R_{\mathfrak{I}} : D_{\mathfrak{I}}^n \rightarrow \{\top, \bot\}$$

  that maps n elements of $D_{\mathfrak{I}}$ to a truth value ($\top$ means **true** and $\bot$ means **false**).

Evaluation of the logical connectives $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$, and $\neg$ is done using their standard semantics. $\mathfrak{I}$ is a Σ-*model* of a quantifier-free Σ-formula ψ, written as $\mathfrak{I} \models \psi$ if ψ evaluates to $\top$ under $\mathfrak{I}$; or $\mathfrak{I}$ is not a Σ-model of ψ, written as $\mathfrak{I} \not\models \psi$ if ψ evaluates to $\bot$ under $\mathfrak{I}$. $\mathfrak{I} \models \exists x.\psi$ iff there exits a value $v \in D_{\mathfrak{I}}$ such that $\mathfrak{I}$ with the mapping $x \rightarrow v$ is a Σ-model for ψ. $\mathfrak{I} \models \forall x.\psi$ iff, for all the values $v \in D_{\mathfrak{I}}$, $\mathfrak{I}$ with the mapping $x \rightarrow v$ is a Σ-model for ψ. We may also say of the Σ-model $\mathfrak{I}$ of the Σ-formula ψ that $\mathfrak{I}$ *satisfies* ψ.

A Σ-formula ψ is *satisfiable* (*unsatisfiable* resp.) iff there exits (does not exist resp.) a Σ-model of ψ. The formula ψ is *valid* iff all Σ-interpretations are Σ-models of ψ. The Σ-formula φ is a *logical consequence* of the Σ-formula ψ, denoted as $\psi \models \varphi$, iff for every $\mathfrak{I}$ such that $\mathfrak{I} \models \psi$ it also holds that $\mathfrak{I} \models \varphi$.

Definition 2.1. *A* Σ-theory $\mathcal{T}$ *is a (possibly infinite) set of* Σ*-models.*

Definition 2.2. *A* Σ*-formula* φ *is:*

- satisfiable *in* $\mathcal{T}$ *(or* $\mathcal{T}$-satisfiable*), if* φ *is satisfiable in a* Σ*-model from* $\mathcal{T}$*;*
- valid *in* $\mathcal{T}$ *(or* $\mathcal{T}$-valid*), if* φ *is satisfiable in every* Σ*-model in* $\mathcal{T}$*;*
- unsatisfiable *in* $\mathcal{T}$ *(or* $\mathcal{T}$-unsatisfiable*), if there is no* Σ*-model in* $\mathcal{T}$ *that makes* φ *satisfiable (or* φ *is not* $\mathcal{T}$*-satisfiable).*

We call a *theory solver* for Σ-theory $\mathcal{T}$ ($\mathcal{T}$-solver) any procedure establishing whether any given finite conjunction (or finite set) of Σ-literals is $\mathcal{T}$-satisfiable or not.

Definition 2.3. *Two* Σ*-formulae* φ *and* ψ *are:*

- $\mathcal{T}$-equivalent*, if* φ *and* ψ *have the same* Σ*-models from* $\mathcal{T}$*;*
- $\mathcal{T}$-equisatisfiable*, if* φ *is* $\mathcal{T}$*-satisfiable iff* ψ *is* $\mathcal{T}$*-satisfiable.*

The *quantifier elimination* is an approach to construct a $\mathcal{T}$-equivalent quantifier-free formula to a given quantified formula [BM07]. A theory admits quantifier elimination if there exists an algorithm that solves the quantifier elimination problem.

For simplicity, we may omit the "Σ-" prefix from term, atom, formula, theory, interpretation, etc. We may call an interpretation of $\mathcal{T}$ a $\mathcal{T}$-interpretation. Similarly, we may call a model from $\mathcal{T}$ a $\mathcal{T}$-model. We denote formulae with φ, ψ, I, T, P, terms with t, s, variables with x, y, constants with a, b, c, functions with f, TF, each possibly with subscripts. If X is a set of variables, we write $\varphi(X)$ to denote the fact that all the variables of φ are in X. We denote with $\varphi\{x \mapsto t\}$ the formula obtained by replacing all the free occurrences of x in φ with t; and we use the same notation for terms and models, and we extend it to ordered sequences of distinct variables in the natural way. (E.g., if $\mathbf{x} \doteq x_1, .., x_k$ and $\mathbf{t} \doteq t_1, .., t_k$, then $\varphi\{\mathbf{x} \mapsto \mathbf{t}\}$ denotes $\varphi\{x_1 \mapsto t_1\}\{... \mapsto ...\}\{x_k \mapsto t_k\}$.) If Γ is a set of formulae, we write $\bigwedge \Gamma$ (or simply Γ) to denote the conjunction of all the formulae in Γ. We abuse the notation and write $t \in \varphi$ to denote that term t occurs in φ. If μ is an interpretation and x is a variable, we write $\mu[x]$ to denote the value of x in μ, and we extend this notation to terms and formulae in the usual way. $abs(t)$ stands for $\textsc{ite}(t < 0, -t, t)$, ITE [KSJ09] being the standard if-then-else term operator. (The semantics of an ITE term is the usual semantics of if-then-else semantics from programming languages.) We write $t_1 < t_2 < t_3$ for $t_1 < t_2 \wedge t_2 < t_3$. (A similar notation is used with "$\leq$".)

We define a bijective function $\mathcal{T}2\mathcal{B}$ (theory to Boolean) and its inverse $\mathcal{B}2\mathcal{T} \doteq \mathcal{T}2\mathcal{B}^{-1}$ (Boolean to theory) for formulae, such that $\mathcal{T}2\mathcal{B}$ maps Boolean atoms into themselves and non-Boolean atoms into fresh Boolean atoms. We call $\mathcal{T}2\mathcal{B}(\varphi)$ a *Boolean abstraction* of a formula φ. We call a *truth assignment* μ_T for a formula φ a truth value assignment to the atoms of φ. We represent μ_T using a set of literals in φ such that for each atom in φ either A or $\neg A$ is present in it, and $A \in \mu_T$ means that A is assigned $\top$ whereas $\neg A \in \mu_T$ means that A is assigned $\bot$. We write $\Gamma \models \varphi$ to denote that φ is a logical consequence of the set Γ of formulae. μ_T is a *propositionally satisfying truth assignment* for a formula φ if $\mathcal{T}2\mathcal{B}(\bigwedge \mu_T) \models \mathcal{T}2\mathcal{B}(\varphi)$.

Example 2.4. Consider the following formula φ:

$$\varphi \doteq (x \leq y) \wedge ((x + 3 = z) \vee (z \geq y)).$$

Then

$$\mathcal{T}2\mathcal{B}(\varphi) \doteq p \wedge (q \vee r)$$

where p, q, r are Boolean variables. A truth assignment μ for φ can be

$$\mu_T \doteq \{(x \leq y), \neg(x + 3 = z), (z \geq y)\}.$$

μ_T is also a propositionally satisfying truth assignment. △

Nonlinear Arithmetic and Transcendental Functions.

We denote with $\mathbb{Z}$, $\mathbb{Q}$ and $\mathbb{R}$ the set of integer, rational and real numbers, respectively. The absolute value of $a \in \mathbb{R}$, denoted by $|a|$, is defined as $|a| = a$ if $a \geq 0$, and $-a$ otherwise. A *monomial* m in variables $x_1, x_2, \ldots, x_n$ is a product $x_1^{\alpha_1} * x_2^{\alpha_2} * \ldots * x_n^{\alpha_n}$, where each α_i is a non-negative integer called *exponent* of the variable x_i. When clear from the context, we may omit the multiplication symbol $*$ and simply write $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$. A *polynomial* p is of the form: $a_n * m_n + a_{n-1} * m_{n-1} + \ldots + a_0$, where $a_n, a_{n-1}, \ldots, a_o$ are constants with the constraint $a_n \neq 0$ and are called *coefficients*, and $m_n, m_{n-1}, \ldots, m_1$ are monomials. We write $\mathbb{Q}[x_1, \ldots, x_n]$ as the sets of the polynomials containing the variables $x_1, \ldots, x_n$ with all the coefficients in $\mathbb{Q}$. A *univariate polynomial* is a polynomial containing one variable, whereas a *multivariate polynomial* contains more than one variable. A *polynomial constraint* P is of the form $p \bowtie 0$ where p is a polynomial and $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$.

The *total degree* of a monomial is the sum of the exponents of its variables. The *total degree* of a polynomial is the highest degree among its monomials. A monomial is *linear* if it has total degree less than or equal to one, otherwise it is *nonlinear*, and similarly for polynomials. A polynomial constraint $p \bowtie 0$ is a *linear constraint* if p is linear and is a *nonlinear constraint* if p is nonlinear.

A real number $c \in \mathbb{R}$ is a *real root* of $p \in \mathbb{Q}[x]$ iff $p(c) = 0$. A real number $a \in \mathbb{R}$ is an *algebraic* number iff it is a root of some $p \in \mathbb{Q}[x]$, otherwise it is *transcendental number*. An example of algebraic number is

$\sqrt{2}$, while π and e are transcendental numbers.

A function over the reals $f : \mathbb{R}^n \rightarrow \mathbb{R}$ maps every element in $\mathbb{R}^n$ into a corresponding element in $\mathbb{R}$. A function $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is called *univariate* when $n = 1$, *bivariate* when $n = 2$, and *multivariate* when $n > 1$. A function $y = f(x_1, \ldots, x_n)$ is *algebraic* iff it satisfies a polynomial equation, i.e., iff there exists a polynomial $p \in \mathbb{Q}[y, x_1, \ldots, x_n]$ such that $\forall x_1, \ldots, x_n.(p(y, x_1, \ldots, x_n) = 0)$. A function is *transcendental* if it is not algebraic [Tow07, Haz93].

We consider continuous and differentiable functions. If f is a univariate function, we write $\frac{d}{dx}f$ for the first-order derivative of f. We also write $f^{(i)}$ for the i-th derivative of f, and f' and f'' for $f^{(1)}$ for $f^{(2)}$, respectively.

Let l and u be two real numbers. We denote open and closed intervals between them as $]l, u[$ and $[l, u]$ respectively. Given a univariate function f over the reals, the *graph* of f is the set of pairs $\{(x, f(x)) \mid x \in \mathbb{R}\}$. We refer to an element $(x, f(x))$ of the graph as a point of f.

Definition 2.5. *The* tangent line *at $a \in \mathbb{R}$ to a univariate function f, denoted with $\textsc{TanLine}_{f,a}(x)$, is the straight line defined as follows:*

$$\textsc{TanLine}_{f,a}(x) \doteq f(a) + f'(a) * (x - a)$$

Definition 2.6. *Given $a, b \in \mathbb{R}$ and $a < b$, the* secant line *at $[a, b]$ to a univariate function f, denoted with $\textsc{SecLine}_{f,a,b}(x)$, is the straight line defined as follows:*

$$\textsc{SecLine}_{f,a,b}(x) \doteq \frac{f(a) - f(b)}{a - b} * (x - a) + f(a).$$

Definition 2.7. *Let f be a univariate function twice differentiable at a point c. The* concavity *of f at c is the sign of $f''(c)$.*

Definition 2.8. *A univariate function f has an* inflection point *at c iff it is twice differentiable at c, $f''(c) = 0$, and there exists $\epsilon > 0$ such that for* *all $x \in [c-\epsilon, c[$ the value $f(x)$ has sign $s \neq 0$ and for all $x \in]c, c+\epsilon]$ the value $f(x)$ has sign $s' \neq 0$ opposite to s.*

Proposition 2.9. *Let f be a univariate function. If $f''(x) \geq 0$ for all $x \in [l,u]$, then for all $a, x \in [l,u]$ $\textsc{TanLine}_{f,a}(x) \leq f(x)$, and for all $a, b, x \in [l,u]$ $((a \neq b \wedge a \leq x \leq b) \rightarrow \textsc{SecLine}_{f,a,b}(x) \geq f(x))$.*

If $f''(x) \leq 0$, then the dual property holds.

Let $f(x,y)$ be a bivariate function. We write $\frac{d}{dx}f(x,y)$ and $\frac{d}{dy}f(x,y)$ for the first-order partial derivatives of $f(x,y)$ w.r.t. x and y, respectively.

Definition 2.10. *The* tangent plane *at a point (a,b) to a bivariate function $f(x,y)$, denoted with $\textsc{TanPlane}_{f,a,b}(x,y)$, is defined as follows:*

$$\begin{aligned}\textsc{TanPlane}_{f,a,b}(x,y) \doteq\ & f(a,b) + \tfrac{d}{dx}f(x,y)\{x \mapsto a\}\{y \mapsto b\} * (x-a) \\ & + \tfrac{d}{dy}f(x,y)\{x \mapsto a\}\{y \mapsto b\} * (y-b)\end{aligned}$$

Taylor Series and Taylor's Theorem.

Definition 2.11. *Let $f(x)$ be n-differentiable at a point a. The* Taylor series *of f of degree n centered around a is the polynomial:*

$$P_{n,f,a}(x) \doteq \sum_{i=0}^{n} \frac{f^{(i)}(a)}{i!} * (x-a)^i$$

The Taylor series centered around zero is also called *Maclaurin series.*

According to *Taylor's theorem*, any continuous function $f(x)$ that is $(n+1)$-differentiable can be written as the sum of the Taylor series and the remainder term:

$$f(x) = P_{n,f,a}(x) + R_{n+1,f,a}(x)$$

where $R_{n+1,f,a}(x)$ is the Lagrange form of the remainder, expressible as

$$R_{n+1,f,a}(x) \doteq \frac{f^{(n+1)}(b)}{(n+1)!} * (x-a)^{n+1}.$$

for some b between x and a.

Although the value of b is not known, an upper bound on the size of the remainder $R^{\mathrm{U}}_{n+1,f,a}(x)$ at a point x can be defined as:

$$R^{\mathrm{U}}_{n+1,f,a}(x) \doteq \max_{c \in [\min(a,x), \max(a,x)]} (|f^{(n+1)}(c)|) * \frac{|(x-a)^{n+1}|}{(n+1)!}.$$

From this, we obtain a lower- and an upper-bound for $f(x)$, given by $P_{n,f,a}(x) - R^{\mathrm{U}}_{n+1,f,a}(x)$ and $P_{n,f,a}(x) + R^{\mathrm{U}}_{n+1,f,a}(x)$ respectively. Clearly, the closer is a to x, the tighter the approximation of $f(x)$ will be.

In this thesis we consider univariate exponential and trigonometric transcendental functions. We recall that the graphs of exponential and trigonometric functions have only a finite number of points in $\mathbb{Q} \times \mathbb{Q}$ (e.g., $(0, 1)$ for exp, $(0, 0)$ for sin). Finally, we recall that trigonometric functions like sin are periodic. For example, for all $i \in \mathbb{Z}$, $\sin(a) = \sin(a + 2i\pi)$.

Chapter 3

Satisfiability Modulo Theories

In this chapter, we define the SAT and SMT problems, and we overview the SMT solving approach followed by most modern SMT solvers. We also mention some of the features and the extended functionalities provided by the solvers. We then introduce the theories relevant to the thesis and discuss the state of the art on the theories.

3.1 The Propositional Satisfiability Problem

Propositional Satisfiability (SAT) is the problem of determining whether a given formula over Boolean variables has a satisfying truth assignment. It has been shown to be NP-Complete [Coo71].

A SAT solver is an implementation of a procedure for solving the SAT problem. Modern SAT solvers, for instance MiniSat [ES03a], take as input a propositional formula in CNF. Most SAT solvers are based on the DPLL procedure [DP60, DLL62] with the clause learning technique [SS99]. The main idea is to construct an interpretation in such a way that all the clauses in the given CNF formula are satisfied. This is performed by searching, i.e., interleaving literal decisions with unit propagations. When a conflicting clause is found, a new clause is learned via conflict analysis and backtracking is performed. If the solver finds all clauses to be satisfied under

```
**bool** CHECKSMT($\varphi$, $\mathcal{T}$):
$\varphi' :=$ PREPROCESS($\varphi$)
$\varphi_B := \mathcal{T}2\mathcal{B}(\varphi')$
**while true**:
    $\langle sat, \mu_B \rangle :=$ CHECKSAT($\varphi_B$)
    **if not** $sat$:
        **return false**
    $\langle sat, \gamma \rangle := \mathcal{T}$-SOLVER($\mathcal{B}2\mathcal{T}(\mu_B)$)
    **if** $sat$:
        **return true**
    $\varphi_B := \varphi_B \land \neg\mathcal{T}2\mathcal{B}(\gamma)$
```

Figure 3.1: An abstract procedure for SMT solving using the lazy/DPLL($\mathcal{T}$) approach

the constructed interpretation, the problem is satisfiable, and if it finds a conflict without making any decision, the problem is unsatisfiable.

3.2 The Satisfiability Modulo Theories Problem

Satisfiability Modulo Theories (SMT) is the problem of deciding the $\mathcal{T}$-satisfiability of a formula with respect to some theory $\mathcal{T}$ or combination of theories ($\mathcal{T}_1 \cup \mathcal{T}_2$). The SMT problem is NP-hard since it subsumes the question of checking the satisfiability of propositional formulae.

An *SMT solver* implements a procedure that solves the SMT problem, e.g., Z3 [dMB08b], YICES [Dut14], MATHSAT [CGSS13], CVC4 [BCD+11], VERIT [BODF09], etc. The most efficient implementations of SMT solvers use the so-called "lazy approach"/DPLL($\mathcal{T}$) [Seb07, BSST09, NOT06]), where a SAT solver is tightly integrated with a $\mathcal{T}$-solver – e.g., a very abstract procedure is shown in Fig. 3.1.

As a first step, SMT solvers apply as preprocessing some satisfiability-preserving simplifications and CNF transformations. The role of the SAT solver is to enumerate truth assignments to the Boolean abstraction of the preprocessed formula. If the Boolean abstraction is found to be unsatisfiable by the SAT solver then the input formula is also unsatisfiable. When the SAT solver finds a satisfying truth assignment μ_B, then a $\mathcal{T}$-solver is invoked to check if $\mathcal{B}2\mathcal{T}(\mu_B)$ is $\mathcal{T}$-satisfiable. In that case, the original formula is also $\mathcal{T}$-satisfiable. Otherwise, the $\mathcal{T}$-solver returns a conflict set γ which identifies a reason for the unsatisfiability. Then, $\neg\mathcal{T}2\mathcal{B}(\gamma)$ (called *$\mathcal{T}$-lemma*) is learned by the SAT solver to prune the search.

Besides deciding $\mathcal{T}$-satisfiability and generating conflict sets, modern $\mathcal{T}$-solvers support several other features relevant to SMT($\mathcal{T}$). Here we recall two other notable features.

Model Generation. When invoked on a $\mathcal{T}$-satisfiable set of literal Γ, a model generating $\mathcal{T}$-solver can return a $\mathcal{T}$-model μ which is a witness for the consistency of Γ in $\mathcal{T}$, i.e., $\mu \models_{\mathcal{T}} \bigwedge \Gamma$.

Incrementality and back-trackability. In the lazy/DPLL($\mathcal{T}$) approach to SMT($\mathcal{T}$), a $\mathcal{T}$-solver is often invoked in a stack-based manner. Thus *incrementality* and *back-tractability* of a $\mathcal{T}$-solver plays a vital role in the overall efficiency of the approach. *Incremental* means that $\mathcal{T}$-solver "remembers" its computation status from one call to the other, so that, whenever a set of literals $\Gamma \doteq \Gamma_1 \cup \Gamma_2$ is given as input such that Γ_1 has been just proved to be $\mathcal{T}$-satisfiable, it avoids restarting the computation from scratch by resuming the calculation from the previous status. *Back-tractable* means that it is possible to undo steps and efficiently return to a prior computation status on the stack.

Theories of Interest

All the theories we consider are the first-order theories with equality. Thus every theory contains the following axioms for every function and every predicate:

$$
\begin{aligned}
&\forall x.(x = x)\\
&\forall x, y.(x = y \to y = x)\\
&\forall x, y, z.((x = y \wedge y = z) \to x = z)\\
&\forall x_1, \ldots, x_n, y_1, \ldots, y_n.((\bigwedge_{i=1}^{n} x_i = y_i) \to f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n))\\
&\forall x_1, \ldots, x_n, y_1, \ldots, y_n.((\bigwedge_{i=1}^{n} x_i = y_i) \to (R(x_1, \ldots, x_n) \leftrightarrow R(y_1, \ldots, y_n)))
\end{aligned}
$$

Now we discuss some theories relevant to this thesis.

Linear Arithmetic

The *theory of linear arithmetic* ($\mathcal{LA}$) on the rationals ($\mathcal{LRA}$) and on the integers ($\mathcal{LIA}$) is the first-order theory whose atoms are linear polynomial constraints. $\mathcal{LRA}$ ($\mathcal{LIA}$ resp.) consists of the set of models that interprets the symbols of linear polynomial constraints in the usual way over $\mathbb{Q}$ ($\mathbb{Z}$ resp.).

The $\mathcal{LRA}$-satisfiability of a conjunction of literals is decidable and polynomial [Kar84]. The main algorithms for $\mathcal{LRA}$ are variants of Simplex and Fourier-Motzkin algorithms, both of which are not the best due to the exponential complexity. In fact, many SMT solvers for $\mathcal{LRA}$ are based on a Simplex variant presented in [DdM06], which provides efficient support for incrementality and back-tractability.

The $\mathcal{LIA}$-satisfiability of a conjunction of literals is decidable and NP-complete [Pap81]. Most SMT solvers approach the $\mathcal{LIA}$-satisfiability by combining an $\mathcal{LRA}$ solving technique with branch-and-bound and the Gomory's cutting plane methods. Similar to $\mathcal{LRA}$, efficient incremental and back-tractable procedures for $\mathcal{LIA}$ have been conceived – e.g., [Gri12].

Nonlinear Arithmetic

The *theory of nonlinear arithmetic* ($\mathcal{NA}$) on the reals ($\mathcal{NRA}$) and on the integers ($\mathcal{NIA}$) is the first-order theory whose atoms are (both linear and nonlinear) polynomial constraints. $\mathcal{NRA}$ ($\mathcal{NIA}$ resp.) consists of the set of models that interprets the symbols of polynomial constraints in the usual way over $\mathbb{R}$ ($\mathbb{Z}$ resp.).

We denote with $\mathcal{NTA}$ the theory of $\mathcal{NRA}$ extended with the transcendental functions: the exponential function and the trigonometric functions, and the transcendental number π.

The $\mathcal{NRA}$-satisfiability of a conjunction of literals is decidable and doubly-exponential [DH88, Wei88, BD07], whereas the case of $\mathcal{NIA}$-satisfiability is undecidable [Mat93], as well as the case of $\mathcal{NTA}$-satisfiability is undecidable [Ric68].

We discuss different solving approaches to SMT($\mathcal{NRA}$), SMT($\mathcal{NIA}$), and SMT($\mathcal{NTA}$) in Section 3.3.

Uninterpreted Functions

The *theory of uninterpreted functions* ($\mathcal{UF}$) is the first-order theory with no restriction on Σ. The $\mathcal{UF}$-satisfiability of conjunctions of literals is decidable and polynomial [Ack54]. An $\mathcal{UF}$-solver is usually implemented on top of data structures and algorithms for computing the *congruence closure* of a set of terms [NO07] – providing important features such as efficient incrementality and back-tractability.

Theory of Bit Vectors

The *theory of fixed-width bit vectors ($\mathcal{BV}$)* is a first-order theory with equality which aims at representing Register Transfer Level (RTL) hardware circuits. It can also be used to encode software verification problems.

The $\mathcal{BV}$-satisfability of conjunctions of literals is decidable and NP-complete. A typical approach to SMT($\mathcal{BV}$) is to apply some word-level preprocessing, and then encode the result into a SAT problem – also known as "bit blasting".

Theory of Arrays

The *theory of arrays* ($\mathcal{AR}$) is a multi-signature theory: it has a signature for index, a signature for array element, and a signature for arrays. The array signature contains two functions namely *read* (of arity two) and *write* (of arity three), and contains one equality predicate. The read function returns the array element at a given index, and the write function returns a new array with one element updated at the given index with the given value while other elements remain unchanged. The equality predicate provides comparison on two arrays.

Axioms for the theory of arrays are:

$$\forall a.\forall i.\forall e.(read(write(a,i,e),i) = e) \tag{3.1}$$

$$\forall a.\forall i.\forall j.\forall e.((i \neq j) \rightarrow read(write(a,i,e),j) = read(a,j)) \tag{3.2}$$

$$\forall a.\forall b.(\forall i.(read(a,i) = read(b,i)) \rightarrow (a = b)) \tag{3.3}$$

The first two rules are called the *McCarthy's axioms* and the last one is called the *extensionality axiom.* The presented theory of arrays is decidable and its complexity is NP-complete, for more results on decidability we suggest the reference [BMS06].

Theory Combinations

In many practical applications of SMT, the theory $\mathcal{T}$ is a combination of two or more theories $\mathcal{T}_1, \ldots, \mathcal{T}_n$. Most modern SMT solvers rely on the *Nelson Oppen logical framework* [NO79, Opp80] for dealing with theory combinations. To this extent, various improvements have been achieved over the past decade, most noticeably the *Delayed Theory Combination* [BBC+06] schema and its extension with "model-based heuristic" [dMB08a]. For interested readers, here are some other references [Fon09, CFR14].

In this thesis, we are interested in using the combination of $\mathcal{UF}$ with $\mathcal{LA}$: we denote with $\mathcal{UFLRA}$ the combined theory of $\mathcal{UF}$ and $\mathcal{LRA}$, $\mathcal{UFLIA}$ the combined theory of $\mathcal{UF}$ and $\mathcal{LIA}$.

Extended SMT Functionality

Many applications of SMT require functionalities beyond just checking the satisfiability of SMT formulae. Here we recall two of such extended functionalities relevant to this thesis.

Extraction of Unsatisfiable Cores

Given an unsatisfiable CNF formula φ, we say that an unsatisfiable CNF formula ψ is an *unsatisfiable core* of φ, if and only if $\varphi \doteq \psi \wedge \psi'$ where ψ' is a CNF formula (possibly $\top$). Several algorithms have been proposed for the unsatisfiable core extraction of propositional formulae over the last decade. However, there are not many approaches in the case of SMT. In fact, a prominent and simple technique for the unsatisfiable core extraction of SMT formulae is the *Lemma-Lifting approach*, which uses a propositional unsatisfiable core extractor under the hood. Let $\Gamma \doteq \{D_1, \ldots, D_k\}$ be the set of $\mathcal{T}$-lemmas learned by an SMT($\mathcal{T}$) solver when proving unsatisfiability of a given formula $\varphi \doteq C_1 \wedge \ldots \wedge C_n$. In order to extract an unsatisfiable core of φ, a propositional unsatisfiable core extractor is invoked on the propositional problem $\mathcal{T}2\mathcal{B}(\varphi \wedge \bigwedge \Gamma)$. Suppose the unsatisfiable core returned by the propositional unsatisfiable core extractor is $\mathcal{T}2\mathcal{B}(C'_1 \wedge \ldots \wedge C'_m \wedge D'_1 \wedge \ldots \wedge D'_j)$, then an unsatisfiable core of φ is $C'_1 \wedge \ldots \wedge C'_m$.

Computation of Craig Interpolants

Let ϕ and ψ be two formulae, $\phi \preceq \psi$ denotes that all the uninterpreted symbols of ϕ occur in ψ.

Definition 3.1. *Given an ordered pair $\langle \phi, \psi \rangle$ of formulae such that $\phi \wedge \psi \models_{\mathcal{T}} \bot$, a Craig interpolant (or simply interpolant) I is defined as follows:*

- $\phi \models_{\mathcal{T}} I$,
- $\psi \wedge I \models_{\mathcal{T}} \bot$,
- $I \preceq \phi$ *and* $I \preceq \psi$.

There has been significant progress in the computation of interpolants in $\mathcal{LA}$ [CGS10, GLS10]. However, in the case of $\mathcal{NA}$, it is still a challenging problem – some attempts include [KB11, GZ16].

3.3 State of the Art

We now discuss the state of the art on $\mathcal{NRA}$, $\mathcal{NTA}$, and $\mathcal{NIA}$.

For SMT($\mathcal{NRA}$) various techniques have been explored, including complete methods based on quantifier elimination and convex programming, and incomplete methods based on interval constraint propagation and linearization. In the case of SMT($\mathcal{NIA}$), most SMT solvers rely on the bit-blasting approach. Some SMT solvers opt for interval constraint propagation and linearization. A more recent treatment to SMT($\mathcal{NIA}$) is to combine the SMT($\mathcal{NRA}$) solving techniques with the branch-and-bound method. For SMT($\mathcal{NTA}$) solving, the approaches include methods based on interval constraint propagation and deductive methods.

First, we focus on the common approaches to $\mathcal{NRA}$, $\mathcal{NTA}$, and $\mathcal{NIA}$, and then we discuss the specific ones.

$\mathcal{NRA}$, $\mathcal{NTA}$, and $\mathcal{NIA}$– Common Approaches

Interval Constraint Propagation

Interval constraint propagation (ICP) [BG06] is an incomplete technique for solving constraints over $\mathcal{NRA}$, $\mathcal{NIA}$, and $\mathcal{NTA}$. A key feature of ICP is to detect inconsistency when the domain of a problem is bounded. Initially, it has been investigated in [GB06, Rat06]. Now it has been integrated into several SMT solvers, most noticeably raSAT [TKO16] for $\mathcal{NRA}$; iSAT3 [FHT+07] for $\mathcal{NIA}$ and $\mathcal{NTA}$; and dReal [GKC13] for $\mathcal{NTA}$.

Interestingly, dReal relies on the notion of delta-satisfiability [GAC12], which guarantees that there exists a variant (within a user-specified δ "radius") of the original problem such that it is satisfiable. The approach cannot ensure that the original problem is satisfiable since it relies on numerical approximation techniques that only compute safe over-approximations of the solution space.

In contrast to dReal, iSAT3 and raSAT may find solutions to problems in some cases.

Linearization

A recent development in solving SMT($\mathcal{NRA}$) is the method of subtropical satisfiability [FOSV17], which is an incomplete method to detect satisfiability of conjunctions of strict inequality constraints. The technique is efficient in returning satisfiable or unknown. It has been implemented in veriT [BODF09]. The method encodes a sufficient condition for satisfiability into an $\mathcal{LRA}$ problem.

In [BLO+12], the SMT($\mathcal{NIA}$) problem is approached by reducing the problem into an SMT($\mathcal{LIA}$) problem via linearization. The linearization is performed by doing case analysis on the variables appearing in nonlinear monomials. The method is geared towards detecting satisfiable instances. If the domain of the problem is bounded then the method can generate an equisatisfiable SMT($\mathcal{LIA}$) formula. Otherwise, it solves a bounded problem and incrementally increases the bounds of some variables until it finds a solution to the linear problem. The authors propose to choose variables that appear in the unsatisfiable core, for increasing the bounds. In some cases, it may detect unsatisfiability of the original problem. The paper also presents an extension of the method to find rational solutions with a fixed denominator. This work has been extended in [LORR14], where the variable selection heuristics are improved via Max-SMT [NO06, CFG$^+$10] and Optimization Modulo Theories (OMT) [ST15].

$\mathcal{NRA}$-Specific Approaches

Quantifier Elimination

The two well-known and well-studied quantifier elimination procedures for the theory of nonlinear polynomials are *Cylindrical Algebraic Decomposition* (CAD) [Col74] and *Virtual Substitution* (VS) [Wei97]. (VS targets problems with low-degree – usually up to degree 3 [Stu17].) They have doubly-exponential worst-case complexity [DH88, Wei88, BD07]. Although these procedures have been studied for decades, their use in the SMT($\mathcal{NRA}$) solving is relatively recent. SMT-RAT [CÁ11, CLJÁ12, CKJ$^+$15] represents the first attempt to integrate CAD and VS in an SMT($\mathcal{NRA}$) solver. [1] Then, Z3 [dMB08b] and Yices [Dut14] (winners of the SMT competition 2017 in the QF-$\mathcal{NRA}$ division) also implemented a variant [JdM12] of CAD. Note that Z3 and Yices use a relatively newer framework – nlSAT [JdM12]/MCSAT [dMJ13] framework for SMT solving – that allows for a much tighter integration of the CAD algorithm with the

[1] SMT-RAT also uses Gröbner bases [Stu94] as a theory solver [JLCÁ13].

Boolean search of the solver than the DPLL($\mathcal{T}$) framework.

A critical issue with the $\mathcal{NRA}$-solvers based on quantifier elimination is that they do not yet offer efficient incrementality and back-tractability features [DE17].

Convex Programming

CalCS [NPSS10] is an SMT solver for a restricted subset of $\mathcal{NRA}$, i.e., nonlinear polynomial constraints that are convex. It is a DPLL($\mathcal{T}$) solver where the $\mathcal{T}$-solver is based on the checking feasibility of convex constraints [BV04]. To detect satisfiable cases, it also performs under-approximations using hyper-planes.

$\mathcal{NTA}$-Specific Approaches

Deductive Methods

The MetiTarski [AP10] theorem prover relies on resolution and on a decision procedure for $\mathcal{NRA}$ to prove quantified inequalities involving transcendental functions. It works by replacing transcendental functions with upper- or lower-bound functions specified by means of axioms (corresponding to either truncated Taylor series or rational functions derived from continued fraction approximations), and then using an external decision procedure for $\mathcal{NRA}$ for solving the resulting formulae. MetiTarski cannot prove the existence nor compute a satisfying assignment of a solution. It may require the user to manually write axioms if the ones automatically selected from a predefined library are not enough.

The approach presented in [EKK+11], where the $\mathcal{NTA}$ theory is referred to as NLA, is similar in spirit to MetiTarski in that it combines the SPASS theorem prover [WDF+09] with the iSAT3 SMT solver. The approach relies on the SUP(NLA) calculus that combines superposition-based first-order logic reasoning with SMT($\mathcal{NTA}$).

Combination of Interval Propagation and Theorem Proving. Gappa [dDLM11, MM16] is a standalone tool and a tactic for the Coq proof assistant, that can be used to prove properties about numeric programs (C-like) dealing with floating-point or fixed-point arithmetic. Another related Coq tactic is Coq.Interval [Mel11]. Both Gappa and Coq.Interval combine interval propagation and Taylor approximations for handling transcendental functions. A similar approach is followed also in [SH13], where a tool written in Hol-Light to handle conjunctions of non-linear equalities with transcendental functions is presented. The work uses Taylor polynomials up to degree two. NLCertify [Mag14] is another related tool which uses interval propagation for handling transcendental functions. It approximates polynomials with sums of squares and transcendental functions with lower and upper bounds using some quadratic polynomials [AGMW13]. Internally, all these tools/tactics rely on multi-precision floating point libraries for computing the interval bounds.

$\mathcal{NIA}$-Specific Approaches

Bit-Blasting

In the bit-blasting approach [FGM^{+}07], an $\mathcal{NIA}$-satisfiability problem is iteratively reduced to a SAT problem by first bounding the integer variables, and then encoding the resulting problem to a SAT problem. The SAT problem is then checked by a SAT solver. This approach is geared towards finding models for the $\mathcal{NIA}$ problem, and it can not prove $\mathcal{NIA}$-unsatisfiability unless the $\mathcal{NIA}$ problem is bounded. If the SAT problem is unsatisfiable then the bounds on the integer variables are increased, and the process of encoding to a SAT problem and SAT check is repeated.

Combination of $\mathcal{NRA}$ and Branch-and-Bound

The approach to combine $\mathcal{NRA}$ solving techniques with the branch-and-bound method has been recently investigated in [Jov17] and [KCÁ16]. The main idea is to relax the $\mathcal{NIA}$ problem by treating as if it was an $\mathcal{NRA}$ problem and apply the $\mathcal{NRA}$ techniques for solving it. Since the relaxed problem is an over-approximation of the original problem, the unsatisfiability of the $\mathcal{NIA}$ problem is implied by the unsatisfiability of the $\mathcal{NRA}$ problem. If the $\mathcal{NRA}$-solver finds a non-integral solution a to a variable x, then a lemma $(x \leq \lfloor a \rfloor \vee x \geq \lceil a \rceil)$ is added to the $\mathcal{NRA}$ problem. Otherwise, an integral solution is found for the $\mathcal{NIA}$ problem. In [Jov17], the CAD procedure (as presented in [JdM12]) is combined with branch-and-bound in the MCSAT framework and has been implemented in Yices [Dut14]. In [KCÁ16], the authors have presented how to combine CAD and VS with branch-and-bound in the DPLL($\mathcal{T}$) framework.

Chapter 4

Verification Modulo Theories

Here we introduce the symbolic transition systems and the VMT problem. We also discuss various VMT techniques and later we overview the state of the art on $\mathcal{NRA}$ and $\mathcal{NTA}$.

4.1 Symbolic Transition Systems

Transition systems are used as models to describe the behavior of systems. They specify how systems can evolve from one state to another, where a state describes some information about a system at a certain moment. A transition system is *finite* if the set of states of the system is finite, otherwise it is *infinite*. A prominent class of transition systems is symbolic transition systems.

Definition 4.1. *A* symbolic transition system $\mathcal{S} \doteq \langle X, I, T \rangle$ *is a tuple where:*

- *X is a finite set of (state) variables,*
- *$I(X)$ is a formula denoting the initial states of the system, and*
- *$T(X, X')$ is a formula expressing the transition relation, where X' is the set obtained by replacing each element $x \in X$ with x'.*

Definition 4.2. *A* state s_i *of* $\mathcal{S}$ *is an assignment to the variables* X*.*

Definition 4.3. *A* path *(execution trace)* $\sigma_k \doteq s_0, s_1, s_2, \ldots, s_{k-1}$ *of length* k *(possibly infinite) for* $\mathcal{S}$ *is a sequence of states such that* $s_0 \models I(X)$ *and* $s_i \wedge s_{i+1}\{X \mapsto X'\} \models T(X, X')$ *for all* $0 \leq i < k - 2$*.*

Let $P(X)$ be a formula whose assignments represent a property over the state variables X. ($P(X)$ can be seen as representing the "good" states, while $\neg P(X)$ represents the "bad" states.)

Definition 4.4. *The* invariant verification problem*, denoted with* $\mathcal{S} \models P(X)$*, is the problem of checking if for all the finite paths* $s_0, s_1, \ldots, s_k$ *of* $\mathcal{S}$*, for all* i*,* $0 \leq i \leq k$*,* $s_i \models P(X)$*.*

Its dual formulation in terms of reachability of $\neg P(X)$ is the problem of finding a path $s_0, s_1, \ldots, s_k$ of $\mathcal{S}$ such that $s_k \models \neg P(X)$. In this thesis, we focus on the invariant verification problem.

We denote with $X^{\langle i \rangle}$ the set obtained by replacing x with $x^{\langle i \rangle}$. We call an *unrolling* of $\mathcal{S}$ of length k the formula $I(X^{\langle 0 \rangle}) \wedge \bigwedge_{i=0}^{k-1} T(X^{\langle i \rangle}, X^{\langle i+1 \rangle})$.

4.2 The Verification Modulo Theories Problem

Verification Modulo Theories (VMT) is the problem of verifying the properties of a symbolic transition system where I and T are expressed as SMT($\mathcal{T}$) formulae for some background theory or some combination of theories $\mathcal{T}$. VMT has received a lot of attention over the last decade and various approaches have been proposed, particularly for the invariant verification problem. We discuss the prominent techniques for the invariant verification problem.

Bounded Model Checking

Bounded Model Checking (BMC) [BCCZ99] is a symbolic technique that explores all the paths of the system from the initial state up to a fixed length. It was first introduced for finite-state transition systems, using a SAT solver. Later, it was extended to infinite-state transition systems in [dMRS02], replacing the underlying SAT solver with an SMT solver.

Given a transition system $\mathcal{S} \doteq \langle X, I, T \rangle$ and invariant property $P(X)$, BMC presents to the solver a sequence of proof obligations of the form:

$$BMC(\mathcal{S}, P, k) \doteq I(X^{\langle 0 \rangle}) \wedge T(X^{\langle 0 \rangle}, X^{\langle 1 \rangle}) \wedge \ldots \wedge T(X^{\langle k-1 \rangle}, X^{\langle k \rangle}) \wedge \neg P(X^{\langle k \rangle})$$

for increasing values of k (exploiting incrementality of the solver), until a counterexample trace is found, or resource limit is exhausted. BMC is an incomplete technique, oriented to finding violations, and likely to work well on unsafe instances.

k-Induction

k-induction [SSS00] is a technique that extends BMC for proving properties in transition systems. By exploiting a reasoning similar to the induction principle, the technique consists of finding a bound k such that both a base step and an inductive step hold.

Given $\mathcal{S} \doteq \langle X, I, T \rangle$ and $P(X)$, the base step consists of proving that $P(X)$ holds for the first k steps of the system. This can be done by checking the unsatisfiability of $BMC(\mathcal{S}, P, i)$, for $0 \leq i \leq k$. In addition, the inductive step attempts to prove the validity of an inductive safety argument of the form:

$$\Big(P(X^{\langle 0 \rangle}) \wedge T(X^{\langle 0 \rangle}, X^{\langle 1 \rangle}) \wedge \ldots \wedge P(X^{\langle k-1 \rangle}) \wedge T(X^{\langle k-1 \rangle}, X^{\langle k \rangle})\Big) \ \rightarrow \ P(X^{\langle k \rangle})$$

Efficient algorithms that exploit the solver incrementality to interleave base and inductive steps have been presented in [ES03b].

Similar to BMC, k-induction may be applied to infinite-state transition systems, and it has been explored in several works, e.g., [dMRS03].

A property P is said to be *k-inductive* in a transition system if it can be proved using k-induction for some finite k.

Interpolation-based Model Checking

Interpolation-based Model Checking (IMC) [McM03, McM05] combines BMC and interpolation to allow unbounded model checking (both for finite- and infinite-state transition systems). It tries to overcome the limitations of BMC and k-induction by finding an inductive invariant R.

Definition 4.5. *A formula $R(X)$ is an* inductive invariant *for a transition system $\mathcal{S} \doteq \langle X, I, T \rangle$ and a property $P(X)$ if:*

$$\begin{aligned} I(X) &\models R(X) \\ T(X, X') \wedge R(X) &\models R(X') \\ R(X) &\models P(X) \end{aligned}$$

The fundamental idea of IMC is to use interpolants for over-approximating the set of reachable states in a transition system and also use them to find an inductive invariant R.

IC3

Incremental Construction of Inductive Clauses for Indubitable Correctness (IC3) [Bra11] is an invariant checking procedure for finite-state transition systems. It is considered to be more powerful than IMC – e.g., it is clear from the results of the recent Hardware Model Checking Competition [CLP$^+$16]. A distinguishing characteristic of IC3 is that it works without unrolling of the transition relation.

The goal of IC3 is to find an inductive invariant for the transition system and the property or find a counterexample.

IC3 works on a sequence of sets of formulae $F_0, \ldots, F_k$, called frames, which over-approximate the set of the reachable states up to a fixed length. It maintains the following invariant conditions:

1. $F_0(X) \models I(X)$
2. for all i, $0 \leq i < k, F_i(X) \models F_{i+1}(X)$
3. for all i, $0 \leq i < k, F_i(X) \wedge T(X, X') \models F_{i+1}(X')$
4. for all i, $0 \leq i < k, F_i(X) \models P(X)$

Frames are represented in CNF, and thus each frame is a set of clauses. This allows to obtain an equivalent condition 2: for all i, $0 \leq i < k, F_i \subseteq F_{i+1}$.

On a high-level, IC3 can be seen as an iterative method in which each iteration performs two different phases: the *blocking phase*, and the *propagation phase.* The goal of the *blocking phase* is to ensure that the last frame in the sequence, F_k, satisfies P, i.e., $F_k(X) \models P(X)$. In case $F_k(X) \not\models P(X)$, it recursively tries to check via backward analysis whether some bad state is reachable to F_0. If it is the case, then a counterexample is found (i.e., $\mathcal{S} \not\models P$). Otherwise, the bad state found in some frame F_j $(j \leq k)$ is blocked in F_j. In the *propagation phase*, IC3 checks if a clause in a frame F_i can also be added to the subsequent frame F_{i+1}. To perform this check, IC3 uses the concept of *relative induction* [Bra11].

Definition 4.6. *Given a transition system $\mathcal{S} \doteq \langle X, I, T \rangle$, the formula $\phi(X)$ is inductive relative to $\psi(X)$ in $\mathcal{S}$, if:*

$$\begin{aligned} I(X) &\models \phi(X) \\ T(X, X') \wedge \psi(X) &\models \phi(X') \end{aligned}$$

	$\mathcal{BV}$	$\mathcal{LRA}$	$\mathcal{LIA}$	$\mathcal{NRA}$	$\mathcal{NTA}$
IC3IA [CGMT16]	✓	✓	✓		
CTIGAR [BBW14]			✓		
IC3QE [CG12]		✓	✓		
z3 [dMB08b, HB12a]	✓	✓	✓		
KIND2 [CMST16]			✓		
ABC [BM10, EMB11]	✓				

Table 4.1: IC3 modulo theories – overview

Essentially, IC3 checks if a clause C in F_i is inductive relative to F_i, i.e.,

$$F_i(X) \wedge T(X, X') \models C(X')$$

During the propagation phase, IC3 may discover that $F_{i-1} = F_i$, thus proving that $S \models P$. If it is not the case, IC3 adds a new frame to the sequence and iterates the two phases.

Unlike the other techniques, replacing the underlying SAT solver with an SMT solver does not give an efficient and effective IC3-based procedure for checking invariants of infinite-state transition systems. It is because IC3 requires relatively much more involvement from the underlying solver.

There have been attempts to lift the ideas of IC3 to deal with the transition system expressed in different first-order theories – Table 4.1 shows an overview of the IC3-based approaches to various theories implemented in tools.

4.3 State of the Art

There are not many tools that deal with $\mathcal{NRA}$ and $\mathcal{NTA}$ transition systems. In fact, IC3-based methods are non-existent for the case of $\mathcal{NRA}$ and $\mathcal{NTA}$ (see Table 4.1).

In literature, we have found two classes of approaches: based on nonlinear solving and based on linearization.

Nonlinear Solving

The iSAT3 SMT solver is also a bounded model checker for transition systems. It supports both $\mathcal{NRA}$ and $\mathcal{NTA}$, and it additionally has support for some kinds of differential equations. iSAT3 is built on an SMT solver based on numeric techniques (interval arithmetic), and can provide results that are accurate up to the specified precision. In fact, besides, to "safe" and "unsafe" answers, iSAT3 may return "maybe unsafe" when it finds an envelope of given precision that may (but is not guaranteed to) contain a counterexample. Recently proposed in [MSN$^+$16], iSAT3 has been extended to use an interpolation-based [KB11, MSN$^+$16] approach to prove invariants.

Another relevant tool is dReach [KGCC15], a bounded model checker implemented on top of the dReal [GKC13] SMT solver, that adopts numerical techniques similar to iSAT3. dReach has an expressiveness identical to iSAT3, but since it is a bounded model checker, it is unable to prove properties.

Linearization

The work in [CGKT16] follows a reduction-based approach to check invariants of $\mathcal{NRA}$ transition systems. It over-approximates the non-linear terms with a coarse abstraction, encoding into $\mathcal{LRA}$ some weak properties of multiplication like identity and sign. Another similar approach is presented in [MFK$^+$16] in the context of program analysis. The idea is to find a (tight) convex approximation of polynomials in form of polyhedron, thus obtaining a conservative linear transition system. To the best of our knowledge, there is no available implementation of the approach [MFK$^+$16] in a program analysis tool.

Chapter 5

Systems Design and Verification Tools

Verification is an important step in systems design and development. In fact, more time and effort are spent on verification than construction. Model-based development offers a way to perform early design verification (using some verification backend) while also dealing with the system's complexity. In model-based development, a model is used to describe the possible system behavior. Using models often lead to the discovery of incompleteness, ambiguities, and inconsistencies in the systems specifications. Model-based development has inspired different design languages and tools. Most of them follow the approach of hierarchical decomposition-based design, which allows for reusing of common parts of a system, and thus helping in designing of complex systems.

In this chapter, we discuss the hierarchical decomposition-based design approach, and also overview several design languages and tools that are widely used in industry and academia. Later, we mention some challenges in the integration of verification backends with design tools.

5.1 Hierarchical Decomposition-based Design

The main idea of *hierarchical decomposition-based design* (HBD) is to decompose a system into subsystems and their interaction, at different abstraction layers of abstraction, where a system/subsystem is represented by a component.

Components

A *component* is a design entity which has concise and rigorous interfaces – input and output ports. The *input ports* are controlled by the environment and are fed to the component, whereas the *output ports* are controlled by the component for communicating to the environment. A component can be further composed of other components called *sub-components*, where decomposition of a component into sub-components can be seen as refinement (moving from higher abstraction layer to the lower one). The interaction among components can be *synchronous* (all components perform steps at the same time) or *asynchronous* (each component can take steps autonomously).

Library of Predefined Components. A key characteristic of the hierarchical decomposition-based design is the reuse of components. Often components that are commonly-used are collected in the form of a *component library.* The components in the library are often called *basic components.*

Hierarchical Decomposition

The decomposition of the components into sub-components defines the hierarchy of a design. (This view is often called system architecture.) Hierarchical decomposition preserves ports of the components. A top-level component is called *system component.*

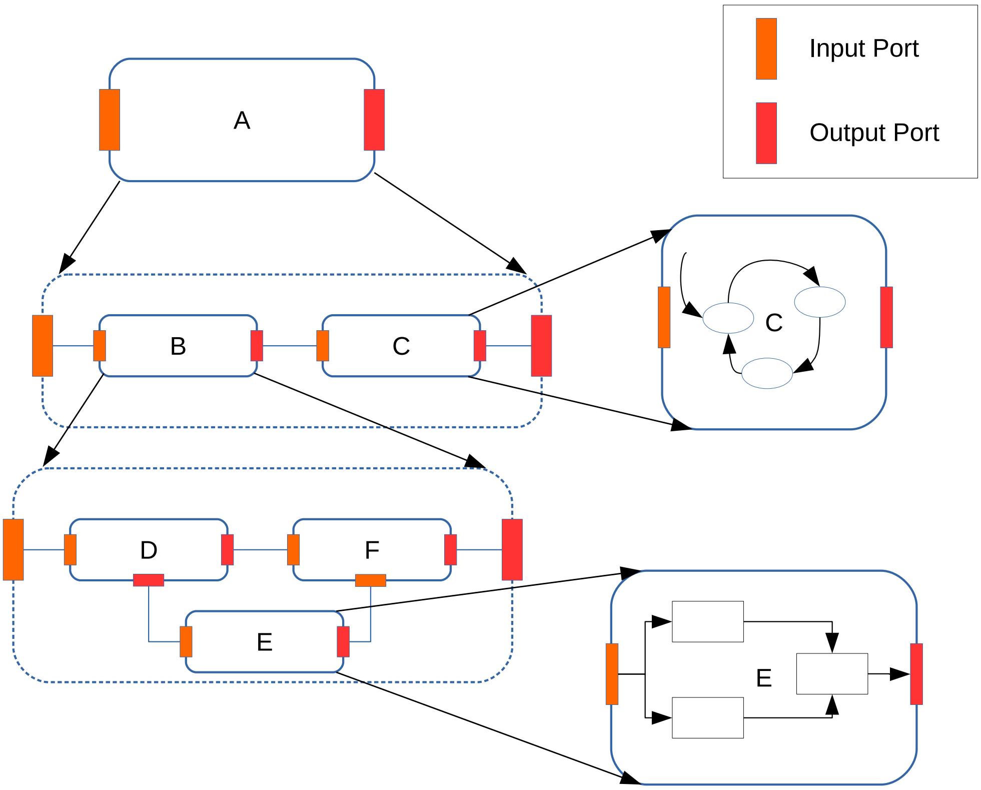


Figure 5.1: An example of hierarchical decomposition

Fig. 5.1 shows an example of hierarchical decomposition: the system component A is decomposed into components B and C, and similarly component B is decomposed into components D, E, and F.

Implementation of Components.

The implementation (behavior) of a component can be given in the form of dataflow or as a state machine. Intuitively, a dataflow is a directed graph where vertices are components and an edge can be seen as a flow from outputs of basic components to inputs of other basic components. It does not have a state. For example, in Fig. 5.1 the implementation of the component E is given in terms of dataflow. On the other hand, an implementation of a component based on a state machine has its own state and internal transitions. For example, the implementation of component C in Fig. 5.1.

Comparison with Contract-based Design

Contract-based design is an emerging paradigm for complex systems design. It is based on mathematical foundations and has been designed keeping formal verification in consideration. In fact, one main goal of contracts is to allow for compositional reasoning which in turns improves the systems verification task. It bears significant similarities with hierarchical decomposition-based design: a contract-based design is also composed of interaction among components. However, a key difference is that each component is associated with a *contract* – a contract is a clear description of the expected behavior of the component. More precisely, a contract is a pair of properties: an *assumption* and a *guarantee* that must be satisfied respectively by the environment and the component.

5.2 Design Languages and Tools

We focus on languages and tools for discrete-time systems design. There are several languages and tools used in industry and academics for systems design. Most of them support the ideas of HBD. These include, but limited to:

Languages: AADL, ALTARICA, LUSTRE, SysML, Verilog,

Tools: AF3, CHESS, COMPASS, Simulink, SCADE, nuXmv & OCRA.

AADL. Architectural Analysis and Design Language (AADL) [FLVC04] focuses on defining system architectures. It is designed for specification, analysis, and automatic code generation of safety-critical systems. It has become an industrial standard for the description of systems. The implementation of a component can be given as dataflow or state machine. It supports modeling of infinite-state systems. Verification of AADL models has been addressed in several works, e.g. [BCK+11].

ALTARICA. The ALTARICA language [alt, APGR99] is a language for the modeling of systems. An ALTARICA model describes a hierarchy of nodes, where nodes can be seen as components in the context of HBD. The implementation of a node is given as dataflow. The language supports for modeling of both synchronous as well asynchronous designs. Although ALTARICA is a textual language, there exist various graphical interfaces – for instance ALTARICA-Studio.

LUSTRE. The LUSTRE language [CPHP87] is a dataflow language for describing synchronous designs. A design in LUSTRE is a hierarchy of nodes (components). It can be used to model infinite-state systems. There are several verification tools for LUSTRE models: Kind2 [CMST16], JKind [Gac16], Zustre [KGC16a] to name a few.

SysML. The graphical Systems Modeling Language (SysML) is a lightweight version of the Unified Modeling Language (UML), (with some extensions) to model discrete-timed systems. Due to the fact that it is very close to UML, it has found a lot of success in the industry. However, its semantics is semi-formal and thus makes the verification task difficult. A SysML design is described as hierarchy of blocks (components) with block parts and flow ports. The behavior (implementation) of a block is given by a state machine or in terms of dataflow.

Verilog. VERILOG [ver06] is a high-level hardware description language, that is widely used in the industry for designing hardware systems. A VERILOG model is described as a hierarchy of modules (components) with clear input and output interface. The properties of VERILOG models are usually written in the SystemVerilog assertions language. There exist many industrial and academic verification tools for VERILOG models.

AF3. AF3 [aut] is an open-source graphical design environment. It supports modeling, simulation, code-generation and verification of designs. Synchronous and infinite-state systems can be modeled in AF3. The implementation of components can be given as dataflow or state machines.

CHESS. The CHESS [che] toolset aims at providing tools and technologies for the design and analysis of complex systems. The tool-set also includes a graphical interface for modeling. It also provides support for design verification. Similar to earlier design environments, a component's implementation can be given as dataflow or state machine in the CHESS framework. Moreover, it also provides support for automatic code generation.

COMPASS. The COMPASS tool-set provides graphical design environment supporting both the hierarchical decomposition- and the contract-based design approaches. The modeling language is called System-Level Integrated Modeling (SLIM) [BCR$^+$09] language, which is an extension of the AADL language and has a formal semantics. The SLIM language can be used to model synchronous and infinite-state designs. The implementation of components can be given as dataflow or state machines. The tool-set also contains formal verification tools.

Simulink. SIMULINK [simb] is a widely-used industrial design environment for systems development and simulation. It is basically a graphical block diagram environment, in which a system can be modeled as hierarchical decomposition-based design – a SIMULINK block can be seen as a component. It comes with a library of blocks that includes: continuous and discrete dynamic blocks, algorithmic blocks, and structural blocks. SIMULINK is extended by specialized block sets (add-on products). For instance, STATEFLOW extends SIMULINK for modeling control logic – develop state machines and flow charts by means of graphical and tabular representation. Both synchronous as well as asynchronous designs can be modeled in SIMULINK. It can also model infinite-state systems. Another powerful feature of SIMULINK is the ability to automatically generate code for certain kinds of designs. A weak point of SIMULINK is that the semantics of SIMULINK is not formally defined, rather they are given by the SIMULINK simulator. Verification of models is usually done by SIMULINK DESIGN VERIFIER.

SCADE. SCADE [sca] is another widely-used graphical design environment supporting the HBD approach. Similar to SIMULINK, the components in SCADE are called blocks. SCADE is further divided into SCADE Architect and SCADE Suite. SCADE Architect is used to describe the architecture of a system and is based on the SYSML language. SCADE Suite is based on the SCADE language which provides capability to model the implementation of the systems components. It also provides functionality for verification and validation, and code generation. It allows for modeling synchronous and also infinite-state systems. Unlike SIMULINK, the SCADE language has a precise formal semantics.

	Contracts	State Machines	DataFlow	Infinite-state Systems	Synchronous	Asynchronous
AADL		✓	✓	✓	✓	✓
ALTARICA			✓		✓	✓
LUSTRE			✓	✓	✓	
SysML		✓	✓		✓	✓
Verilog		✓	✓		✓	
Simulink		✓	✓	✓	✓	✓
SCADE		✓	✓	✓	✓	
COMPASS	✓	✓	✓	✓	✓	
AF3	✓	✓	✓	✓	✓	
CHESS	✓	✓	✓	✓	✓	
nuXmv & OCRA	✓		✓	✓	✓	✓

Table 5.1: Overview of the features provided by the design languages and tools (languages above and tools below)

nuXmv and OCRA. nuXmv [CCD+14] allows for modeling synchronous and infinite-state systems. It also provides several formal verification techniques for the analysis. A model in the nuXmv language can be described as a hierarchy of modules (components in the sense of HBD). OCRA [CDT13] is a tool that provides formal analysis support for contract-based designs. It is built on top of nuXmv. The OCRA language (that is an extension of the nuXmv language) can be used to model infinite-state systems, and also supports synchronous and asynchronous models.

5.3 Verification Support

As mentioned earlier, verification is an important activity in system design development. We have seen that several design languages and tools are used in industry, and some of the tools come with verification support. However, the support depends on what kind of models can be handled by the verification backends. It may be the case that a certain class of models cannot be checked by the verification backend. For instance, the design languages and tools can model infinite-state systems (see Table 5.1) and some of those systems may have nonlinear dynamics. Unfortunately, the verification capabilities of these design tools is limited to the linear case. In this thesis, we propose a technique to prove properties of transition systems with nonlinear dynamics. We also aim at integrating it with a design tool. Nevertheless, integrating more than one verification backend gives more confidence in the verification results.

There are some important points to consider when integrating verification backends with design languages and tools:

- Handling of the hierarchical decomposition of a design and the implementations (dataflow and state machines) of its components.

- The languages of verification tools mostly contain a small set of language constructs. This is not the case for design languages as they need to offer user-friendliness and flexibility in modeling.

- Verification tools are based on precise formal semantics, while some design languages and tools use semi-formal semantics.

- Verification properties can be specified either by annotating or embedding in the form of monitors in designs. An annotated property can be expressed in temporal logic formulae or using some property specification language. The annotation task may not be easy for design engineers. Therefore, representing properties using monitors in the design is often preferred. Intuitively, a monitor can be seen as a particular component that implements the property logic, and its output port indicates the current status of the property.

- The design languages are often different from the ones used by verification backends. A model transformation is performed to connect design tools with verification backends. The model transformation is done by a compiler. Most of the design tools come with a black-box compiler – black-box in the sense that there is no access to the internals of the compiler. On the other hand, a white-box compiler allows access to its internal data structures.
- Due to the fact that the languages for both design and verification tools continuously evolve – new language constructs, changes in syntax and semantics, etc. – the model transformation is a complex task. Moreover, dealing with the hierarchy and wide range of components (in the components library) in designs is also a challenge for model transformation.

Part II

Satisfiability Modulo Nonlinear Arithmetic and Transcendental Functions

Overview

SMT has been a thriving research area since its inception, also leveraging developments in SAT, with applications in formal verification [CMT13, FCN$^+$10, ACKS02], planning [CMR15, CMR16, CMR17], security [AR12, ARTW16, TdHRZ17], and synthesis [RDK$^+$15, BBC16, RKFB17, CMR16].

Powerful and effective SMT techniques and tools are available for the quantifier-free theories of Uninterpreted Functions ($\mathcal{UF}$) and Linear Arithmetic ($\mathcal{LA}$), either over the reals ($\mathcal{LRA}$) or the integers ($\mathcal{LIA}$), as well as their combinations ($\mathcal{UFLRA}$, $\mathcal{UFLIA}$, $\mathcal{UFLIRA}$). A fundamental challenge is to go beyond the linear case, by introducing nonlinear polynomials – over the reals ($\mathcal{NRA}$) or over the integers ($\mathcal{NIA}$) – plus transcendental functions such as exponentiation and trigonometric functions ($\mathcal{NTA}$). In fact, the expressive power of nonlinear arithmetic and transcendental functions is required by many application domains (e.g., railways, aerospace, control software, and cyber-physical systems).

Unfortunately, dealing with nonlinearity is a tough challenge. Going from SMT($\mathcal{LRA}$) to SMT($\mathcal{NRA}$) yields a complexity gap that results in a computational barrier in practice – most available complete solvers rely on Cylindrical Algebraic Decomposition (CAD) techniques [Col74], which require double exponential time in worst case. Adding transcendental functions to $\mathcal{NRA}$ exacerbates the problem even further, because reasoning on $\mathcal{NTA}$ has been shown to be undecidable [Ric68]. Similarly, reasoning in

$\mathcal{NIA}$ is undecidable [Mat93] (the result of Hilbert's 10th problem).

In this part, we take on the challenge of dealing with $\mathcal{NRA}$, $\mathcal{NIA}$, and $\mathcal{NTA}$ in SMT. Our main contributions are:

1. We propose a practical and unifying approach, referred to as *Incremental Linearization*, which trades the use of expensive, exact solvers for nonlinear arithmetic for an abstraction-refinement loop on top of much less expensive solvers for linear arithmetic and uninterpreted functions. It also exploits the incrementality feature provided by the later solvers.

2. Irrational numbers are basically inevitable with the most common transcendental functions, such as exponential and sine, as shown by the Hermite and Niven theorems [Niv61]. We address the challenge to obtain provably correct (rational-coefficient) approximations in $\mathcal{LRA}$.

3. We provide an encoding to deal with the periodicity property of *trigonometric functions*.

4. In order to increase the chances of finding a model for $\mathcal{NRA}$ and $\mathcal{NIA}$, we use heuristic method that relies on a linear arithmetic and uninterpreted functions solver.

5. In the case of $\mathcal{NTA}$, we adopt a logical method to conclude the existence of a solution without explicitly constructing it.

6. We have implemented incremental linearization within the MathSAT SMT solver [CGSS13]. We experimentally evaluate MathSAT on a large set of benchmarks and contrast it against various SMT solvers.

We cover the points 1-5 in Chapter 6 and point 6 in Chapter 7.

Chapter 6

SMT via Incremental Linearization

Incremental linearization is based on an abstraction-refinement loop, using $\mathcal{LA}$ and $\mathcal{UF}$ as abstract domain. The uninterpreted functions are used to model nonlinear and transcendental functions that are iteratively and incrementally axiomatized with a lemma-on-demand approach. Specifically, we eliminate spurious interpretations in the abstract domain, by tightening the piecewise-linear envelope around the (uninterpreted counterpart of the) transcendental functions. The underlying rationale is that, for many practical problems, reasoning with full precision over nonlinear and transcendental functions may not be necessary.

In the abstract domain, nonlinear multiplication between variables is modeled as a binary uninterpreted function. When spurious models are found, the abstraction is tightened by the incremental introduction of linear constraints, including tangent planes resulting from differential calculus, and monotonicity constraints.

In order to deal with transcendental functions, we model them with a unary uninterpreted function, and we rely on several insights. We use Taylor series to exactly compute suitable accurate rational coefficients for the piecewise-linear envelopes. Notice that, nonlinear (Taylor) polynomials are only used to numerically compute the coefficients, i.e., no SMT solving in the theory of nonlinear arithmetic is needed. The refinement is based on the addition, in the abstract domain, of piecewise-linear axiom instantiations, which upper- and lower-bound the candidate solutions, ruling out spurious interpretations. To compute such piecewise-linear bounding functions, the concavity of the curve is taken into account to identify the actual approximation interval. Moreover, to deal with *trigonometric functions*, we leverage the property of periodicity, so that the axiomatization is only done in the interval between $-\pi$ and π, and deal with the external intervals by reduction.

Structure of the Chapter. To describe the approach in more detail, we first present the procedure based on incremental linearization for solving the SMT($\mathcal{NTA}$) problem. In this way, we address $\mathcal{NRA}$ and $\mathcal{NTA}$ together since $\mathcal{NRA}$ is a strict subset of $\mathcal{NTA}$. §6.1 provides a high-level description of the procedure for SMT($\mathcal{NTA}$). In §6.2 we discuss the refinement, while in §6.3 we focus on how to check for spuriousness and detect satisfiability. In §6.4 we present the correctness of the approach. Then, in §6.5 we move the attention towards handling the SMT($\mathcal{NIA}$) problem with incremental linearization. Interestingly, going from $\mathcal{NRA}$ (a subset of $\mathcal{NTA}$) to $\mathcal{NIA}$ requires little modification in the procedure we present. In §6.6 we discuss related work.

6.1 Incremental Linearization for SMT($\mathcal{NTA}$)

We now provide a high-level description of the algorithm for SMT solving on $\mathcal{NTA}$ (and hence $\mathcal{NRA}$) based on incremental linearization. Further details will be provided in the next sections.

To simplify the presentation, and when not explicitly stated otherwise, we often implicitly assume w.l.o.g. that all multiplications in the input formula φ are either between variables (e.g., $x * y$) or between one constant and one variable (e.g., $3x$), and that all transcendental functions in φ are applied to variables (e.g., exp(x)). This can be obtained by recursively substituting each non-variable term t inside multiplications and transcendental function applications with a fresh variable x_t, and by conjoining $(x_t = t)$ to φ.

6.1.1 The Main Procedure

The main algorithm is shown in Fig. 6.1. The main function SMT-NTA-CHECK takes as input a formula φ containing non-linear constraints with polynomials and transcendental functions, and returns a Boolean value asserting if φ is satisfiable or not. When the formula is found to be unsatisfiable it returns also the set of $\mathcal{UFLRA}$ constraints Γ such that the formula $\varphi \wedge \bigwedge \Gamma$ can be shown unsatisfiable using an $\mathcal{UFLRA}$ SMT solver. Notice that, SMT-NTA-CHECK is not guaranteed to terminate, so that we implicitly assume that it is stopped as soon as some given resource budget (e.g., time, memory, number of iterations) is exhausted.

In order to deal with transcendental models, we adopt a mechanism based on rational approximations of irrational values; a variable ϵ keeps track of the current precision of approximation, and it is incremented on demand.

First, in line 1 the formula undergoes some $\mathcal{NTA}$-satisfiability-preserving preprocessing step, which produces the formula $\varphi' \doteq \varphi \wedge \varphi_{shift}$ by introducing some fresh real variables ω_x and by conjoining to φ a formula φ_{shift} which defines univocally the values of the ω_x's in terms of some variables x's in φ (see §6.2).

Then the formula φ' is abstracted into an over-approximating formula $\widehat{\varphi}$ over the combined theory of linear arithmetic and uninterpreted functions ($\mathcal{UFLRA}$) by invoking SMT-INITIAL-ABSTRACTION (line 2). $\widehat{\varphi}$ is

```
⟨bool, constraint-set⟩ SMT-NTA-CHECK ($\varphi$):
1.  $\varphi' :=$ SMT-PREPROCESS($\varphi$)
2.  $\widehat{\varphi} :=$ SMT-INITIAL-ABSTRACTION($\varphi'$)
3.  $\Gamma := \emptyset$
4.  $\epsilon :=$ INITIAL-PRECISION ()
5.  while true:
6.      $\langle sat, \widehat{\mu}\rangle :=$ SMT-UFLRA-CHECK ($\widehat{\varphi} \wedge \bigwedge \Gamma$)
7.      if not $sat$:
8.          return ⟨false, $\Gamma$⟩
9.      $\langle sat, \Gamma'\rangle :=$ CHECK-REFINE ($\varphi'$, $\widehat{\varphi}$, $\widehat{\mu}$, $\epsilon$)
10.     if $sat$:
11.         return ⟨true, $\emptyset$⟩
12.     $\Gamma := \Gamma \cup \Gamma'$
```

Figure 6.1: The main procedure for solving SMT($\mathcal{NTA}$) via abstraction to SMT($\mathcal{UFLRA}$) and refinement

the result of replacing each nonlinear term $x * y$ with $f_*(x, y)$, and each transcendental term TF(x) with $f_{\text{TF}}(x)$, s.t. $f_*(.)$ and $f_{\text{TF}}(.,.)$ are uninterpreted functions, and the symbol π with the new symbol $\widehat{\pi}$ (see §6.2). (We remark that, linear multiplications, like e.g., $c * x$ where c is a constant, are not replaced.) The set of constraints Γ is initialized to the empty set, and the precision variable ϵ is initialized to some (small) real value by calling the function INITIAL-PRECISION (lines 3-4).

Then the algorithm enters a loop (lines 5-12). At each iteration, the approximation $\widehat{\varphi}$ of φ' is refined by adding new $\mathcal{UFLRA}$ constraints to Γ that rule out spurious solutions. The loop maintains the invariant that $\widehat{\varphi} \wedge \bigwedge \Gamma$ is an over-approximation of φ, (see Lemma 6.7). The process iterates until either the formula $\widehat{\varphi} \wedge \bigwedge \Gamma$ is proved unsatisfiable in SMT($\mathcal{UFLRA}$) by invoking the standard SMT-solving function SMT-UFLRA-CHECK (lines 6-8), or $\widehat{\varphi} \wedge \bigwedge \Gamma$ is proved $\mathcal{UFLRA}$-satisfiable and the satisfiability result can be lifted to a satisfiability result for the original formula φ by means of a refinement process (lines 9-11). The function CHECK-REFINE (see §6.1.2) takes as input the formula φ', its abstracted version $\widehat{\varphi}$, the current abstract model $\widehat{\mu}$ and the precision ϵ, and it returns $\langle \mathbf{true}, \emptyset \rangle$ if it achieves proving the $\mathcal{NTA}$-satisfiability of φ', it returns $\langle \mathbf{false}, \Gamma' \rangle$ if it fails, Γ' being a non-empty set of $\mathcal{UFLRA}$ constraints that rule out $\widehat{\mu}$ (and other spurious solutions). When none of the above loop-exit condition occurs, the novel constraints in Γ' which were found by CHECK-REFINE are added to Γ before entering into next loop.

Unsatisfiable-core Extraction. As a byproduct of the procedure in Fig. 6.1, since state-of-the-art SMT solvers for $\mathcal{UFLRA}$ can return unsatisfiable cores when the input formula is found unsatisfiable, we can easily modify SMT-NTA-CHECK to produce also an unsatisfiable core for $\mathcal{NTA}$ when φ is $\mathcal{NTA}$-unsatisfiable. This is done by a variation of the lemma-lifting technique in [CGS11]: when SMT-UFLRA-CHECK returns **false**, then it can be asked to produce an unsatisfiable core $\widehat{\psi}$ of $\widehat{\varphi} \wedge \Gamma$. Then we drop from $\widehat{\psi}$ all the conjuncts which belong either to Γ or to the abstraction of φ_{shift}, and produce the final $\mathcal{NTA}$ unsatisfiable core by un-abstracting the result, rewriting back each f_*, f_{TF} and $\widehat{\pi}$ into $*$, TF and π respectively. The conjuncts in $\bigwedge \Gamma$ and in the abstraction of φ_{shift} are safely ignored because they would respectively produce $\mathcal{NTA}$-valid subformulae and simple definitions of variables which do not occur in φ. (See §6.2.)

6.1.2 Abstraction Refinement and Spuriousness Check

The process of checking for spuriousness and refining the abstraction is carried out by the procedure CHECK-REFINE (reported in Fig. 6.2). CHECK-REFINE first calls the function CHECK-MODEL on the original formula φ, the abstract model $\widehat{\mu}$ and the value ϵ, which tries to determine whether $\widehat{\mu}$ does indeed imply the existence of a model for φ (lines 1-2). (CHECK-MODEL

```
$\langle$bool, constraint-set$\rangle$ CHECK-REFINE ($\varphi$, $\widehat{\varphi}$, $\widehat{\mu}$, $\epsilon$):
1.  if CHECK-MODEL ($\varphi$, $\widehat{\mu}$, $\epsilon$):
2.      return $\langle$true, $\emptyset\rangle$
3.  $\Gamma$ := BLOCK-SPURIOUS-PRODUCT-TERMS($\widehat{\varphi}$, $\widehat{\mu}$)    # refinement of products
4.  while true:   # refinement of transcendental functions,
      # for progressively improving precision
5.      $\Gamma := \Gamma \cup$ BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS($\widehat{\varphi}$, $\widehat{\mu}$, $\epsilon$)
6.      if $\Gamma \neq \emptyset$:
7.          return $\langle$false, $\Gamma\rangle$
8.      $\epsilon$ := IMPROVE-PRECISION ($\epsilon$)
9.      if CHECK-MODEL ($\varphi$, $\widehat{\mu}$, $\epsilon$):
10.         return $\langle$true, $\emptyset\rangle$
```

Figure 6.2: The main procedure for spuriousness check and refinement

is described in §6.3). The check is formulated as a SMT($\mathcal{UFLRA}$) search problem over a constrained version of φ, guided by the current abstract model and the current precision, either yielding a sufficient criterion for concluding the existence of a model for φ, returning **true**, or stating that $\widehat{\mu}$ is spurious, returning **false**. If CHECK-MODEL succeeds, then CHECK-REFINE returns $\langle$**true**, $\emptyset\rangle$, and the whole process terminates. Otherwise $\widehat{\mu}$ is spurious –because it violates some multiplications, or some transcendental functions, or both– and CHECK-MODEL could not prove the existence of another model within the current precision. (Nevertheless, one such model could exist, and might be found using a better precision.)

The rest of the procedure tries to refine the spurious model $\widehat{\mu}$ by adding $\mathcal{UFLRA}$ refinement constraints that rule out $\widehat{\mu}$ (and other spurious solutions), which are collected into the set Γ, interleaving this process with calls to CHECK-MODEL with increasingly improved precision. This is performed in two steps.

The first step is the *refinement of products* (line 3). BLOCK-SPURIOUS-PRODUCT-TERMS is invoked on $\widehat{\varphi}$ and $\widehat{\mu}$ and looks for $\mathcal{UFLRA}$ constraints on multiplication terms in the form $f_*(x, y)$ occurring in $\widehat{\varphi}$ which are violated by $\widehat{\mu}$. These constraints are stored in Γ. Importantly, this process does not depend on the precision ϵ. (BLOCK-SPURIOUS-PRODUCT-TERMS is described in §6.2.)

The second step is the *refinement of transcendental functions*, which is performed for progressively-improving precision (lines 4-10). At each iteration, first BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS is invoked on $\widehat{\varphi}$, $\widehat{\mu}$ and ϵ, and looks for $\mathcal{UFLRA}$ constraints on transcendental terms in the form $f_{\text{TF}}(x)$ occurring in $\widehat{\varphi}$ which are violated by $\widehat{\mu}$. These constraints (if any) are added to Γ. (BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS is described in §6.2.) Then, if Γ contains at least one refinement constraint ruling out $\widehat{\mu}$, then $\langle \textbf{false}, \Gamma \rangle$ is returned. If not so, then no result in either direction was obtained with the current precision. Then, the current precision is increased (in the current implementation, we simply reduce ϵ by one order of magnitude), and CHECK-MODEL is invoked again with the improved precision, returning $\langle \textbf{true}, \emptyset \rangle$ if it succeeds, like in lines 1-2. The whole process in lines 5-10 is iterated until either φ is found satisfiable, or some refinement constraint is produced (or the process is terminated due to resource-budget exhaustion).

Remark 1. It is important to notice that Fig. 6.2 describes the strategy which is currently implemented, which is only one of the many alternative strategies by which refinement can be performed. E.g., the calls to CHECK-MODEL, BLOCK-SPURIOUS-PRODUCT-TERMS and BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS are not bound to be executed necessarily in this sequence, and can be interleaved in different ways. For instance, one could adopt the strategy to call BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS only if $\Gamma = \emptyset$, so that CHECK-REFINE will be repeatedly called to refine only multiplications (line 3) until a fixpoint is reached, and to refine the transcendental functions only after then. Further information on these issues will be provided in §6.2 and §7.1.

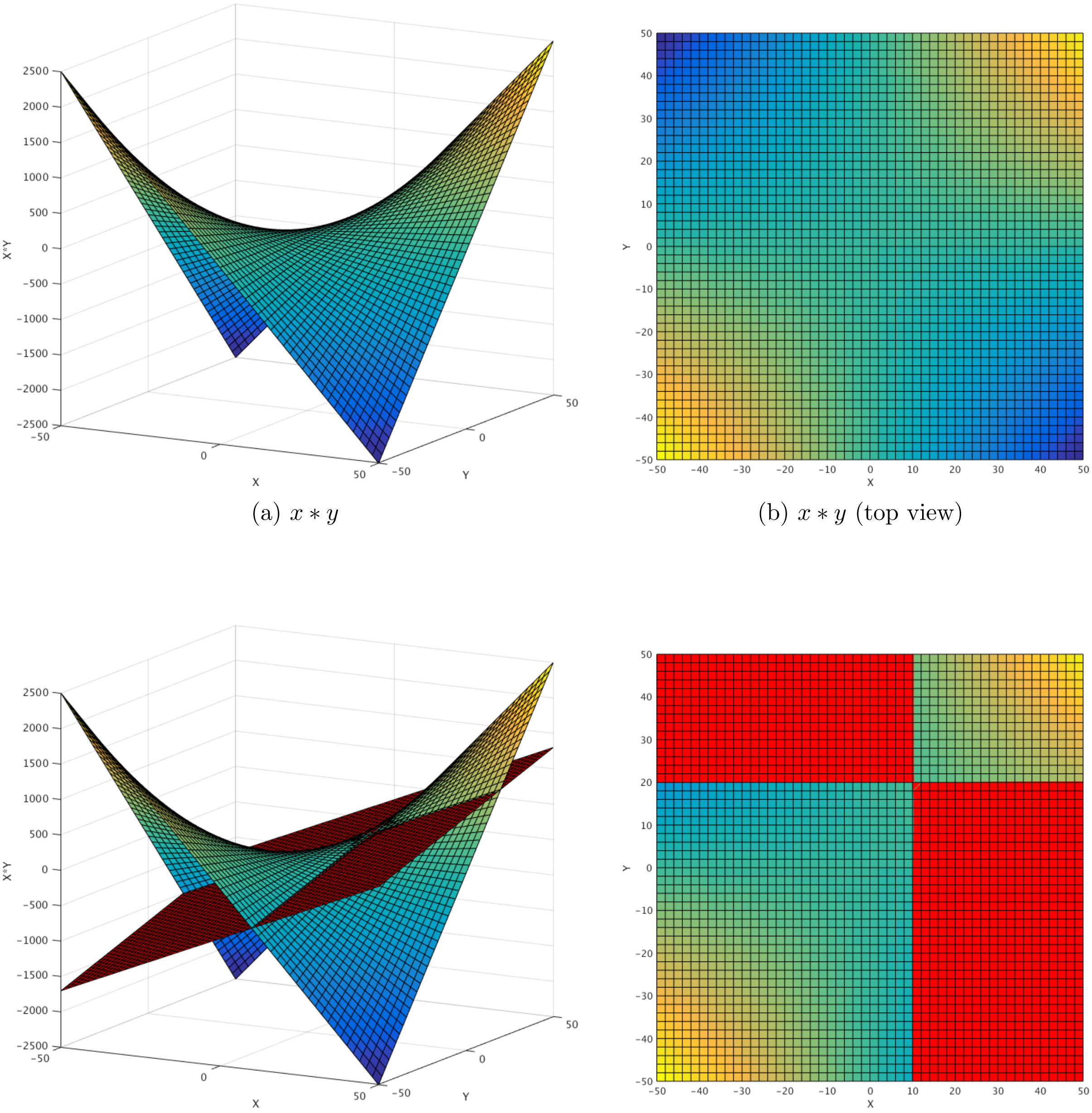
6.2 Abstraction Refinement

In this section we focus on the refinement part of our procedure. We first describe how to perform the refinement of multiplication terms $f_*(x, y)$ in the function BLOCK-SPURIOUS-PRODUCT-TERMS (§6.2.1) and then how to perform refinement of transcendental functions in the function BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS (§6.2.2).

6.2.1 Refinement for $\mathcal{NRA}$

We describe the refinement of multiplication terms in the function BLOCK-SPURIOUS-PRODUCT-TERMS. The refinement is based on selecting suitable instantiations of given constraint schemata which prevent spurious assignments to multiplication terms. We consider the refinement constraint schemata in Fig. 6.4, where x, x_i, y, y_i are variables and a, b are generic rational values. It is straightforward to verify that all the constraints are valid formulae in any theory interpreting $f_*()$ as $*$. Notice that, the Zero constraints refer to a single multiplication term, that the Sign, Commutativity and Monotonicity constraints refer to pairs of multiplication terms, whereas the Tangent plane constraints refer to a single multiplication term and a single point (a, b). The Zero, Sign, Commutativity and Monotonicity constraints are self-explanatory; the Tangent-plane constraints, instead, deserve some explanations.

The equalities in the Tangent-plane constraints are providing *multiplication lines* which enforce the correct value of $f_*(x, y)$ when $x = a$ or $y = b$; the inequalities are providing bounds for $f_*(x, y)$ when x and y are not on the multiplication lines. The constraints specialize the notion of tangent plane to the case of nonlinear multiplication. Geometrically, the surface



(a) $x * y$

(b) $x * y$ (top view)

(c) $x * y$ and tangent plane

(d) $x * y$ and tangent plane (top view)

Figure 6.3: Multiplication function and tangent plane

$$
\begin{array}{rl}
\textit{Zero:} & \forall x, y.((x = 0 \lor y = 0) \leftrightarrow f_*(x, y) = 0) \\
& \forall x, y.(((x > 0 \land y > 0) \lor (x < 0 \land y < 0)) \leftrightarrow f_*(x, y) > 0) \\
& \forall x, y.(((x < 0 \land y > 0) \lor (x > 0 \land y < 0)) \leftrightarrow f_*(x, y) < 0) \\
\textit{Sign:} & \forall x, y.f_*(x, y) = f_*(-x, -y) \\
& \forall x, y.f_*(x, y) = -f_*(-x, y) \\
& \forall x, y.f_*(x, y) = -f_*(x, -y) \\
\textit{Commutativity:} & \forall x, y.f_*(x, y) = f_*(y, x) \\
\textit{Monotonicity:} & \forall x_1, y_1, x_2, y_2.((abs(x_1) \leq abs(x_2) \land abs(y_1) \leq abs(y_2)) \rightarrow abs(f_*(x_1, y_1)) \leq abs(f_*(x_2, y_2))) \\
& \forall x_1, y_1, x_2, y_2.((abs(x_1) < abs(x_2) \land abs(y_1) \leq abs(y_2) \land y_2 \neq 0) \rightarrow abs(f_*(x_1, y_1)) < abs(f_*(x_2, y_2))) \\
& \forall x_1, y_1, x_2, y_2.((abs(x_1) \leq abs(x_2) \land abs(y_1) < abs(y_2) \land x_2 \neq 0) \rightarrow abs(f_*(x_1, y_1)) < abs(f_*(x_2, y_2))) \\
\textit{Tangent plane:} & \forall x, y.(f_*(a, y) = a * y \land f_*(x, b) = b * x \land \\
& \quad (((x > a \land y < b) \lor (x < a \land y > b)) \rightarrow f_*(x, y) < \textsc{TanPlane}_{*,a,b}(x, y)) \land \\
& \quad (((x < a \land y < b) \lor (x > a \land y > b)) \rightarrow f_*(x, y) > \textsc{TanPlane}_{*,a,b}(x, y)))
\end{array}
$$

Figure 6.4: The refinement $\mathcal{UFLRA}$ constraint schemata for multiplication (We recall that "$abs(x)$" is a shortcut for "$\mathsf{ite}(x < 0, -x, x)$" and that "$\textsc{TanPlane}_{*,a,b}(x, y)$" is a shortcut for "$b * x + a * y - a * b$".)

generated by the multiplication function $mul(x, y) \doteq x * y$ is shown in Fig. 6.3a and 6.3b. This kind of surface is known in geometry as hyperbolic paraboloid. A hyperbolic paraboloid is a doubly-ruled surface, i.e., for every point on the surface, there are two distinct lines projected from the surface such that they pass through the point. A tangent plane to a hyperbolic paraboloid has the property that the two projected lines from the surface are also in the tangent plane, and they define how the plane cuts the surface. In case of the multiplication surface, the projected lines basically lie on the surface (see Fig. 6.3c and 6.3d).

Let $\widehat{\mu}$ be the spurious interpretation which is given as input to BLOCK-SPURIOUS-PRODUCT-TERMS. Let $f_*(x, y)$ be one generic multiplication term occurring in $\widehat{\varphi}$, let $a \doteq \widehat{\mu}[x]$ and $b \doteq \widehat{\mu}[y]$. If $\widehat{\mu}[f_*(x, y)] \neq a * b$, then $f_*(x, y)$ is a *spurious term* in $\widehat{\mu}$. The idea is thus to add refinement constraints to block the spurious terms $f_*(x, y)$ in $\widehat{\varphi}$ so that a new interpretation will be constructed. In what follows, $\mathcal{ST}_*^{\widehat{\mu}}$ denotes the set of multiplication terms in $\widehat{\varphi}$ which are made spurious by $\widehat{\mu}$.

Single-term Refinements. If $f_*(t_1, s_1)$ is the only multiplication term occurring in a constraint schema $\forall x, y.\psi$ in Fig. 6.4 (i.e., a Zero constraint schema), and $f_*(t_1, s_1) \in \mathcal{ST}_*^{\widehat{\mu}}$, then we say that $\widehat{\mu}$ *violates* $\forall x, y.\psi$ *on* $f_*(t_1, s_1)$ *wrt.* $f_*(x, y)$ if and only if $\psi\{x, y \mapsto \widehat{\mu}[t_1], \widehat{\mu}[s_1]\}$ is false in any theory interpreting $f_*()$ as $*$. Then, for every term $f_*(t_1, s_1) \in \mathcal{ST}_*^{\widehat{\mu}}$, and for every constraint schema $\forall x, y.\psi$ as above, if $\widehat{\mu}$ violates $\forall x, y.\psi$ on $f_*(t_1, s_1)$ wrt. $f_*(x, y)$, then we can produce the refinement constraint $\psi\{x, y \mapsto t_1, s_1\}$. By construction, $\widehat{\mu} \not\models \psi'$, that is, ψ' rules out $\widehat{\mu}$.

Double-term Refinements. Similarly, if $f_*(t, s)$ and $f_*(u, w)$ for some terms s, t, u, w are the only multiplication terms occurring in a constraint schema $\forall \mathbf{x}.\psi$ (i.e., a Sign, Commutativity, or Monotonicity constraint schema), and $f_*(t_1, s_1), f_*(u_1, w_1) \in \mathcal{ST}_*^{\widehat{\mu}}$ s.t. $\langle f_*(t, s), f_*(u, w)\rangle$ can be mapped into $\langle f_*(t_1, s_1), f_*(u_1, w_1)\rangle$ by some variable instantiation $\sigma : \mathbf{x} \mapsto \mathbf{t}$, we say that $\widehat{\mu}$ *violates* ψ *on* $\langle f_*(t_1, s_1), f_*(u_1, w_1)\rangle$ *wrt* $\langle f_*(t, s), f_*(u, w)\rangle$ if and only if $\psi\{\mathbf{x} \mapsto \widehat{\mu}[\mathbf{t}]\}$ is false in any theory interpreting $f_*()$ as $*$. Then, for every pair of terms $\langle f_*(t_1, s_1), f_*(u_1, w_1)\rangle$ and for every constraint schema ψ as above, if $\widehat{\mu}$ violates ψ on $f_*(t_1, s_1)$ wrt $\langle f_*(t, s), f_*(u, w)\rangle$ as above, then we can produce the refinement constraint $\psi' \doteq \psi\{\mathbf{x} \mapsto \mathbf{t}\}$. By construction, $\widehat{\mu} \not\models \psi'$, that is, ψ' rules out $\widehat{\mu}$.

Tangent-Plane Refinements. For every term $f_*(t_1, s_1) \in \mathcal{ST}_*^{\widehat{\mu}}$, and for each Tangent-plane constraint schema $\forall x, y.\psi$, we can produce the refinement constraint $\psi' \doteq \psi\{x, y, a, b \mapsto t_1, s_1, \widehat{\mu}[t_1], \widehat{\mu}[s_1]\}$. By construction, $\widehat{\mu} \not\models \psi'$, that is, ψ' rules out $\widehat{\mu}$.

Example 6.1. Consider the case where φ contains the multiplications $u_1 * w_1$ and $u_2 * w_2$, so that $\widehat{\varphi}$ contains the multiplication terms $f_*(u_1, w_1)$ and $f_*(u_2, w_2)$. Let $\widehat{\mu}$ be a spurious assignment s.t.

$$\widehat{\mu}[u_1] = 2,\ \widehat{\mu}[w_1] = 3,\ \widehat{\mu}[f_*(u_1, w_1)] = 7,\ \widehat{\mu}[u_2] = 3,\ \widehat{\mu}[w_2] = -4,$$
$$\widehat{\mu}[f_*(u_2, w_2)] = 5.$$

$\widehat{\mu}$ violates the third Zero constraint on $f_*(u_2, w_2)$, it does not violate any Sign or Commutativity constraint, and it violates the three Monotonicity constraints on $\langle f_*(u_1, w_1), f_*(u_2, w_2)\rangle$. Overall, this leads to the addition of the Zero and Monotonicity constraints, plus the Tangent-plane ones in the points $(2, 3)$ and $(3, -4)$, which are reported at the top of Fig. 6.5.

If φ contains also $f_*(u_1, -w_1)$ and $f_*(w_2, u_2)$, and if $\widehat{\mu}[f_*(u_1, -w_1)] = -3$ and $\widehat{\mu}[f_*(w_2, u_2)] = 9$, then we add also the Sign and Commutativity constraints in the bottom part of Fig. 6.5 (plus the other Zero, Monotonicity and Tangent-plane constraints). △

$$
\begin{array}{rl}
\hline
\textit{Zero:} & ((u_2 < 0 \wedge w_2 > 0) \vee (u_2 > 0 \wedge w_2 < 0)) \leftrightarrow f_*(u_2, w_2) < 0 \\
\textit{Monotonicity:} & (abs(u_1) \leq abs(u_2) \wedge abs(w_1) \leq abs(w_2)) \rightarrow abs(f_*(u_1, w_1)) \leq abs(f_*(u_2, w_2)) \\
 & (abs(u_1) < abs(u_2) \wedge abs(w_1) \leq abs(w_2) \wedge w_2 \neq 0) \rightarrow abs(f_*(u_1, w_1)) < abs(f_*(u_2, w_2)) \\
 & (abs(u_1) \leq abs(u_2) \wedge abs(w_1) < abs(w_2) \wedge u_2 \neq 0) \rightarrow abs(f_*(u_1, w_1)) < abs(f_*(u_2, w_2)) \\
\textit{Tangent plane:} & f_*(2, w_1) = 2 * w_1 \\
 & f_*(u_1, 3) = 3 * u_1 \\
 & ((u_1 > 2 \wedge w_1 < 3) \vee (u_1 < 2 \wedge w_1 > 3)) \rightarrow f_*(u_1, w_1) < 3 * u_1 + 2 * w_1 - 6 \\
 & ((u_1 < 2 \wedge w_1 < 3) \vee (u_1 > 2 \wedge w_1 > 3)) \rightarrow f_*(u_1, w_1) > 3 * u_1 + 2 * w_1 - 6 \\
 & f_*(3, w_2) = 3 * w_2 \\
 & f_*(u_2, -4) = -4 * u_2 \\
 & ((u_2 > 3 \wedge w_2 < -4) \vee (u_2 < 3 \wedge w_2 > -4)) \rightarrow f_*(u_2, w_2) < -4 * u_2 + 3 * w_2 + 12 \\
 & ((u_2 < 3 \wedge w_2 < -4) \vee (u_2 > 3 \wedge w_2 > -4)) \rightarrow f_*(u_2, w_2) > -4 * u_2 + 3 * w_2 + 12 \\
\hline
\textit{Sign:} & f_*(u_1, w_1) = -f_*(u_1, -w_1) \\
\textit{Commutativity:} & f_*(u_2, w_2) = f_*(w_2, u_2) \\
\hline
\end{array}
$$

Figure 6.5: Top: example of instantiation of constraint schemata for the multiplication terms $f_*(u_1, w_1)$ and $f_*(u_2, w_2)$, where $\widehat{\mu}[u_1] = 2,\ \widehat{\mu}[w_1] = 3,\ \widehat{\mu}[f_*(u_1, w_1)] = 7,\ \widehat{\mu}[u_2] = 3,\ \widehat{\mu}[w_2] = -4,\ \widehat{\mu}[f_*(u_2, w_2)] = 5$.
Bottom: example of instantiation of Sign and Commutativity if we consider also $f_*(u_1, -w_1)$ and $f_*(w_2, u_2)$

Remark 2. The above narration describes a very "eager" refinement strategy, in which all possible refinement constraints for all possible spurious multiplication terms are generated. Notice, however, that in order to rule out $\widehat{\mu}$ it is sufficient to produce one single refinement constraint for one single spurious multiplication term. Thus, a great variety of more "lazy" strategies can be adopted, in which only some of the schemata are instantiated, and only for some of the multiplication terms. For example, rather than refining all spurious terms, it might be a good idea to refine only terms occurring in atomic subformulae whose truth-value assignment in $\widehat{\mu}$ actually contributed to satisfy $\widehat{\varphi}$. (E.g., atoms occurring only positively in $\widehat{\varphi}$ which are true in $\widehat{\mu}$, see e.g., [BSST09].)

Also, the instantiation of each constraint schema can be implemented at different levels of granularity, because some constraint schema may correspond to the conjunction of more than one clause (e.g., three clauses in the case of each Zero constraint schema [1]) so that one may decide to instantiate all, some or only one of the clauses that are actually violated by $\widehat{\mu}$.

Notice that, even more eager refinement strategies are possible, e.g., by instantiating the Sign and Commutativity constraint schemata $\forall x, y.\psi$ also when only one multiplication term $f_*(t_1, s_1)$ in $\mathcal{ST}_*^{\widehat{\mu}}$ matches one of the two multiplication terms in $\forall x, y.\psi$, adding a new multiplication term.

We will discuss the strategies which we have actually chosen and implemented in §7.1.

6.2.2 Refinement for $\mathcal{NTA}$

We consider now the problem of eliminating spurious assignments to transcendental functions. The pseudo-code for BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS is shown in Fig. 6.6. We iterate on all the

[1] A formula in the form $(A \vee B) \leftrightarrow C$ can be rewritten as $(A \vee B \vee \neg C) \wedge (\neg A \vee C) \wedge (\neg B \vee C)$.

```
constraint-set BLOCK-SPURIOUS-TRANSCENDENTAL-TERMS ($\widehat{\varphi}$, $\widehat{\mu}$, $\epsilon$):
$\Gamma := \emptyset$
for all $f_{\mathrm{TF}}(x) \in \widehat{\varphi}$:
    $\langle P_l(x), P_u(x) \rangle$ := GET-POLYNOMIAL-BOUNDS ($f_{\mathrm{TF}}(x)$, $\widehat{\mu}$, $\epsilon$)
    if $\widehat{\mu}[f_{\mathrm{TF}}(x)] \notin [P_l(\widehat{\mu}[x]), P_u(\widehat{\mu}[x])]$:
        $\Gamma := \Gamma \cup$ BLOCK-SPURIOUS-NTA-TERM($f$TF$(x), \widehat{\mu}, P_l(x), P_u(x))$
return $\Gamma$
```

Figure 6.6: Refinement of transcendental functions

(abstract) transcendental function applications $f_{\mathrm{TF}}(x)$ in $\widehat{\varphi}$, in order to check whether the SMT($\mathcal{UFLRA}$)-model $\widehat{\mu}$ is consistent with the $\mathcal{NTA}$ semantics. In principle, this amounts to checking if $\widehat{\mu}[f_{\mathrm{TF}}(x)] = \textsc{tf}(\widehat{\mu}[x])$. In practice, the check cannot be implemented, since transcendental functions at rational points most often have irrational values (see e.g., [Niv61]), which cannot be represented in SMT($\mathcal{UFLRA}$).

Therefore, for each term $\textsc{tf}(x)$ in φ, we instead compute two rational values, namely QL and QU, with the property that $\textsc{ql} \leq \textsc{tf}(\widehat{\mu}[x]) \leq \textsc{qu}$. The computation of QL and QU is based on polynomials computed using Taylor series, according to the given current precision, by the function GET-POLYNOMIAL-BOUNDS of Fig. 6.8. This is done by expanding the Maclaurin series of TF, generating polynomials for the upper and lower bounds according to Taylor's theorem (see Chapter 2), until the requested precision ϵ is met. The two values QL and QU are simply the results of evaluating the two computed polynomials $P_l(x)$ and $P_u(x)$ at $\widehat{\mu}[x]$. As an additional requirement that will be explained below, the function also ensures (lines 7–9) that the concavity of the Taylor polynomials is the same as that of TF at $\widehat{\mu}[x]$.

If the value of $\textsc{tf}(x)$ in $\widehat{\mu}$ is not included in the interval $[\textsc{ql}, \textsc{qu}]$, the function BLOCK-SPURIOUS-NTA-TERM of Fig. 6.9 is used to generate (piecewise) linear constraints that remove the point $(\widehat{\mu}[x], \widehat{\mu}[\textsc{tf}(x)])$ (and possibly many others) from the graph of f_{TF}, thus refining the abstraction.

The refinement is performed by BLOCK-SPURIOUS-NTA-TERM of Fig. 6.9 in two steps. First, it attempts to exclude the bad point by invoking BLOCK-SPURIOUS-NTA-BASIC, which instantiates some *basic constraint schemata* describing very general properties of the transcendental function TF under consideration (lines 1-3). These constraints encode some simple properties of transcendental functions (such as sign and monotonicity conditions, or bounds at noteworthy values) via linear relations, and are described in §6.2.2. If the current abstract model $\widehat{\mu}$ violates any of the basic constraints, BLOCK-SPURIOUS-NTA-TERM simply returns their corresponding instantiations.

If none of the basic constraints is violated, then two situations are possible, as illustrated in Fig. 6.7. Let the green line be the graph of some transcendental function TF. The points $p_1 = (x_1, y_1)$ and $p_2 = (x_2, y_2)$, say (1.0,1.0) and (2.2, 0.5), represent transcendental terms that are spurious in the current assignment $\widehat{\mu}$ —that is, $p_1 = (\widehat{\mu}[x_1], \widehat{\mu}[f_{\mathrm{TF}}(x_1)])$ and $p_2 = (\widehat{\mu}[x_2], \widehat{\mu}[f_{\mathrm{TF}}(x_2)])$, for some x_1, x_2. In order to eliminate them, we need to discover linear constraints that are guaranteed to safely approximate TF. Clearly, a major role is played by the position of the spurious value $\widehat{\mu}[f_{\mathrm{TF}}(x)]$ relative to the correct value $\mathrm{TF}(\widehat{\mu}[x])$, and by the concavity of TF around the point $\widehat{\mu}[x]$. If the concavity is negative or equal to zero, and the point lies above the function, then the tangent to the function would be adequate to block the spurious assignment –and tighten the approximation of TF. (This is the case for p_1.) However, if the concavity is negative but the point lies below the function, then a tangent would be not adequate. (This is the case for p_2.) For this reason, secants are required, which are unfortunately not unique.

The main problem, however, is that the coefficients of the tangent or of a secant to a transcendental function for a rational value are likely to

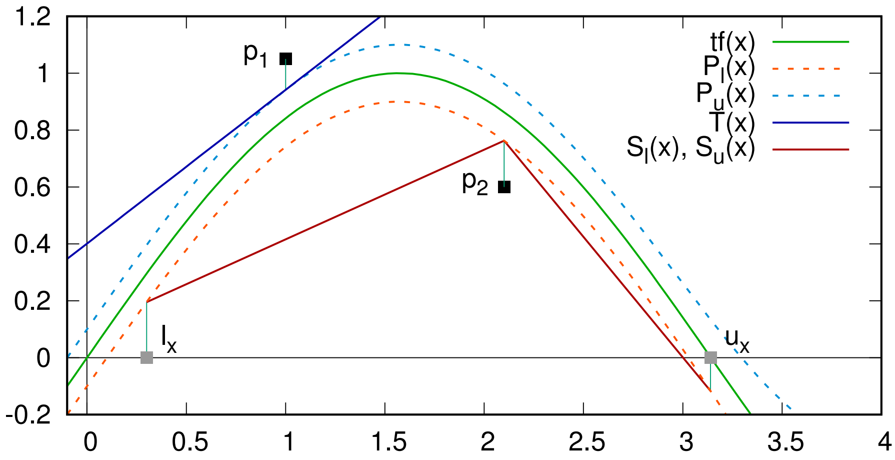


Figure 6.7: Piecewise-linear refinement illustration

```
⟨polynomial, polynomial⟩ GET-POLYNOMIAL-BOUNDS ($f_{\text{TF}}(x)$, $\widehat{\mu}$, $\epsilon$):
$c := \widehat{\mu}[x]$
$\text{conc}_{\text{TF}}$ := GET-CONCAVITY (TF, $c$)
$i := 1$
while true:
    $\langle P_l(x), P_u(x)\rangle$ := MACLAURIN-APPROX (TF, $i$, $x$, $c$)
    $\delta := P_u(c) - P_l(c)$
    $\text{conc}_l$ := GET-CONCAVITY ($P_l$, $c$)
    $\text{conc}_u$ := GET-CONCAVITY ($P_u$, $c$)
    if $\text{conc}_l = \text{conc}_u = \text{conc}_{\text{TF}}$ and $\delta \leq \epsilon$:
        return $\langle P_l(x), P_u(x)\rangle$
    else:
        $i := i + 1$
```

Figure 6.8: Polynomial bounds computation for transcendental functions

be irrational, which means that the constraints of the tangent/secant line can not be readily dealt with by an SMT($\mathcal{UFLRA}$) solver. As shown in Fig. 6.7, the idea is to rely on polynomials to approximate the transcendental function, making sure that they also agree on the concavity with the transcendental function. In this way, the polynomial P_u approximating TF from above, depicted as a dashed blue line, has a tangent that is guaranteed to approximate TF from above (see Proposition 2.9 in Chapter 2). Similarly, for the (red dashed) polynomial P_l approximating TF from below, any piecewise linear combination of secants is guaranteed to approximate TF from below. The key property of polynomials is that the coefficients for tangents and secants are guaranteed to be rationals, and thus amenable to $\mathcal{LRA}$ reasoning. These polynomials are computed using the Maclaurin series of the corresponding transcendental function and Taylor's theorem. Notice that, we use the Maclaurin series (i.e., the Taylor series centered around 0) because we can always compute the exact derivative of any order at 0 for the transcendental functions we support, namely the exponential (exp) and the sine (sin) function. In fact, $\exp(0) = 1$, $\sin(0) = 0$, $\exp^{(i)}(x) = \exp(x)$ for all i, and $|\sin^{(i)}(x)|$ is $|\cos(x)|$ if i is odd and $|\sin(x)|$ otherwise. Thus, the computation of the Maclaurin series and of the remainder polynomial is exact.

The rest of the primitive BLOCK-SPURIOUS-NTA-TERM (lines 4-26), which blocks spurious transcendental terms, is based on the above considerations. If the concavity is positive (resp. negative) or equal to zero, and the point lies below (resp. above) the function, then the linear approximation is given by a tangent to the lower (resp. upper) bound polynomial P_l (resp. P_u) at $\widehat{\mu}[x]$ (lines 7–12 of Fig. 6.9); otherwise, i.e., the concavity is negative (resp. positive) and the point is below (resp. above) the function, the linear approximation is given by a pair of secants to the lower (resp. upper) bound polynomial P_l (resp. P_u) around $\widehat{\mu}[x]$ (lines 13–25 of Fig. 6.9).

In the case of tangent refinement, the function GET-TANGENT-BOUNDS (line 10) returns an interval $[l, u]$ inside which the tangent line is guaranteed not to cross the transcendental function TF. In practice, this interval can be (under)approximated quickly by exploiting known properties of the specific function TF under consideration. For example, for the exponential

```
constraint-set BLOCK-SPURIOUS-NTA-TERM ($\text{TF}(x)$, $\widehat{\mu}$, $P_l(x)$, $P_u(x)$):
    # basic refinement
    $\Gamma$ := BLOCK-SPURIOUS-NTA-BASIC ($\text{TF}(x)$, $\widehat{\mu}$)
    if $\Gamma \neq \emptyset$:
        return $\Gamma$
    # general refinement
    $c := \widehat{\mu}[x]$
    $v := \widehat{\mu}[f_{\text{TF}}(x)]$
    conc := GET-CONCAVITY ($\text{TF}(x)$, $c$)
    if ($v \leq P_l(c)$ and conc $\geq 0$) or ($v \geq P_u(c)$ and conc $\leq 0$):
    # tangent refinement
        $P := (v \leq P_l(c))\ ?\ (P_l)\ :\ (P_u)$
        $T(x) := \text{TanLine}_{P,c}(x)$   # tangent of $P$ at $c$
        $\langle l, u\rangle$ := GET-TANGENT-BOUNDS ($\text{TF}(x)$, $c$, $\frac{d}{dx}P(c)$)
        $\psi := (\text{conc} < 0)\ ?\ (f_{\text{TF}}(x) \leq T(x))\ :\ (f_{\text{TF}}(x) \geq T(x))$
        $\Gamma := \{((x \geq l) \wedge (x \leq u)) \rightarrow \psi\}$
    else:   # $(v \leq P_l(c) \wedge conc < 0) \vee (v \geq P_u(c) \wedge conc > 0)$
    # secant refinement
        prev := GET-PREVIOUS-SECANT-POINTS ($\text{TF}(x)$)
        $l := \max\{p \in \text{prev} \mid p < c\}$
        $u := \min\{p \in \text{prev} \mid p > c\}$
        $P := (v \leq P_l(c))\ ?\ (P_l)\ :\ (P_u)$
        $S_l(x) := \text{SecLine}_{P,l,c}(x)$   # secant of $P$ between $l$ and $c$
        $S_u(x) := \text{SecLine}_{P,c,u}(x)$
        $\psi_l := (\text{conc} < 0)\ ?\ (f_{\text{TF}}(x) \geq S_l(x))\ :\ (f_{\text{TF}}(x) \leq S_l(x))$
        $\psi_u := (\text{conc} < 0)\ ?\ (f_{\text{TF}}(x) \geq S_u(x))\ :\ (f_{\text{TF}}(x) \leq S_u(x))$
        $\phi_l := (x \geq l) \wedge (x \leq c)$
        $\phi_u := (x \geq c) \wedge (x \leq u)$
        STORE-SECANT-POINT ($\text{TF}(x)$, $c$)
        $\Gamma := \{(\phi_l \rightarrow \psi_l), (\phi_u \rightarrow \psi_u)\}$
    return $\Gamma$
```

Figure 6.9: Piecewise-linear refinement for the transcendental function $\text{TF}(x)$ at point c

function GET-TANGENT-BOUNDS always returns $[-\infty, +\infty]$; for other functions, the computation can be based, e.g., on an analysis of the (known, precomputed) inflection points of TF around the point of interest $\widehat{\mu}[x]$ and the slope $\frac{d}{dx}P(c)$ of the tangent line.

In the case of secant refinement, a second value, different from $\widehat{\mu}[x]$, is required to draw a secant line. The function GET-PREVIOUS-SECANT-POINTS returns the set of all the points at which a secant refinement was performed in the past for $\text{TF}(x)$. From this set, we take the two points l and u closest to $\widehat{\mu}[x]$, such that $l < \widehat{\mu}[x] < u$ and that there is no inflection point in $[l, u]$ [2], and use those points to generate two secant lines and their validity intervals. Before returning the set of the two corresponding constraints, we also store the new secant refinement point $\widehat{\mu}[x]$ by calling STORE-SECANT-POINT.

Remark 3. Similarly to the case of $\mathcal{NRA}$ (see Remark 2), we remark that also the above description is only one of the possible strategies for refinement, and that in particular it is possible to adopt lazier variants.

Example 6.2. In order to rule out a spurious interpretation $\widehat{\mu}[x] = 2.0$, $\widehat{\mu}[f_{\exp}(x)] = 3.0$ (where $f_{\exp}(x)$ is the abstraction of the exponential function) we may exploit the positive concavity of $\exp(x)$ and obtain a linear lower-bound constraint, e.g., $f_{\exp}(x) > \frac{155}{21} + \frac{331}{45} * (x - 2)$. Notice that, $\exp(2.0) \approxeq 7.389$, $\frac{155}{21} \approxeq 7.381 \lessapprox \exp(2.0)$, and $\frac{331}{45} \approxeq 7.356 \lessapprox \frac{d}{dx}\exp(2.0)$. These values are such that the above linear constraint "approximates" the tangent of $\exp(x)$ in $x = 2$, since it always lower-bounds $\exp(x)$ and its value and derivative are very near to those of $\exp(x)$ for $x = 2.0$. △

In the following, we discuss our approach for generating refinement constraints for the transcendental functions exp and sin. Other transcendental

[2] For simplicity, we assume that this is always possible. If needed, this can be implemented, e.g., by generating the two points at random while ensuring that $l < \widehat{\mu}[x] < u$ and that there is no inflection point in $[l, u]$.

$$\begin{aligned}
\textit{Lower Bound:}\quad & \forall y.(f_{\mathrm{exp}}(y) > 0)\\
\textit{Zero:}\quad & \forall y.(y = 0 \leftrightarrow f_{\mathrm{exp}}(y) = 1)\\
& \forall y.(y < 0 \leftrightarrow f_{\mathrm{exp}}(y) < 1)\\
& \forall y.(y > 0 \leftrightarrow f_{\mathrm{exp}}(y) > 1)\\
\textit{Zero Tangent Line:}\quad & \forall y.(y \neq 0 \leftrightarrow f_{\mathrm{exp}}(y) > y + 1)\\
\textit{Monotonicity:}\quad & \forall y_1, y_2.(y_1 < y_2 \leftrightarrow f_{\mathrm{exp}}(y_1) < f_{\mathrm{exp}}(y_2))
\end{aligned}$$

Figure 6.10: Basic constraint schemata for the exponential function

functions such as log, cos, tan, arcsin, arccos, arctan can be handled by means of rewriting. For example, $\cos(x)$ is rewritten to $\sin(x + \frac{\pi}{2})$, whereas if φ contains $\log(x)$, we rewrite it as $\varphi\{\log(x) \mapsto l_x\} \wedge \exp(l_x) = x$, where l_x is a fresh variable.

The Exponential Function

Basic Linear Constraints. Our implementation of BLOCK-SPURIOUS-NTA-BASIC for exp uses the linear constraint schemata in Fig. 6.10. For each exponential function $\exp(x)$ violating its rational bounds, we instantiate the basic constraint schemata with $\widehat{\mu}[x]$; if the result evaluates to false, we generate the corresponding instantiation by replacing the quantified variable y with x and removing the quantifier. In the case of the monotonicity constraint schema, we check all possible combinations of exponential function applications $\exp(x_1)$ and $\exp(x_2)$ that are violating their rational bounds.

Polynomial Approximation. Since $\frac{d}{dx}\exp(x) = \exp(x)$, all the derivatives of exp are positive. The polynomial $P_{n,\exp,0}(x)$ is given by the Maclaurin series

$$P_{n,\exp,0}(x) = \sum_{i=0}^{n} \frac{x^i}{i!}$$

and behaves differently depending on the sign of x. Thus, GET-POLYNOMIAL-BOUNDS distinguishes three cases for finding the polynomials $P_l(x)$ and $P_u(x)$:

Case $x = 0$: since $\exp(0) = 1$, we have $P_l(0) = P_u(0) = 1$;

Case $x < 0$: we have that $P_{n,\exp,0}(x) < \exp(x)$ if n is odd, and $P_{n,\exp,0}(x) > \exp(x)$ if n is even; we therefore set $P_l(x) = P_{n,\exp,0}(x)$ and $P_u(x) = P_{n+1,\exp,0}(x)$ for a suitable n so that the required precision ϵ is met;

Case $x > 0$: we have that $P_{n,\exp,0}(x) < \exp(x)$ and $P_{n,\exp,0}(x) * (1 - \frac{x^{n+1}}{(n+1)!})^{-1} > \exp(x)$ when $(1 - \frac{x^{n+1}}{(n+1)!}) > 0$, therefore we set $P_l(x) = P_{n,\exp,0}(x)$ and $P_u(x) = P_{n,\exp,0}(x) * (1 - \frac{x^{n+1}}{(n+1)!})^{-1}$ for a suitable n. [3]

Since the concavity of exp is always positive, the tangent refinement will always give lower bounds for $\exp(x)$, and the secant refinement will give upper bounds. Moreover, as exp has no inflection points, GET-TANGENT-BOUNDS always returns $[-\infty, +\infty]$.

The Sine Function

Dealing with Periodicity: Base Period Shifting. The correctness of our refinement procedure relies crucially on being able to compute the concavity of the transcendental function TF at a given point c. This is needed in order to know whether a computed tangent or secant line constitutes a valid upper or lower bound for TF around c (see Fig. 6.7 and 6.9). In the case of the sin function, computing the concavity at an arbitrary point c is problematic, since this essentially amounts to computing the value $c' \in [-\pi, \pi[$ s.t. $c = 2\pi n + c'$ for some integer n, because in $[-\pi, \pi[$ the concavity of $\sin(c')$ is the opposite of the sign of c'. This is not easy to compute because π is a transcendental number.

[3] We slightly abuse the notation: $P_u(x)$ is not a polynomial but a rational function.

In order to solve this problem, we exploit another property of sin, namely its periodicity (with period 2π). More precisely, we split the reasoning about sin depending on two kinds of periods: base period, with argument from $-\pi$ to π, and extended period. For each $\sin(x)$ term we introduce an "artificial" sin term $\sin(\omega_x)$, where ω_x is a fresh variable called *base variable.* Base variables are constrained to be interpreted over the base period, where the sin value for the corresponding variable in the extended period is computed. This is done by adding the following constraint during formula preprocessing:

$$\varphi_{shift} \quad \doteq \quad \bigwedge_{\sin(x)\in\varphi} \left(\begin{array}{l} -\pi \leq \omega_x < \pi \ \wedge \ \sin(x) = \sin(\omega_x)) \wedge \\ ((-\pi \leq x < \pi) \rightarrow x = \omega_x) \end{array} \right) \tag{6.1}$$

The first conjunct constrains ω_x to the base period. The second conjunct constrains $\sin(x)$ to have the same value as $\sin(\omega_x)$. The third conjunct states that if x is interpreted in the base period then it has the same value as its base variable. In order to reason about the irrational π, we introduce a variable $\widehat{\pi}$, and add the constraint $l_\pi < \widehat{\pi} < u_\pi$ to φ. l_π and u_π are valid rational lower and upper bounds for the actual value of π that can be computed with various methods. Using this transformation, we can easily compute the concavity of sin at $\widehat{\mu}[\omega_x]$ by just looking at the sign of $\widehat{\mu}[\omega_x]$, provided that $-l_\pi \leq \widehat{\mu}[\omega_x] \leq l_\pi$, where l_π is the current lower bound for $\widehat{\pi}$. (We recall that in the interval $[-\pi, \pi[$, the concavity of $\sin(c)$ is the opposite of the sign of c.)

Let $\mathcal{F}^{\mathrm{B}}_{\sin}$ be the set of $f_{\sin}(\omega_x)$ terms in $\widehat{\varphi}$ that have base variables as arguments, $\mathcal{F}_{\sin}$ be the set of all $f_{\sin}(x)$ terms, and $\mathcal{F}^{\mathrm{E}}_{\sin} \doteq \mathcal{F}_{\sin} \setminus \mathcal{F}^{\mathrm{B}}_{\sin}$, where $f_{\sin}$ is the uninterpreted function that represents sin in $\widehat{\varphi}$. Both the basic linear refinement and the tangent/secant refinement is performed for the terms in $\mathcal{F}^{\mathrm{B}}_{\sin}$ only; we then use *linear shift* constraints (described below) for refining terms in $\mathcal{F}^{\mathrm{E}}_{\sin}$, as follows.

For each $f_{\sin}(x) \in \mathcal{F}^{\mathrm{E}}_{\sin}$ with the corresponding base variable ω_x, we check whether the value $\widehat{\mu}[x]$ after shifting to the base period is equal to the value of $\widehat{\mu}[\omega_x]$. We calculate the integer shift value s of x as the rounding towards zero of $(\widehat{\mu}[x] + \widehat{\mu}[\widehat{\pi}])/(2 * \widehat{\mu}[\widehat{\pi}])$, and we then compare $\widehat{\mu}[\omega_x]$ with $\widehat{\mu}[x] - 2 * s * \widehat{\mu}[\widehat{\pi}]$. If the values are different, we add the following *linear shift* constraint for relating x with ω_x in the extended period s:

$$\left(\begin{array}{l} -\widehat{\pi} \leq \omega_x < \widehat{\pi} \ \wedge \ f_{\sin}(x) = f_{\sin}(\omega_x) \ \wedge \\ \widehat{\pi} * (2 * s - 1) \leq x < \widehat{\pi} * (2 * s + 1)) \end{array}\right) \rightarrow \ \omega_x = x - 2 * s * \widehat{\pi}.$$

In this way, we do not need the tangent and secant refinement for the extended period and we can reuse the refinements done in the base period. Notice that, even if the calculated shift value is wrong (due to the imprecision of $\widehat{\mu}[\widehat{\pi}]$ with respect to the real value π), the constraint we generate may be useless, but it is never wrong.

Basic Linear Constraints. We use the constraint schemata of Fig. 6.11 for implementing BLOCK-SPURIOUS-NTA-BASIC for sin. As written above, these constraints are only checked for terms in $\mathcal{F}^{\mathrm{B}}_{\sin}$.

Polynomial Approximation. For each term $f_{\sin}(\omega_x)$ that needs to be refined, we first check whether $\widehat{\mu}[\omega_x] \in [-l_\pi, l_\pi]$, where l_π is the current lower bound for $\widehat{\pi}$. If this is the case, then we derive the concavity of sin at $\widehat{\mu}[\omega_x]$ by just looking at the sign of $\widehat{\mu}[\omega_x]$. We can therefore perform tangent or secant refinement as shown in Fig. 6.9. More precisely, GET-POLYNOMIAL-BOUNDS finds the lower and upper polynomials using Taylor's theorem, which ensures that:

$$P_{n,\sin,0}(\omega_x) - R^{\mathrm{U}}_{n+1,\sin,0}(\omega_x) \leq \sin(\omega_x) \leq P_{n,\sin,0}(\omega_x) + R^{\mathrm{U}}_{n+1,\sin,0}(\omega_x)$$

$$\begin{aligned}
\textit{Symmetry:}\ & \forall\omega_x.(f_{\sin}(\omega_x) = -f_{\sin}(-\omega_x))\\
\textit{Phase:}\ & \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (0 < \omega_x \leftrightarrow f_{\sin}(\omega_x) > 0))\\
& \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (-\widehat{\pi} < \omega_x < 0 \leftrightarrow f_{\sin}(\omega_x) < 0))\\
\textit{Zero Tangent:}\ & \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (\omega_x > 0 \leftrightarrow f_{\sin}(\omega_x) < \omega_x))\\
& \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (\omega_x < 0 \leftrightarrow f_{\sin}(\omega_x) > \omega_x))\\
\pi\ \textit{Tangent:}\ & \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (f_{\sin}(\omega_x) < -\omega_x + \widehat{\pi}))\\
& \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (\omega_x > -\widehat{\pi} \leftrightarrow f_{\sin}(\omega_x) > -\omega_x - \widehat{\pi}))\\
\textit{Significant Values:}\ & \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (f_{\sin}(\omega_x) = 0 \leftrightarrow (\omega_x = 0 \vee \omega_x = -\widehat{\pi})))\\
& \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (f_{\sin}(\omega_x) = 1 \leftrightarrow \omega_x = \frac{\widehat{\pi}}{2}))\\
& \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (f_{\sin}(\omega_x) = -1 \leftrightarrow \omega_x = -\frac{\widehat{\pi}}{2}))\\
& \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (f_{\sin}(\omega_x) = \frac{1}{2} \leftrightarrow (\omega_x = \frac{\widehat{\pi}}{6} \vee \omega_x = \frac{5 * \widehat{\pi}}{6})))\\
& \forall\omega_x.(-\widehat{\pi} \leq \omega_x < \widehat{\pi} \rightarrow (f_{\sin}(\omega_x) = -\frac{1}{2} \leftrightarrow (\omega_x = -\frac{\widehat{\pi}}{6} \vee \omega_x = -\frac{5 * \widehat{\pi}}{6})))\\
\textit{Monotonicity:}\ & \forall\omega_{x_1},\omega_{x_2}.(-\widehat{\pi} \leq \omega_{x_1} < \omega_{x_2} \leq -\frac{\widehat{\pi}}{2} \rightarrow f_{\sin}(\omega_{x_1}) > f_{\sin}(\omega_{x_2}))\\
& \forall\omega_{x_1},\omega_{x_2}.(-\frac{\widehat{\pi}}{2} \leq \omega_{x_1} < \omega_{x_2} \leq \frac{\widehat{\pi}}{2} \rightarrow f_{\sin}(\omega_{x_1}) < f_{\sin}(\omega_{x_2}))\\
& \forall\omega_{x_1},\omega_{x_2}.(\frac{\widehat{\pi}}{2} \leq \omega_{x_1} < \omega_{x_2} < \widehat{\pi} \rightarrow f_{\sin}(\omega_{x_1}) > f_{\sin}(\omega_{x_2}))
\end{aligned}$$

Figure 6.11: Basic constraint schemata for sin function

where

$$P_{n,\sin,0}(\omega_x) = \sum_{k=0}^{n} \frac{(-1)^k * \omega_x^{2k+1}}{(2k+1)!}$$
$$R^{\mathrm{U}}_{n+1,\sin,0}(\omega_x) = \frac{\omega_x^{2(n+1)}}{(2(n+1))!}$$

We set $P_l(x) = P_{n,\sin,0}(x) - R^{\mathrm{U}}_{n+1,\sin,0}(x)$ and $P_u(x) = P_{n,\sin,0}(x) + R^{\mathrm{U}}_{n+1,\sin,0}(x)$. Under the above hypothesis that $\widehat{\mu}[\omega_x] \in [-l_\pi, l_\pi]$, also the function GET-TANGENT-BOUNDS can easily be implemented by looking at the sign of $\widehat{\mu}[\omega_x]$: if $\widehat{\mu}[\omega_x] \geq 0$, then the validity interval is $[0, \widehat{\pi}[$, otherwise, it is $[-\widehat{\pi}, 0]$.

The remaining case to discuss is when the value of ω_x in $\widehat{\mu}$ is not within the interval $[-l_\pi, l_\pi]$ (which means that $|\widehat{\mu}[\omega_x]| \in (l_\pi, u_\pi])$. In this case, we cannot reliably compute the concavity of sin at $\widehat{\mu}[\omega_x]$. Therefore, instead of performing a tangent/secant refinement, we refine the precision of $\widehat{\pi}$ by computing a tighter interval (l'_π, u'_π) for it, using Machin's formula [BBB13].[4] For a positive integer n:

$$\pi > 4 * \sum_{k=0}^{2n+1} \left(\frac{(-1)^k}{2k+1} * \left(\frac{4}{5^{2k+1}} - \frac{1}{239^{2k+1}} \right) \right)$$
$$\pi < 4 * \sum_{k=0}^{2(n+1)} \left(\frac{(-1)^k}{2k+1} * \left(\frac{4}{5^{2k+1}} - \frac{1}{239^{2k+1}} \right) \right)$$

6.3 Spuriousness Check and Detecting Satisfiability

We concentrate now on the problem of checking the spuriousness of the abstract model $\widehat{\mu}$ and of detecting the existence of solutions for satisfiable formulae. We deal with $\mathcal{NRA}$ first in §6.3.1, and then consider $\mathcal{NTA}$ in §6.3.2.

6.3.1 Finding Rational Models for $\mathcal{NRA}$

We first describe the behavior of the function CHECK-MODEL restricted to $\mathcal{NRA}$, which we represent by the function CHECK-NRA-MODEL. Notice that the parameter ϵ has no role for $\mathcal{NRA}$, so that CHECK-NRA-MODEL receives as input only $\widehat{\varphi}$ and $\widehat{\mu}$.

In its simplest form, the function CHECK-NRA-MODEL could simply be implemented by checking if $\widehat{\mu}[x] * \widehat{\mu}[y] = \widehat{\mu}[f_*(x, y)]$ for every multiplication term $f_*(x, y) \in \widehat{\varphi}$, returning **true** if and only if this is the case. It is easy to see, however, that this very simple algorithm can return **true** only if the

[4] This is not explicitly shown in the pseudocode of Fig. 6.9, but it is part of BLOCK-SPURIOUS-NTA-BASIC.

```
⟨bool, model⟩ CHECK-NRA-MODEL ($\widehat{\varphi}$, $\widehat{\mu}$):
    $\widehat{\psi}$ := GET-ASSIGNMENT($\widehat{\varphi}, \widehat{\mu}$)          # truth assignment induced by $\widehat{\mu}$ (6.2)
    $\widehat{\psi}^*$ := $\widehat{\psi} \wedge$ LINEARIZATION-AXIOMS($\widehat{\psi}$)   # add multiplication-line constraints (6.3)
    return SMT-UFLRA-CHECK ($\widehat{\psi}^*$)
```

Figure 6.12: An incomplete procedure using an SMT($\mathcal{UFLRA}$) solver

$\mathcal{UFLRA}$ solver "guesses" a model that is consistent with all the nonlinear multiplications. In an infinite and dense domain like the rationals or the reals, the chances that this will happen are very low in general.

Thus, in practice it is not enough for CHECK-NRA-MODEL to check the spuriousness of $\widehat{\mu}$. In order to detect satisfiable cases more effectively in the very-likely case in which $\widehat{\mu}$ is spurious, we also want that CHECK-NRA-MODEL searches for the existence of an actual model for φ "in the surroundings" of $\widehat{\mu}$. Our idea is to extract the truth assignment $\widehat{\psi}$ induced by $\widehat{\mu}$ on the atoms of $\widehat{\varphi}$:

$$\widehat{\psi} \doteq \bigwedge_{[\widehat{A}\ \in\ \text{atoms}(\widehat{\varphi})\ \text{s.t.}\widehat{\mu}\models\widehat{A}]} \widehat{A}\ \wedge \bigwedge_{[\widehat{A}\ \in\ \text{atoms}(\widehat{\varphi})\ \text{s.t.}\widehat{\mu}\not\models\widehat{A}]} \neg\widehat{A}, \tag{6.2}$$

and then to look for another model $\widehat{\eta}$ for $\widehat{\psi}$. Notice that, any such model $\widehat{\eta}$ shares with $\widehat{\mu}$ the truth assignment on the atoms $\widehat{\psi}$, but with different values of the real variables.

The algorithm we propose is outlined in Fig. 6.12, where we extract the truth assignment $\widehat{\psi}$ induced by the $\mathcal{UFLRA}$ model $\widehat{\mu}$ on the atoms of $\widehat{\varphi}$, and we conjoin to it the *multiplication-line constraints*:

$$\widehat{\psi}^* \quad = \quad \widehat{\psi}\ \wedge \bigwedge_{f_*(x,y)\in\widehat{\psi}} \begin{pmatrix} (\ x = \widehat{\mu}[x] \wedge f_*(x,y) = \widehat{\mu}[x] * y\ ) \vee \\ (\ y = \widehat{\mu}[y] \wedge f_*(x,y) = \widehat{\mu}[y] * x\ ) \end{pmatrix}. \tag{6.3}$$

The main idea is to build an $\mathcal{UFLRA}$ under-approximation $\widehat{\psi}^*$ of the NRA formula ψ, in which all multiplications are forced to be linear. Notice that, this corresponds to searching for a solution along the multiplication lines described in §6.2.1, Fig. 6.3c and Fig. 6.3d.

Example 6.3. Consider the following formula φ:

$$\varphi \doteq x * y = 10 \wedge (2 \leq x \leq 4) \wedge (2 \leq y \leq 4).$$

Then, $\widehat{\varphi} \doteq f_*(x, y) = 10 \wedge (2 \leq x \leq 4) \wedge (2 \leq y \leq 4)$. Suppose the following model $\widehat{\mu}$ is returned by SMT-UFLRA-CHECK (line 6 in Fig. 6.1.2):

$$\widehat{\mu}[x] = 2,\ \widehat{\mu}[y] = 4,\ \widehat{\mu}[f_*(x, y)] = 10.$$

$\widehat{\mu}$ is a spurious interpretation since $2 * 4 \neq 10$ in $\mathcal{NRA}$. However, using CHECK-NRA-MODEL we can still find an $\mathcal{NRA}$-compliant model from the "guesses", which solves the following $\mathcal{UFLRA}$-satisfiable formula (see (6.3)):

$$\begin{aligned}\widehat{\psi} \doteq\ & f_*(x, y) = 10 \wedge (2 \leq x \leq 4) \wedge (2 \leq y \leq 4) \wedge \\ & ((x = 2 \wedge f_*(x, y) = 2 * y) \vee (y = 4 \wedge f_*(x, y) = 4 * x)).\end{aligned}$$

A possible $\mathcal{UFLRA}$-model $\widehat{\mu}^*$ for $\widehat{\varphi}$ is:

$$\widehat{\mu}^*[x] = \frac{5}{2},\ \widehat{\mu}^*[y] = 4,\ \widehat{\mu}^*[f_*(x, y)] = 10,$$

that is also compliant with $\mathcal{NRA}$. △

Given the simplicity of the Boolean structure of the under-approximated formula, the check should in general be very cheap. In substance, we trade the complexity of $\mathcal{NRA}$-solving with some extra Boolean reasoning. The drawback is that this is (clearly) still an incomplete procedure (see Example 6.4). However, in our experiments (for which we refer to §7.2) we have found it to be surprisingly effective for many problems.

Example 6.4. Consider the following $\mathcal{NRA}$-satisfiable formula:

$$\varphi \doteq x * x = 2.$$

```
bool CHECK-MODEL ($\widehat{\varphi}$, $\widehat{\mu}$, $\epsilon$):
    $\langle sat, \widehat{\mu}^* \rangle$ := CHECK-NRA-MODEL ($\widehat{\varphi}$, $\widehat{\mu}$)
    if $sat$:
        let $\varphi^{\text{sat}}_{\widehat{\mu}^*}$ be the formula as defined in (6.5)
        return not SMT-LRA-CHECK ($\neg\varphi^{\text{sat}}_{\widehat{\mu}^*}$)
    else:
        return false
```

Figure 6.13: Detecting satisfiability using an SMT($\mathcal{LRA}$) solver

CHECK-NRA-MODEL would never find a model that is compliant with $\mathcal{NRA}$, because it is driven by the “guesses” returned by SMT-UFLRA-CHECK which only produces rational number guesses (line 6 in Fig. 6.1.2) and φ has models with irrational numbers only. Moreover, for the same reason the procedure SMT-NTA-CHECK would never terminate and will keep performing the refinement. △

6.3.2 Detecting Satisfiability with $\mathcal{NTA}$

We describe now how to extend the CHECK-MODEL procedure to deal with transcendental functions. The pseudo-code for CHECK-MODEL is shown in Fig. 6.13. As already written earlier, since our $\mathcal{UFLRA}$ solver is not able to deal with irrational numbers, in general we are not able to precisely represent a model μ for a formula with transcendental functions, since in most cases the model value for a term TF(x) is irrational if the value for x is rational. [5]

In general, therefore, we are not able to construct a model for a formula with transcendental functions. However, we may exploit this simple observation: we can still conclude that φ is satisfiable if we are able to show that $\widehat{\varphi}$ is satisfiable *under all possible interpretations of* f_{TF} that are

[5] With the notable exception of 0, at which both exp and sin have rational values.

guaranteed to include also TF. In order to do this, we proceed as follows.

Starting from the abstract model $\widehat{\mu}$ for $\widehat{\varphi}$, we first try to obtain a model $\widehat{\mu}^*$ that is consistent with multiplication terms, using the procedure CHECK-NRA-MODEL described in §6.3.1, while still treating all the transcendental functions as uninterpreted. Then, we compute safe lower and upper bounds $\underline{\text{TF}(\widehat{\mu}^*[x])}_l$ and $\overline{\text{TF}(\widehat{\mu}^*[x])}^u$ for the function TF at point $\widehat{\mu}^*[x]$ with the GET-POLYNOMIAL-BOUNDS function (see §6.2.2), with the current value of ϵ.

Let ψ be the formula obtained by substituting every $f_*(x,y) \in \widehat{\varphi}$ by $x*y$ and every variable $x \in \widehat{\varphi}$ by $\widehat{\mu}^*[x]$ with the exception of $\widehat{\pi}$. [6] We notice that, the satisfiability of the original formula φ follows from the validity of the following formula:

$$\rho \doteq \left(l_\pi < \widehat{\pi} < u_\pi \wedge \bigwedge_{f_{\text{TF}}(x) \in \widehat{\varphi}} \underline{\text{TF}(\widehat{\mu}^*[x])}_l \leq f_{\text{TF}}(\widehat{\mu}^*[x]) \leq \overline{\text{TF}(\widehat{\mu}^*[x])}^u \right) \rightarrow \psi, \tag{6.4}$$

that is, from the fact that ψ holds for all possible interpretations of $\widehat{\pi}$ and the uninterpreted functions f_{TF} which fit in the bounds in the given points. In fact, since by construction $l_\pi < \pi < u_\pi \wedge \underline{\text{TF}(\widehat{\mu}^*[x])}_l \leq \text{TF}(\widehat{\mu}^*[x]) \leq \overline{\text{TF}(\widehat{\mu}^*[x])}^u$, then the validity of the above formula implies that the formula is satisfied also by all interpretations which assign to $\widehat{\pi}$ the value of π and all $f_{\text{TF}}(\widehat{\mu}^*[x])$ the values of $\text{TF}(\widehat{\mu}^*[x])$, which by construction satisfy the original formula φ.

In order to be able to use a quantifier-free SMT($\mathcal{LRA}$)-solver, we reduce the problem to the validity check of an $\mathcal{LRA}$ formula. Let CT be the set of all terms $f_{\text{TF}}(\widehat{\mu}^*[x])$ occurring in ψ. We replace each occurrence of $f_{\text{TF}}(\widehat{\mu}^*[x])$ in CT with a corresponding fresh variable $y_{f_{\text{TF}}(\widehat{\mu}^*[x])}$ from a set

[6] Notice that, we are treating π as a zero-argument transcendental function.

Y. We then check the validity of the formula:

$$\varphi^{\text{sat}}_{\widehat{\mu}^*} \doteq \forall \widehat{\pi}, Y.(\rho\{CT \mapsto Y\}). \tag{6.5}$$

If $\neg\varphi^{\text{sat}}_{\widehat{\mu}^*}$ is unsatisfiable, we conclude that φ is $\mathcal{NTA}$-satisfiable. Clearly, this can be checked with a quantifier-free SMT($\mathcal{LRA}$)-solver, since $\neg\forall x.\phi$ is equivalent to $\exists x.\neg\phi$, and x can then be removed by Skolemization.

Example 6.5. Consider the following $\mathcal{NTA}$-satisfiable formula:

$$\varphi \doteq \exp(x) > 0,$$

and its initial abstraction:

$$\widehat{\varphi} \doteq f\text{exp}(x) > 0.$$

We demonstrate an execution of CHECK-MODEL (Fig. 6.13). Let $\widehat{\mu}^*$ be a model returned by CHECK-NRA-MODEL (line 1 in Fig. 6.13), where:

$$\widehat{\mu}^*[x] = 1,\ \widehat{\mu}^*[f\text{exp}(x)] = 1.$$

Using bounds computed with GET-POLYNOMIAL-BOUNDS (Fig. 6.8), (6.4) and (6.5) become:

$$\begin{aligned}
\rho &\doteq \left((\frac{333}{106} < \widehat{\pi} < \frac{355}{113}) \wedge (\frac{65}{24} \leq f\text{exp}(1) \leq \frac{325}{119}) \right) \to f\text{exp}(1) > 0 \\
\varphi^{\text{sat}}_{\widehat{\mu}^*} &\doteq \forall \widehat{\pi}, y. \left(\left(\frac{333}{106} < \widehat{\pi} < \frac{355}{113}) \wedge (\frac{65}{24} \leq y \leq \frac{325}{119}) \right) \to y > 0 \right) \\
\neg\varphi^{\text{sat}}_{\widehat{\mu}^*} &\doteq (\frac{333}{106} < \widehat{\pi} < \frac{355}{113}) \wedge (\frac{65}{24} \leq y \leq \frac{325}{119}) \wedge y \leq 0
\end{aligned}$$

Note that $\neg\varphi^{\text{sat}}_{\widehat{\mu}^*}$ clearly $\mathcal{LRA}$-unsatisfiable because of the constraints on y and that can be shown by using any complete SMT($\mathcal{LRA}$) solver. Therefore, CHECK-MODEL returns true. △

6.4 Proofs of Correctness

We prove the correctness of the SMT-NTA-CHECK procedure (Fig. 6.1), by first proving that the procedure maintains an over-approximation of the input problem φ. This is done by showing that the SMT-PREPROCESS function (Line 1) is $\mathcal{NTA}$-satisfiability preserving – Lemma 6.6; and that the over-approximation is maintained in the loop (Line 5-12) – Lemma 6.7.

In what follows φ' is the result of SMT-PREPROCESS(φ), so that $\varphi' \doteq \varphi \wedge \varphi_{shift}$ (see (6.1) in §6.2.2).

Lemma 6.6. *φ is $\mathcal{NTA}$-satisfiable if and only if φ' is $\mathcal{NTA}$-satisfiable.*

Proof. The "if" case is obvious. For the "only if" case, let μ be an $\mathcal{NTA}$-interpretation satisfying φ. Then we can build another $\mathcal{NTA}$-interpretation μ' by extending μ to the new symbols $\omega_x \in \varphi_{shift}$ introduced during preprocessing as follows. For each $\omega_x \in \varphi'$, if $\mu[x] \in [-\pi, \pi[$ then $\mu'[\omega_x] \doteq \mu[x]$; otherwise, we can choose $\mu'[\omega_x] \doteq c$ such that $\mu[sin(c)] = \mu[sin(x)]$, where $c \in \mathbb{R}$ and $c \in [-\pi, \pi[$. We can always choose such c because sin is periodic, and $[-\pi, \pi]$ defines a period for it. Then by construction $\mu' \models \varphi_{shift}$, so that the statement holds. □

Let $\widehat{\varphi}$ be the result of SMT-INITIAL-ABSTRACTION(φ'). Consider a generic loop in SMT-NTA-CHECK(φ) and the value Γ at the end of such loop. By construction all constraints in Γ are either instances of those in Fig. 6.4, Fig. 6.10, Fig. 6.11, or tangent/secant constraints from Fig. 6.9, or linear shift constraints as discussed in §6.2.2.

Lemma 6.7. *If φ is $\mathcal{NTA}$-satisfiable, then $\widehat{\varphi} \wedge \bigwedge \Gamma$ is $\mathcal{UFLRA}$-satisfiable.*

Proof. Let μ be an $\mathcal{NTA}$-interpretation satisfying φ. By Lemma 6.6, we have an $\mathcal{NTA}$-interpretation μ' that satisfies φ'. We build an $\mathcal{UFLRA}$-interpretation $\widehat{\mu}$ satisfying $\widehat{\varphi} \wedge \bigwedge \Gamma$ as follows:

- for each $x \in \varphi'$, $\widehat{\mu}[x] \doteq \mu'[x]$
- for each $\omega_x \in \varphi'$, $\widehat{\mu}[\omega_x] \doteq \mu'[\omega_x]$
- for $\widehat{\pi}$, $\widehat{\mu}[\widehat{\pi}] \doteq \mu'[\pi]$
- for each $f_*(x, y) \in \widehat{\varphi}$, $\widehat{\mu}[f_*(x, y)] \doteq \mu'[x * y]$
- for each $f_{\mathrm{TF}}(x) \in \widehat{\varphi}$, $\widehat{\mu}[f_{\mathrm{TF}}(x)] \doteq \mu'[\textsc{tf}(x)]$

$\widehat{\mu}$ clearly satisfies $\widehat{\varphi}$ and all the constraints (constraints in Fig. 6.4, Fig. 6.10, Fig. 6.11; tangent/secant constraints from Fig. 6.9 and the linear shift constraints as discussed in §6.2.2) in Γ, because they are valid in any theory which interprets $f_*()$, $f_{\mathrm{TF}}()$ and $\widehat{\pi}$ as $*$, $\textsc{tf}()$ and π respectively. Hence the statement holds. □

We now prove the correctness of the method to detect satisfiability. Let φ, φ' be as above and let $\widehat{\varphi}$, $\widehat{\mu}$, *sat*, $\widehat{\mu}^*$, and $\varphi^{\mathrm{sat}}_{\widehat{\mu}^*}$ be as in CHECK-MODEL (Fig. 6.13), so that $\widehat{\mu}$ is an $\mathcal{UFLRA}$-model for $\widehat{\varphi}$. $\varphi^{\mathrm{sat}}_{\widehat{\mu}^*}$ (6.5) is the formula for detecting the $\mathcal{NTA}$-satisfiability of φ as discussed in §6.3.2. (We recall that $\varphi^{\mathrm{sat}}_{\widehat{\mu}^*}$ is a closed $\mathcal{LRA}$ formula.)

Lemma 6.8. *If* CHECK-MODEL$(\varphi, \widehat{\mu}, \epsilon)$ *returns true, then* φ *is* $\mathcal{NTA}$-*satisfiable.*

Proof. From Fig. 6.13, CHECK-MODEL$(\varphi, \widehat{\mu}, \epsilon)$ returns *true* if and only if CHECK-NRA-MODEL returns $\langle True, \widehat{\mu}^* \rangle$ and $\neg\varphi^{\mathrm{sat}}_{\widehat{\mu}^*}$ is $\mathcal{LRA}$-unsatisfiable – that is, $\varphi^{\mathrm{sat}}_{\widehat{\mu}^*}$ is $\mathcal{LRA}$-valid. From Fig. 6.12, if CHECK-NRA-MODEL returns $\langle True, \widehat{\mu}^* \rangle$, then $\widehat{\mu}^*$ is an $\mathcal{UFLRA}$-model of a conjunction of literals $\widehat{\psi}^*$ (6.3) which tautologically entails $\widehat{\varphi}$ and it is s.t. each $f_*(x, y)$ equals $x * y$ in $\widehat{\mu}^*$. Thus $\widehat{\mu} \models \widehat{\varphi}$ and $\widehat{\mu}[f_*(x, y)] = \widehat{\mu}[x] * \widehat{\mu}[y]$ for each term $f_*(x, y) \in \widehat{\varphi}$.

Then we can construct an $\mathcal{NTA}$-interpretation μ for φ' as follows:

- for every $x, \omega_x \in \varphi'$, $\mu[x] \doteq \widehat{\mu}^*[x]$ and $\mu[\omega_x] \doteq \widehat{\mu}^*[\omega_x]$

- for $\widehat{\pi}$, $\mu[\widehat{\pi}] \doteq \pi$
- for every $\textsc{tf}(x) \in \varphi : \mu[\textsc{tf}(x)] \doteq \textsc{tf}(\widehat{\mu}^*[x])$

We show that μ is a model for φ'. The validity of $\varphi^{\text{sat}}_{\widehat{\mu}^*}$, and hence of (6.4), implies that φ' is satisfied by every interpretation extending $\widehat{\mu}^*$ which assigns to each uninterpreted term $f_{\text{TF}}(\widehat{\mu}^*[x])$ a value within the bounds $[\underline{\textsc{tf}(\widehat{\mu}^*[x])}_l, \overline{\textsc{tf}(\widehat{\mu}^*[x])}^u]$ and assigns $\widehat{\pi}$ a value in $]l_\pi, u_\pi[$. μ is one of such interpretations because $\mu[\widehat{\pi}] \doteq \pi$ and $\mu[f_{\text{TF}}(\widehat{\mu}^*[x])] \doteq \textsc{tf}(\widehat{\mu}^*[x])$ which is in $[\underline{\textsc{tf}(\widehat{\mu}^*[x])}_l, \overline{\textsc{tf}(\widehat{\mu}^*[x])}^u]$ by Taylor's Theorem ((2) in Chapter 2). Thus μ is an $\mathcal{NTA}$-model for φ', and hence for φ. Therefore φ is $\mathcal{NTA}$-satisfiable. □

Theorem 6.9. SMT-NTA-CHECK *is sound, i.e., when it returns* $\langle \textbf{true}, \emptyset \rangle$ *then* φ *is* $\mathcal{NTA}$*-satisfiable and when it returns* $\langle \textbf{false}, \Gamma \rangle$ *then* φ *is not* $\mathcal{NTA}$*-satisfiable.*

Proof. SMT-NTA-CHECK returns $\langle \textbf{false}, \Gamma \rangle$ only when SMT-UFLRA-CHECK returns **false**. In this case, by Lemma 6.6 and Lemma 6.7, φ is not $\mathcal{NTA}$-satisfiable.

SMT-NTA-CHECK returns $\langle \textbf{true}, \emptyset \rangle$ only when CHECK-MODEL inside CHECK-REFINE returns **true**. In this case φ is $\mathcal{NTA}$-satisfiable by Lemma 6.8. □

6.5 Modifications for SMT($\mathcal{NIA}$)

Now we focus on solving the SMT($\mathcal{NIA}$) problem. The procedures presented earlier require little modifications to make them also work for the SMT($\mathcal{NIA}$) case. Interestingly, if we replace SMT-UFLRA-CHECK in line 6 of Fig. 6.1 and line 3 of Fig. 6.12 with SMT-UFLIA-CHECK, then we obtain a procedure for solving SMT($\mathcal{NIA}$). The resultant procedure is shown in Fig. 6.14. We have highlighted the modified code with the red

```
$\langle$**bool**, constraint-set$\rangle$ SMT-NIA-CHECK ($\varphi$):
  $\varphi' :=$ SMT-PREPROCESS($\varphi$)
  $\widehat{\varphi} :=$ SMT-INITIAL-ABSTRACTION($\varphi'$)
  $\Gamma := \emptyset$
  **while true**:
        $\langle sat, \widehat{\mu}\rangle :=$ SMT-UFLIA-CHECK ($\widehat{\varphi} \wedge \bigwedge \Gamma$)
        **if not** $sat$:
              **return** $\langle$**false**, $\Gamma\rangle$
        $\langle sat, \Gamma'\rangle :=$ CHECK-REFINE ($\varphi'$, $\widehat{\varphi}$, $\widehat{\mu}$)
        **if** $sat$:
              **return** $\langle$**true**, $\emptyset\rangle$
        $\Gamma := \Gamma \cup \Gamma'$

$\langle$**bool**, constraint-set$\rangle$ CHECK-REFINE ($\varphi$, $\widehat{\varphi}$, $\widehat{\mu}$):
  **if** CHECK-MODEL ($\varphi$, $\widehat{\mu}$):
        **return** $\langle$**true**, $\emptyset\rangle$
  $\Gamma :=$ BLOCK-SPURIOUS-PRODUCT-TERMS($\widehat{\varphi}$, $\widehat{\mu}$)     # *refinement of products*
  **return** $\langle$**false**, $\Gamma\rangle$

**bool** CHECK-MODEL ($\widehat{\varphi}$, $\widehat{\mu}$):
  $\langle sat, \widehat{\mu}^*\rangle :=$ GET-NIA-MODEL ($\widehat{\varphi}$, $\widehat{\mu}$)
  **if** $sat$:
        **return true**
  **else:**
        **return false**

$\langle$**bool**, model$\rangle$ GET-NIA-MODEL ($\widehat{\varphi}$, $\widehat{\mu}$):
  $\widehat{\psi} :=$ GET-ASSIGNMENT($\widehat{\varphi}$, $\widehat{\mu}$)            # *truth assignment induced by* $\widehat{\mu}$ (6.2)
  $\widehat{\psi}^* := \widehat{\psi} \wedge$ LINEARIZATION-AXIOMS($\widehat{\psi}$)   # *add multiplication-line constraints* (6.3)
  **return** SMT-UFLIA-CHECK ($\widehat{\psi}^*$)
```

Figure 6.14: SMT($\mathcal{NIA}$) modifications – abstraction to SMT($\mathcal{UFLIA}$) and refinement

color and removed the code related to transcendental functions since they do not have part of $\mathcal{NIA}$.

Here are some key observations on the SMT($\mathcal{NIA}$) procedure:

- $\widehat{\mu}$ assigns integer values to the variables in the $\mathcal{NIA}$-formula $\widehat{\varphi}$;
- the constraints returned by BLOCK-SPURIOUS-PRODUCT-TERMS refine multiplications between integer variables;
- CHECK-MODEL returns true when it finds an $\mathcal{NIA}$-compliant model using the technique described in §6.3.1;
- CHECK-MODEL does not have to deal with irrational numbers as it was the case for $\mathcal{NRA}$ (see Example 6.4);
- the correctness can be proved in the similar way we did for $\mathcal{NRA}$ (§6.4).

Remark 4. A procedure for the combined theory of $\mathcal{NTA}$ and $\mathcal{NIA}$ can also be obtained by using a SMT-UFLIRA-CHECK instead of SMT-UFLRA-CHECK in line 6 of Fig. 6.1 and line 3 of Fig. 6.12.

6.6 Related Work

Here we compare incremental linearization with the other approaches discussed in §3.3. We also relate to the works that use similar ideas as ours.

$\mathcal{NRA}$, $\mathcal{NTA}$, and $\mathcal{NIA}$– Common Approaches

Interval Constraint Propagation

There are a few key insights that differentiate our approach from the interval constraint propagation approaches implemented in iSAT3 [FHT$^+$07], dReal [GKC13], and raSAT [TKO16]. First, our approach is based on linearization, it relies on solvers for $\mathcal{LA}$ and $\mathcal{UF}$, and it proceeds by incrementally axiomatizing the multiplication and transcendental functions. Compared to interval propagation, we avoid numerical approximations (even if within the bounds from DeltaSat). In a sense, the precision of the approximation is selectively detected at run time, while in iSAT3 and dReal this is a user defined threshold that is uniformly adopted in the computations. Second, our method relies on piecewise linear approximations, which can provide substantial advantages when approximating a slope – intuitively, interval propagation ends up computing a piecewise-constant approximation. Third, a distinguishing feature of our approach is the ability to (sometimes) prove the existence of a solution even if the actual values are irrationals, by reduction to an SMT-based validity check.

Linearization

Recent versions of the CVC4 [BCD$^+$11] SMT solver also implement a variant of the incremental linearization procedure. In comparison to our work, it does not support SMT($\mathcal{NTA}$). Moreover, it does not uses the $\mathcal{NRA}/\mathcal{NIA}$ model-finding heuristic, as described in §6.3.1. In another work [NPSS10], the idea of using tangent planes has been explored in the context of SMT($\mathcal{NRA}$). A key difference is that the tangent planes are used to under-approximate predicates, while in our approach they are used to refine the over-approximation of the multiplication function.

Interestingly, the subtropical satisfiability [FOSV17] implemented in veriT [BODF09] encodes a sufficient condition for satisfiability into an $\mathcal{LRA}$ problem, which is a similarity to our approach for checking satisfiability of $\mathcal{NRA}$ constraints. A difference is that our approach (though incomplete) can be used to detect both satisfiable and unsatisfiable cases.

In the context of SMT($\mathcal{NTA}$), the work in [Tiw15] approximates the natural logarithm ln with tangent lines, whereas we approximate not only a monotonic exp function (ln can be rewritten in terms exp) but we can also handle periodic trigonometric functions. Moreover, we also exploit other properties (e.g., monotonicity) to derive additional axioms.

In the context of SMT($\mathcal{NIA}$), our approach for finding $\mathcal{NIA}$-models is similar in spirit to the proposal in [BLO+12]: In a sense that we also under-approximate the problem by encoding into an SMT($\mathcal{LIA}$) problem via linearization. However, the approach in [BLO+12] focuses mainly on detecting satisfiable cases, whereas we try to handle both satisfiable and unsatisfiable instances. Another difference is that in [BLO+12] the linearization is done upfront if the problem is bounded; otherwise, it is done on-the-fly via some heuristics. In contrast, for finding models, we linearize using the guesses provided by the SMT($\mathcal{UFLIA}$) solver – solving a linear over-approximation of the original problem to detect unsatisfiability.

$\mathcal{NRA}$-Specific Approaches

z3 [dMB08b], Yices [Dut14], and SMT-RAT [CKJ+15] SMT solvers are based on quantifier elimination approaches. In contrast to these approaches, incremental linearization is an incomplete technique and is not based on quantifier elimination.

CalCS [NPSS10] SMT solver based on convex programming approach can handle a strict subset (convex constraints) of $\mathcal{NRA}$, whereas our approach does not have this restriction.

$\mathcal{NTA}$-Specific Approaches

MetiTarski [AP10], which is based on a deductive method, cannot prove the existence nor compute a satisfying assignment of a solution, while we are able to (sometimes) prove the existence of a solution even if the actual values are irrationals. An important point to note is that MetiTarski may require the user to manually write axioms if the ones automatically selected from a predefined library are not enough. Our approach is completely automatic.

Similarly to our work, a $\mathcal{UFLRA}$ approximation-based approach is presented in [EKK+11]. The approximation is however done only as a first check before calling the iSAT3 solver. In contrast, we rely on solvers for SMT($\mathcal{UFLRA}$), and we proceed by incrementally axiomatizing transcendental functions instead of calling directly an $\mathcal{NTA}$ solver. Another similarity with our work is the possibility of finding solutions in some cases. This is done by post-processing an inconclusive iSAT3 answer, trying to compute a certificate for a (point) solution for the narrow intervals returned by the solver, using an iterative analysis of the formula and of the computed intervals. Although similar in spirit, our technique for detecting satisfiable instances is completely different, being based on a logical encoding of the existence of a solution as an SMT($\mathcal{UFLRA}$) problem.

A similarity between the approaches presented in [dDLM11, MM16, Mel11, SH13, Mag14] and ours is the use of the Taylor polynomials. However, one distinguishing feature is that we use them to find lower and upper linear constraints by computing tangent and secant lines. Moreover, we do not rely on any floating point arithmetic library, and unlike the mentioned approaches, we can also prove the existence of a solution. On the other hand, some of the above tools employ more sophisticated/specialized approximations for transcendental functions, which might allow them to succeed in proving unsatisfiability of formulae for which our technique is not sufficiently precise.

Finally, since we are in the context of SMT, our approach also has the benefits of being:

1. fully automatic, unlike some of the above which are meant to be used within interactive theorem provers;

2. able to deal with formulae with an arbitrary Boolean structure, and not just conjunctions of inequalities; and

3. capable of handling combinations of theories (including uninterpreted functions, bit-vectors, arrays), which are beyond what the above, more specialized tools, can handle.

$\mathcal{NIA}$-Specific Approaches

In comparison to the bit-blasting approaches which either use a SAT solver [FGM$^+$07] or SMT($\mathcal{BV}$) [ZM10], our model finding method uses an SMT($\mathcal{UFLIA}$) solver. Moreover, the bit-blasting approaches mainly focus on finding models and have limited capabilities to detect unsatisfiable cases unless the input problems are bounded. As pointed out in [BLO$^+$12], they are not adequate for the satisfiable instances when the only solutions are the ones with large integer numbers. In contrast, incremental linearization is capable of handling both satisfiable and unsatisfiable instances – even for the unbounded problems.

Yices [Dut14] is the state of the art in SMT($\mathcal{NIA}$) – it solved the most number of problems in the QF-$\mathcal{NIA}$ division of the SMT competition 2017. As discussed in §3.3, Yices combines CAD (quantifier elimination method for $\mathcal{NRA}$) with the branch-and-bound method [Jov17]. In a sense, Yices uses an abstraction-refinement approach, where the abstract domain is $\mathcal{NRA}$, and the branch-and-bound constraints refine the abstraction. Our method is also an abstraction-refinement approach, but in contrast to Yices we use an SMT($\mathcal{UFLIA}$) for solving an over-approximation of the input problem and (incrementally) add linear constraints as refinements. In this way we can leverage on all state-of-the-art techniques for the integers reasoning implemented in SMT($\mathcal{UFLIA}$) solvers – methods include branch-and-bound, cutting planes, etc. (We suggest [Gri12] for more details).

Chapter 7

Implementation and Experimental Evaluation

The SMT procedures described in the previous chapter have been implemented within the MathSAT SMT solver [CGSS13]. The description of the procedures leaves some flexibility for different heuristics regarding refinement strategies and implementation choices. In this chapter, we describe the most important ones. We remark that these choices do not affect the soundness of the approach, but they can have an important impact on performance.

We have extended MathSAT with the procedures for $\mathcal{NRA}$, $\mathcal{NIA}$, and $\mathcal{NTA}$. The core solver only supports the exp and sin transcendental functions and nonlinear multiplications. Other functions (such as nonlinear division, square root, log, cos, tan, arcsin, arccos, arctan) are handled by encoding them in terms of the supported ones. For example, if the input formula φ contains $\sqrt{x}$, it is rewritten as $\varphi\{\sqrt{x} \mapsto y\} \wedge (y \geq 0 \rightarrow y * y = x)$ where y is a fresh variable. For formulae not involving transcendental functions, MathSAT can produce a model (in which all variables are assigned a rational value) when the formula is found to be satisfiable. Model generation is however not supported for transcendental functions. In this case, in principle it would be possible to produce rational bounds for the transcendental functions within which a model is guaranteed to exist (see §6.3.2), but this has not been implemented yet.

7.1 Implementation Details

Normalization

Normalization of $\mathcal{T}$-literals is an essential preprocessing step [Seb07] for the efficiency of SMT solvers. In this regard, during a call to to SMT-PREPROCESS, we normalize $\mathcal{NRA}$-literals by using the distributive property of multiplication on the nonlinear polynomials.

Arbitrary-precision Rationals

We use the GMP arbitrary-precision library to represent rational numbers. In our (model-driven) approach, we may have to deal with numbers with very large numerators and/or denominators. It may happen that we get such rational numbers from the bad model $\widehat{\mu}$ for the variables appearing as arguments of transcendental functions. As a result of the piecewise-linear refinement (e.g., for the instantiation of the tangent plane constraints), we may end-up feeding to the SMT($\mathcal{UFLRA}$) solver numbers that may have (even exponentially) larger numerators and/or denominators (due to the fact that GET-POLYNOMIAL-BOUNDS uses power series). This might significantly slow-down the performance of the solver.

We address this issue along the following dimensions:

1. *Continued fractions.* We approximate values in $\widehat{\mu}[x]$ having too large numerators and/or denominators by using continued fractions [NW88]. The precision of the rational approximation is increased periodically over the number of iterations. Thus, we delay the use of

numbers with larger numerator and/or denominator, and eventually find those numbers if they are really needed.

2. *Selective instantiation of tangent plane constraints.* While instantiating tangent plane constraints at the point (a, b) for $x * y$ we observe that, in order to block a model $\widehat{\mu}$ such that $\widehat{\mu}[f_*(x, y)] \neq \widehat{\mu}[x] * \widehat{\mu}[y]$, it is sufficient to add one of the two equalities of the tangent plane constraints in Fig. 6.4; instead of instantiating a constraint at (a, b), we can instantiate it at either $(a + \delta, b)$ or at $(a, b + \delta)$, for any value of δ. In practice, if a (resp. b) is a rational constant with a very large numerator or denominator, instead of instantiating one constraint at (a, b), we instantiate two tangent constraints at $(\lfloor a \rfloor, b)$ and $(\lceil a \rceil, b)$.

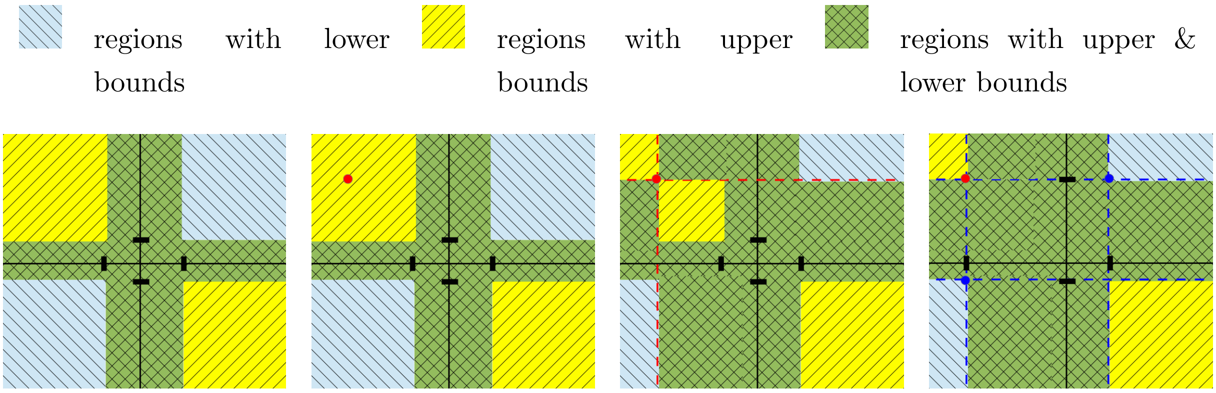
3. *Initial approximation for π.* We initialize the algorithm by choosing the values $l_\pi = \frac{333}{106}$ and $u_\pi = \frac{355}{113}$ since they give a very small difference $(\frac{1}{11978})$ with a limited number of digits.

Heuristics for Refinement

The descriptions of BLOCK-SPURIOUS-NRA-TERM and GET-POLYNOMIAL-BOUNDS leave some flexibility in deciding which constraints to add (and how many of them) at each iteration. It is possible to conceive strategies with an increasing degree of eagerness, from very lazy (e.g., adding only a single constraint per iteration) to more aggressive ones. The simple strategy we currently adopted consists in eagerly adding all the refinement constraints that are violated by the current abstract solution $\widehat{\mu}$, leaving the investigation of alternative strategies as future work.

Tangent Plane Frontiers for Multiplications

$x * y$ is a hyperbolic paraboloid surface, and a tangent plane to such surface cuts the surface into four regions: in two of them, the tangent plane


regions with lower bounds
regions with upper bounds
regions with upper & lower bounds

(a) current frontier

(b) new point (a, b)

(c) tangent plane instantiation at (a, b)

(d) additional instantiations and updated frontier

Figure 7.1: Illustration of the tangent lemma frontier strategy

is above the surface, thus providing an upper bound, whereas in the other two regions the tangent plane is below the surface (see Fig. 6.3). Each instantiation of the tangent plane constraints, therefore, only provides either a lower or an upper bound for a given region. Hence, there is a risk that the refinement procedure may go into an infinite loop, in which at each iteration a refined lower bound (respectively, upper bound) for a given point is added, when instead an upper bound (a lower bound, resp.) would be appropriate. In order to address the problem, we adopt the following strategy. For each $f_*(x, y)$ in the input formula, we maintain a *frontier* $\langle [l_x, u_x], [l_y, u_y] \rangle$ with the invariant that whenever $x \in [l_x, u_x]$ or $y \in [l_y, u_y]$, then $f_*(x, y)$ has both an upper and a lower bound in the abstract formula $\widehat{\varphi}$. Fig. 7.1 shows a graphical illustration of the strategy. Initially, the frontiers are set to $\langle [0, 0], [0, 0] \rangle$, corresponding to the "Zero" constraint of Fig. 6.4. Whenever a tangent plane constraint for $f_*(x, y)$ is instantiated at a point (a, b), we also add further instantiations of the constraint and update the frontier as follows:

case $a < l_x$ **and** $b < l_y$**:** add tangent planes at (a, u_y) and at (u_x, b), and set the frontier to $\langle [a, u_x], [b, u_y] \rangle$;

case $a < l_x$ **and** $b > u_y$**:** add tangent planes at (a, l_y) and at (u_x, b), and set the frontier to $\langle [a, u_x], [l_y, b] \rangle$;

case $a > u_x$ **and** $b > u_y$**:** add tangent planes at (a, l_y) and at (l_x, b), and set the frontier to $\langle [l_x, a], [l_y, b] \rangle$;

case $a > u_x$ **and** $b < l_y$**:** add tangent planes at (a, u_y) and at (l_x, b), and set the frontier to $\langle [l_x, a], [b, u_y] \rangle$.

7.2 Experimental Setup

We now experimentally evaluate the proposed approach for SMT. The experiments were run on a cluster of identical machines equipped with 2.6GHz Intel Xeon X5650 processors. The memory limit was set to 6 GB. We used 1000 seconds for the SMT experiments.[1] The results of the various solvers were automatically cross checked, and no discrepancies were reported.

We present the (most significant) data using tables, survival plots, and scatters plots. The tables present detailed results: each column represents a benchmark family, each entry shows the number of sat/unsat results reported, for tools reporting MaybeSat the number is shown in parentheses, the best performer for each family is highlighted in boldface, and the overall best is underlined. The survival plots compare the performance of multiple approaches. The x-axis shows the solving time in log-scale, and the y-axis shows the number of instances solved within the corresponding time. (Notice that we may use different scales on different plots.) The scatters plots compare pairwise solvers S_1 and S_2 on individual benchmarks: each point (t_1, t_2) in a scatters represents a benchmark problem that was solved in t_i time by solver S_i. We adopt a logarithmic scale, and report time out and

[1] The adopted time out for SMT is very close to the time out adopted in the SMT competition.

memory out as separate lines. Red dots are for satisfiable instances, and blue dots for unsatisfiable ones. Green dots indicate unknown instances. Diagonal lines mark 2x and 10x performance differences. Points on the inner edges indicate timeouts, those on the outer edges indicate other errors (memory outs or aborts). In the comparison, by VirtualBest we mean the results of a virtual portfolio solver that performs on each benchmark as the best of the solvers in the portfolio.

Benchmarks

For the experimental evaluation on SMT we selected the following benchmarks. For $\mathcal{NRA}$ and $\mathcal{NIA}$, we used all the SMT-LIB [BFT16] benchmarks from the QF-$\mathcal{NRA}$ and QF-$\mathcal{NIA}$ categories, respectively. The QF-$\mathcal{NRA}$ class contains 11354 benchmarks, among which 4963 are satisfiable, 5296 are unsatisfiable, and 1095 have unknown status. In the QF-$\mathcal{NIA}$ category, there are 23876 benchmarks: 14124 satisfiable, 3130 unsatisfiable, and 6622 with unknown status. Since SMT($\mathcal{NTA}$) is not standardized in SMT-LIB, for $\mathcal{NTA}$ we adopted an extended version of SMT-LIB including special function symbols for sin, exp and π. We collected and encoded SMT($\mathcal{NTA}$) benchmarks from the following sources, for a total of 2512 benchmarks:

- verification queries over transcendental transition systems [MSN$^+$16] deriving from SMT-based verification engines, including discretization of Bounded Model Checking of hybrid automata [RPV17, BBJ15];
- all the benchmarks from the MetiTarski distribution [AP10];
- all the SMT($\mathcal{NTA}$) benchmarks [2] from the dReal distribution [GKC13];

[2] dReal is also able to deal with Ordinary Differential Equations.

- all the benchmarks from the iSAT3 distribution [MSN+16].

iSAT3 requires that each variable is constrained to some interval. Therefore, we used a very large bound[3] (allowed by iSAT3) for each variable in the $\mathcal{NRA}$ benchmarks. However, it also requires the intervals to be small when there are transcendental functions. In this case we generated scaled-down versions of the $\mathcal{NTA}$ benchmarks, by adding bound constraints that force all the real variables in the problem to assume values in the $[-300, 300]$ interval. Since the SMT-LIB format is not accepted by iSAT3 and MetiTarski, we wrote scripts to cross-convert the benchmarks in the respective formats.

Other Approaches

We compared MathSAT with various SMT solvers. For SMT($\mathcal{NRA}$), we considered z3 [dMB08b, JdM12], Yices [Dut14], and SMT-RAT [CKJ+15], which implement expensive and complete techniques based on variants of CAD, and iSAT3 and dReal, based on interval constraint propagation. We also considered the recent version of CVC4 [BCD+11] that, as discussed in [RTJB17], is an independent implementation of our incremental linearization approach as described in [CGI+17a]. For SMT($\mathcal{NTA}$), we considered iSAT3, dReal and MetiTarski [AP10], that implements a deductive approach. MetiTarski has been developed to deal with hard $\mathcal{NTA}$ problems with a few dozens of predicates, and it is not particularly suited for problems with a complicated Boolean structure.

[3] $[-10000000000000000000000, 10000000000000000000000]$

	Total (11354)	**Total w/o MetiTarski (4348)**
MathSAT	3514/**5338**	400/3045
CVC4	3050/5322	315/**3068**
z3	**4883**/5039	**492**/2428
Yices	4817/5057	449/2480
SMT-RAT	4317/4475	133/1957
dReal	0(5396)/4239	0(516)/2154
iSAT3	2404(2517)/4402	10(667)/2486
VirtualBest	5158/5756	767/3141

	MetiTarski (7006)	**Heizmann (69)**	**Hong (20)**	**HyComp (2752)**	**Kissing (45)**	**LassoRanker (821)**	**Sturm-MBO (405)**	**Sturm-MGC (9)**	**UltimateAutomizer (61)**	**Zankl (166)**
MathSAT	3114/2293	**3**/1	0/**20**	17/2267	18/0	**299**/432	0/**285**	0/0	32/6	31/**34**
CVC4	2735/2254	0/2	0/**20**	9/2253	4/0	298/**465**	0/**285**	0/2	4/8	0/33
z3	**4391**/**2611**	0/0	0/9	**243**/2210	**33**/0	105/115	0/47	**2**/**7**	**47**/**13**	**62**/27
Yices	4368/2577	0/**11**	0/8	219/2182	10/0	120/233	0/2	0/0	40/12	60/32
SMT-RAT	4184/2518	0/0	0/**20**	98/1632	9/0	0/0	0/**285**	0/1	2/0	24/19
dReal	0(4880)/2085	0(0)/0	0(0)/**20**	0(351)/2129	0(17)/**1**	0(0)/0	0(0)/0	0(0)/0	0(45)/0	0(103)/4
iSAT3	2394(1850)/1916	0(15)/0	0(0)/14	0(393)/**2276**	1(29)/**1**	0(121)/27	0(7)/148	0(3)/0	0(27)/2	9(72)/18
VirtualBest	4391/2615	3/12	0/20	299/2303	33/1	310/466	0/285	2/7	48/13	72/34

Table 7.1: Summary of SMT($\mathcal{NRA}$) experimental results

7.3 Results

Results for SMT($\mathcal{NRA}$)

The results for SMT($\mathcal{NRA}$) are reported in Table 7.1.

The MetiTarski benchmark class in SMT-LIB contains proof obligations deriving from MetiTarski executions. The MetiTarski benchmarks are about two thirds of the SMT($\mathcal{NRA}$) benchmarks in SMT-LIB, and have a very specific structure – they are conjunctions with no Boolean part, so that the SMT solvers are in fact being activated as $\mathcal{T}$-solvers for NRA. Given that MathSAT is also able to deal with the original problems in SMT($\mathcal{NTA}$), we also report the results of the comparison limited to the other benchmarks.

Table 7.1 demonstrates several interesting trends. First, on satisfiable instances, the complete techniques of Yices and z3 have superior performances than MathSAT in the overall case. However, if we exclude the MetiTarski benchmarks, we obtain comparable performance to complete solvers, and substantial advantage over incomplete solvers. We also notice that dReal is unable to conclude Sat in any of the benchmarks, and – regardless of the precision adopted – it claims MaybeSat on 583 unsatisfiable benchmarks.

Incremental linearization shines on unsatisfiable benchmarks. MathSAT (and also CVC4, that basically implements the same technique) demonstrates very good performance. Overall, MathSAT solves 8852 benchmarks (behind z3 with 9922 and Yices 9874), and is the strongest solver with 3445 solved (followed by CVC4 with 3383) on the benchmarks without MetiTarski problems. Interestingly, incremental linearization is highly complementary with respect to the more expensive techniques implemented in z3, Yices and SMT-RAT. MathSAT is able to solve 317 benchmarks that cannot be solved by other solvers (with the exception of

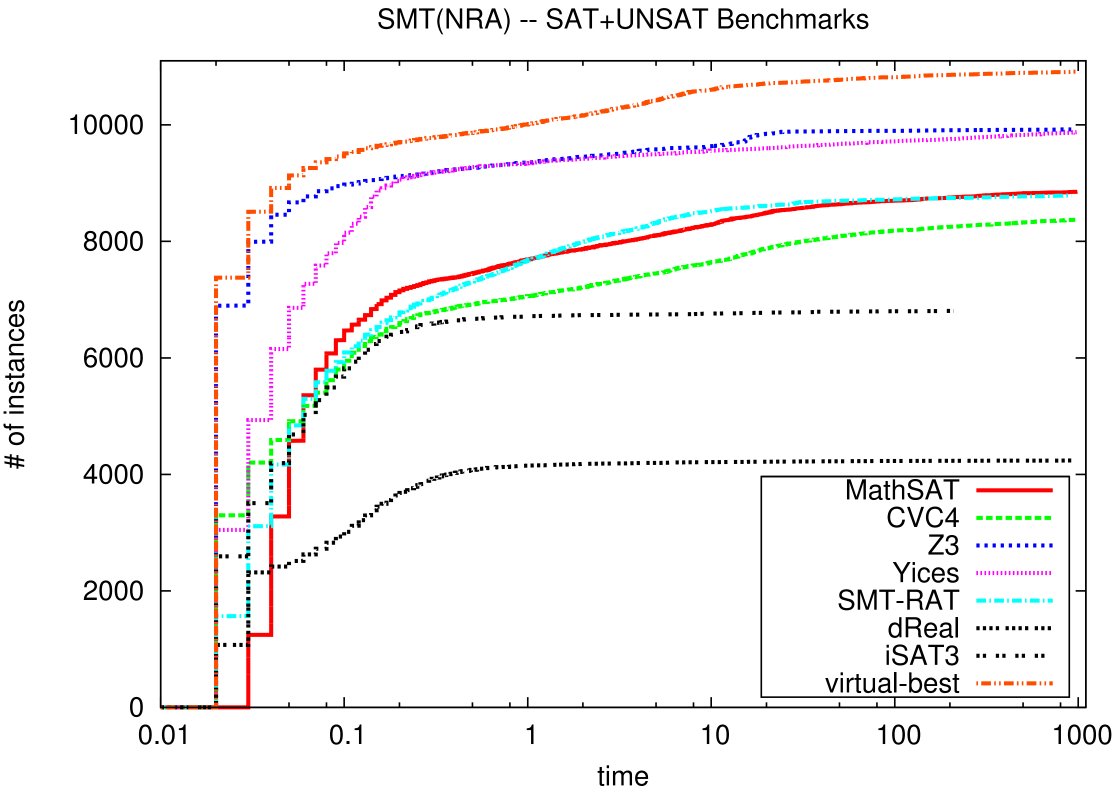
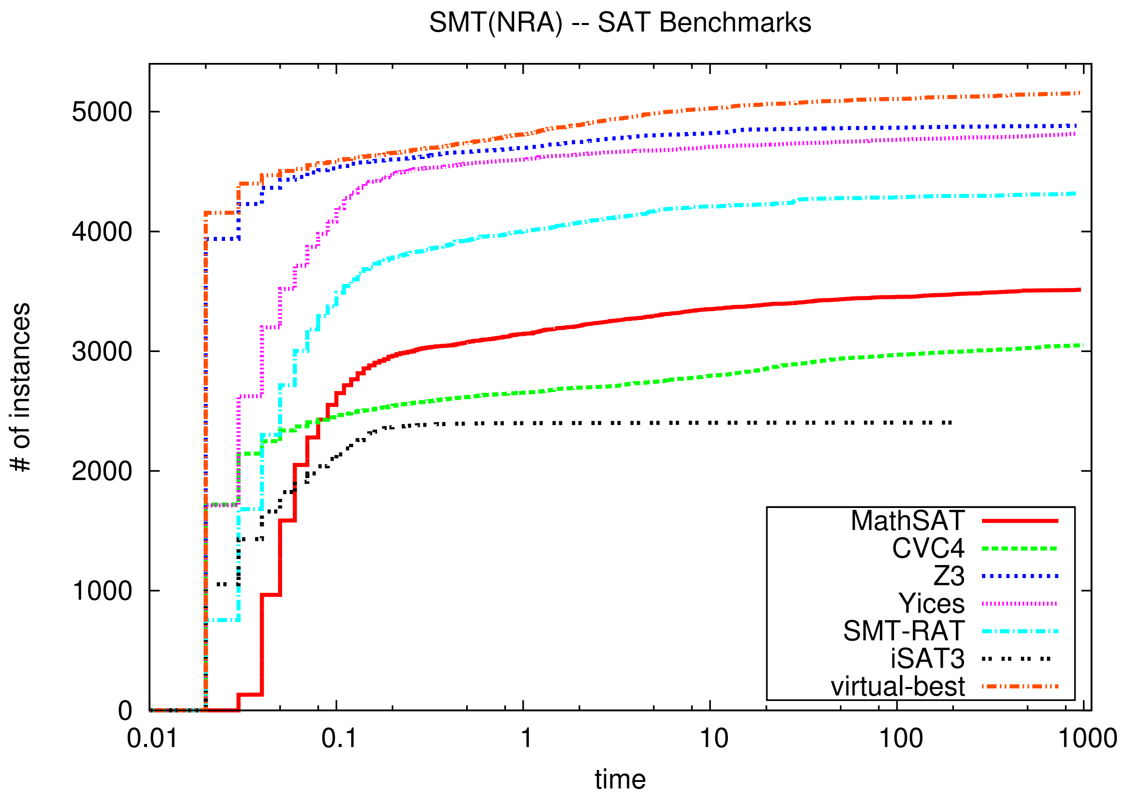


Figure 7.2: Survival plots for SMT($\mathcal{NRA}$) benchmarks

CVC4), and it has been able to prove 579 benchmarks that have unknown status in the SMT-LIB.[4]

The scatters plots for all SMT($\mathcal{NRA}$) benchmarks are reported in Fig. 7.3; in Fig. 7.5 are reported the results excluding the MetiTarski benchmarks. The diagrams comparing MathSAT with z3, Yices and SMT-RAT show that the techniques to detect satisfiable instances are very complementary. The diagrams comparing MathSAT with iSAT3 and dReal show that the interval-based techniques are not good at detecting satisfiable instances. We also notice that incremental linearization and interval-based solvers are complementary on unsatisfiable instances. This suggests that integrating interval propagation within incremental linearization may be sometimes beneficial for efficiency.

[4] This value refers to the tagging of the SMT-LIB on 2017-06-17.



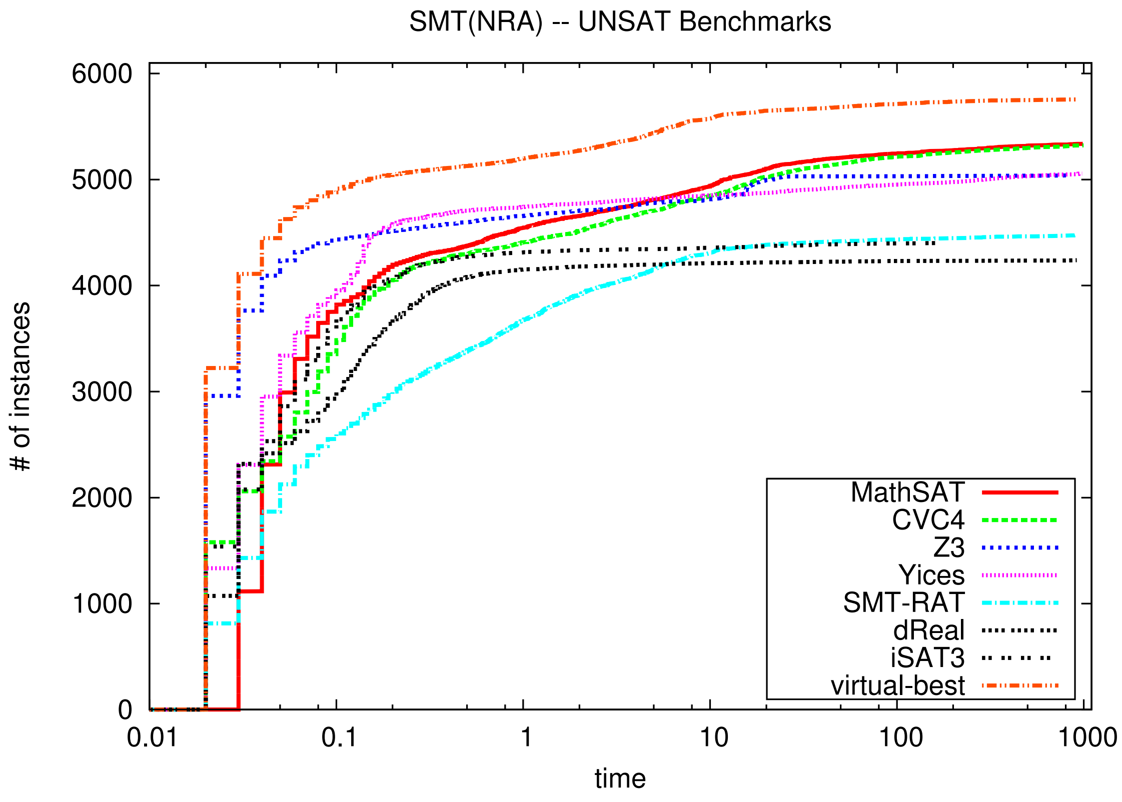
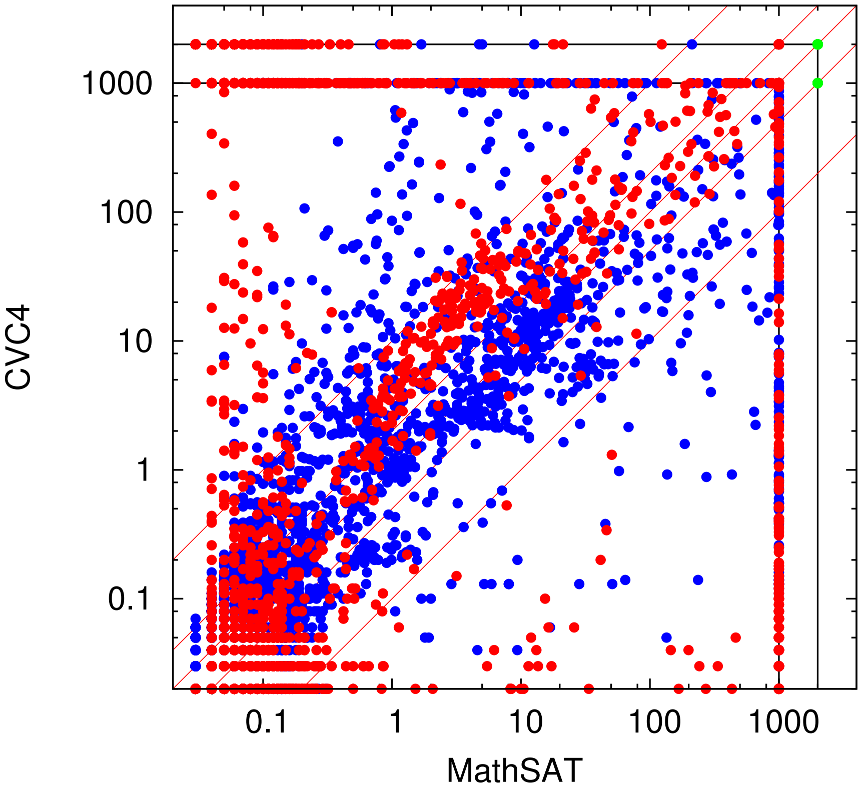
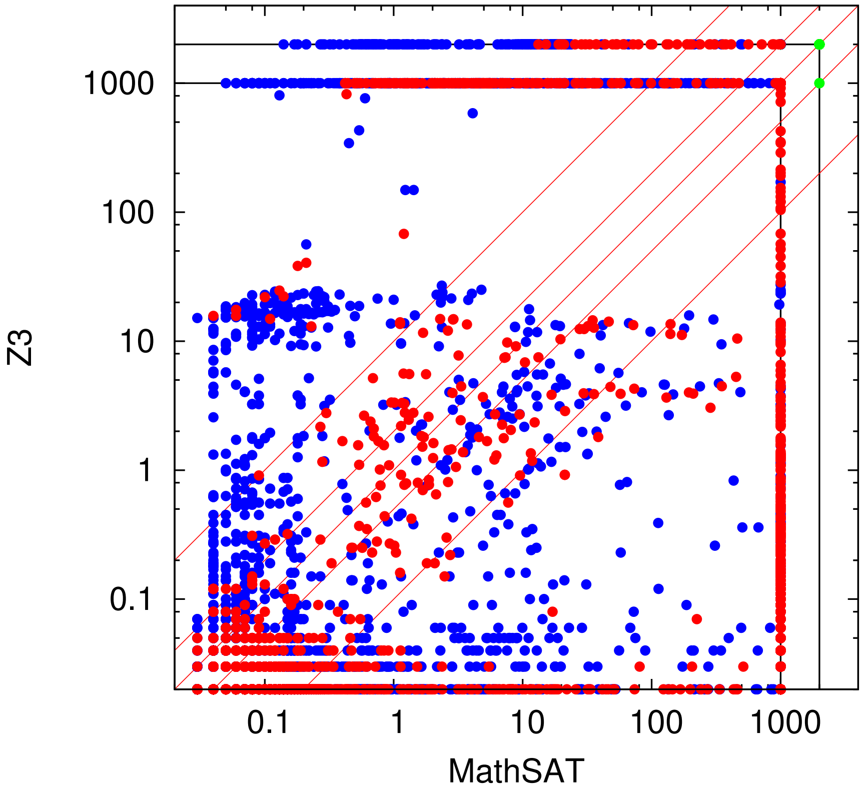


Figure 7.2: Survival plots for SMT($\mathcal{NRA}$) benchmarks

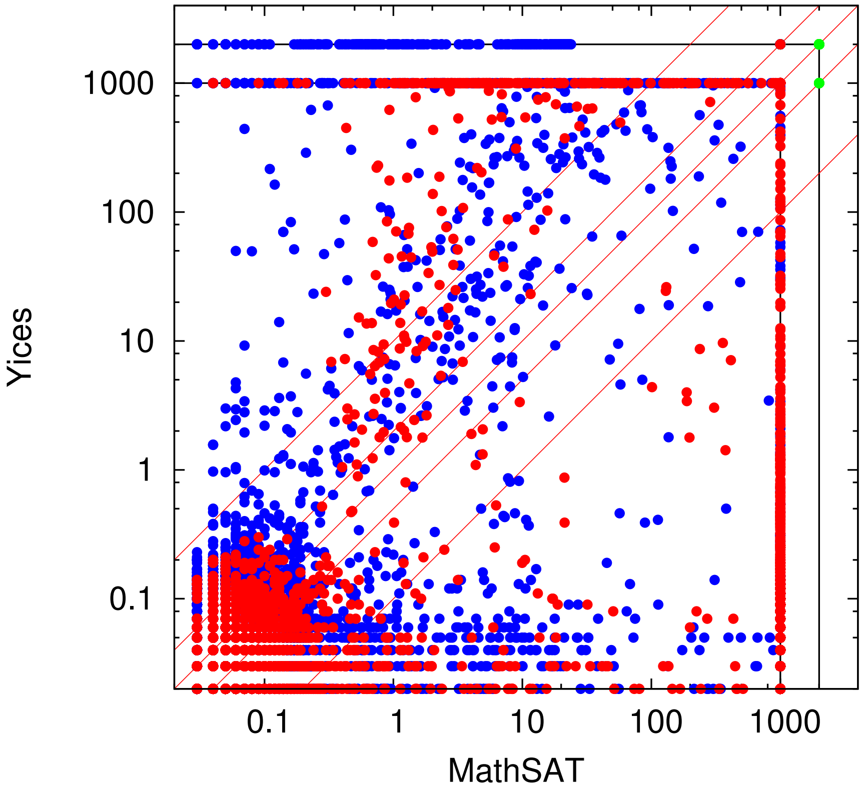


MathSAT solves 805 benchmarks that are not solved by CVC4, and 325 vice versa.

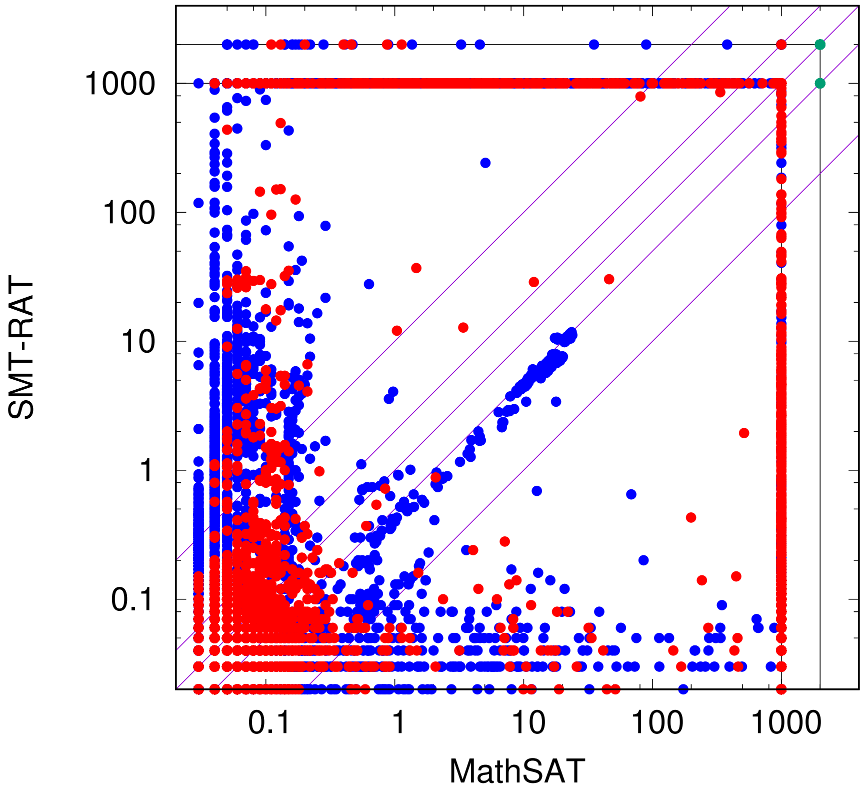


MathSAT solves 849 benchmarks that are not solved by z3, and 1919 vice versa.

Figure 7.3: Scatters plots for SMT($\mathcal{NRA}$) benchmarks

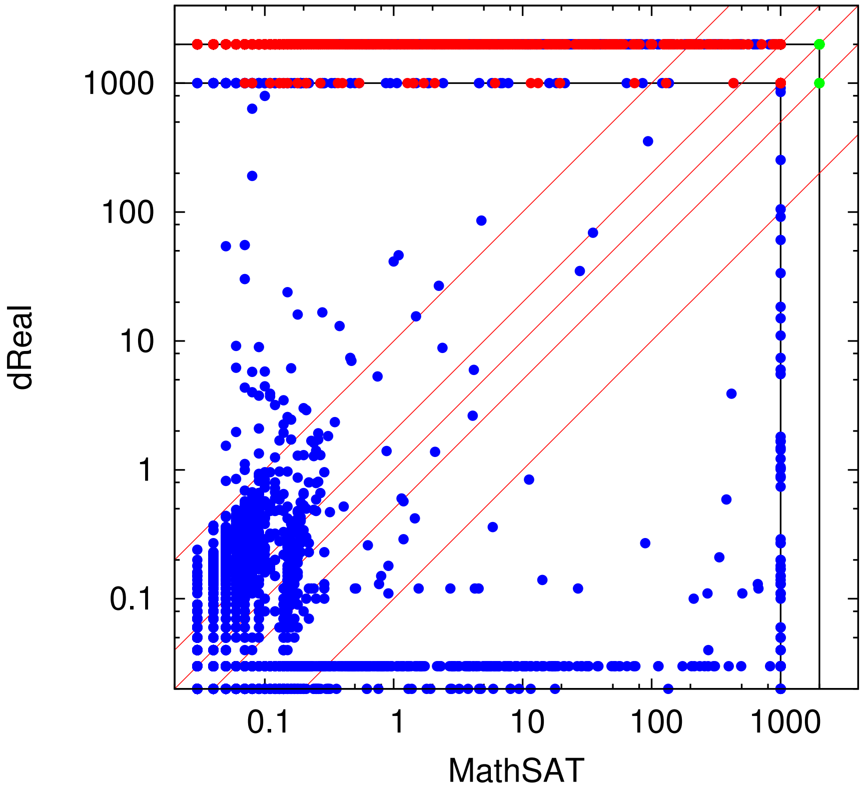


MathSAT solves 817 benchmarks that are not solved by Yices, and 1839 vice versa.

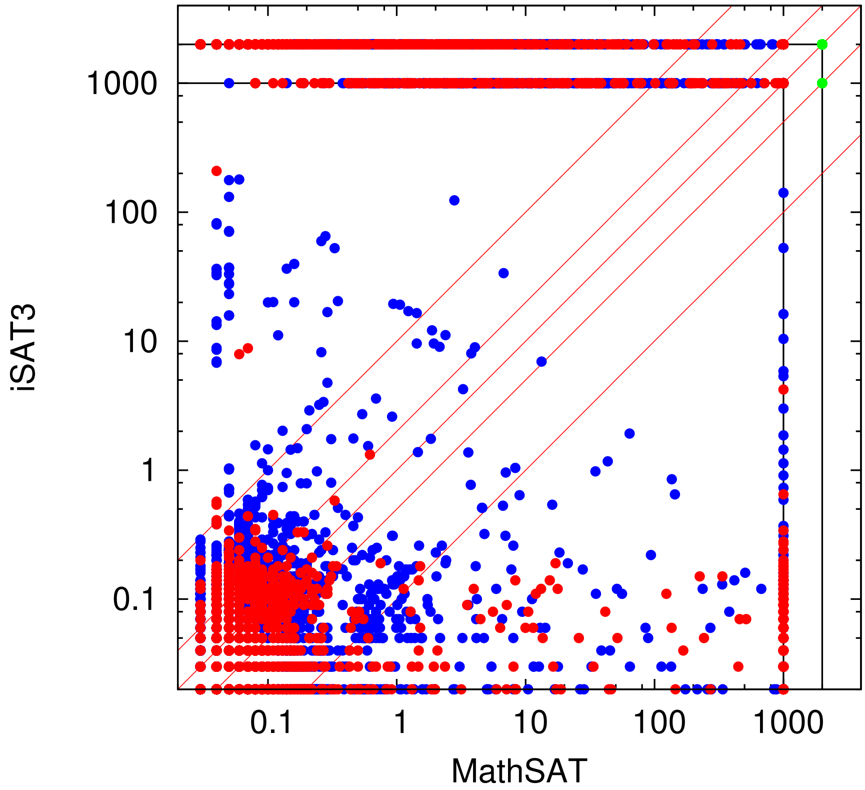


MathSAT solves 1567 benchmarks that are not solved by SMT-RAT, and 1516 vice versa.

Figure 7.3: Scatters plots for SMT($\mathcal{NRA}$) benchmarks



MathSAT solves 4714 benchmarks that are not solved by dReal, and 101 vice versa.



MathSAT solves 2360 benchmarks that are not solved by iSAT3, and 314 vice versa.

Figure 7.3: Scatters plots for SMT($\mathcal{NRA}$) benchmarks

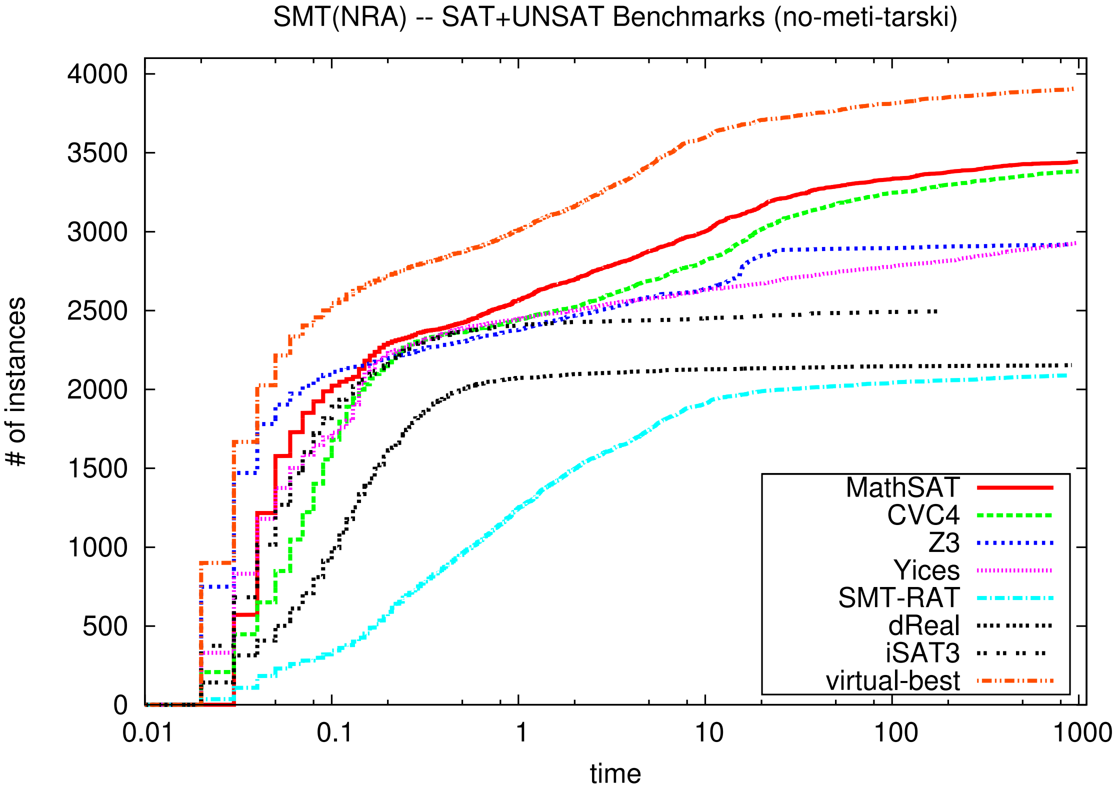


Figure 7.4: Survival plots for SMT($\mathcal{NRA}$) benchmarks excluding MetiTarski

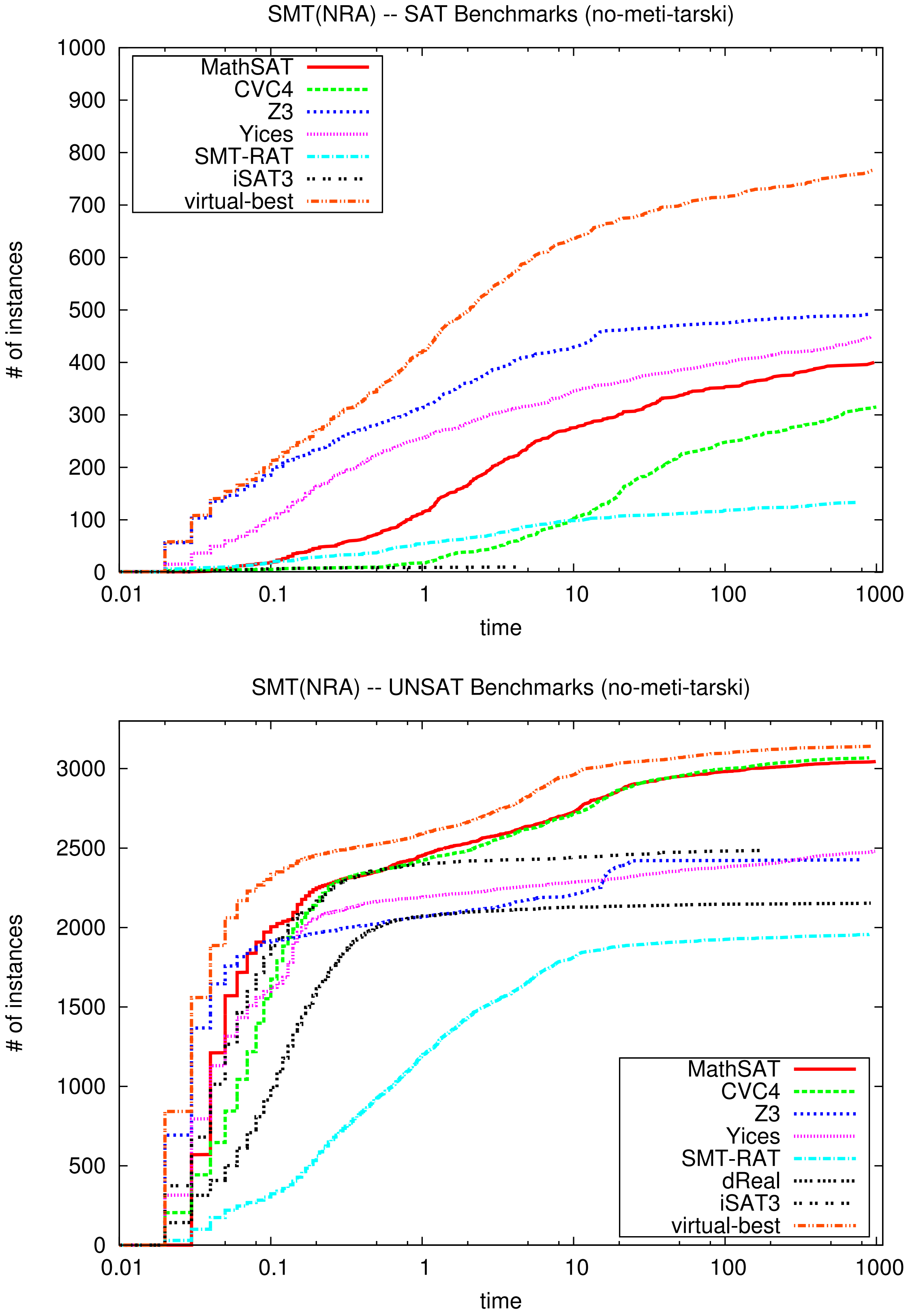
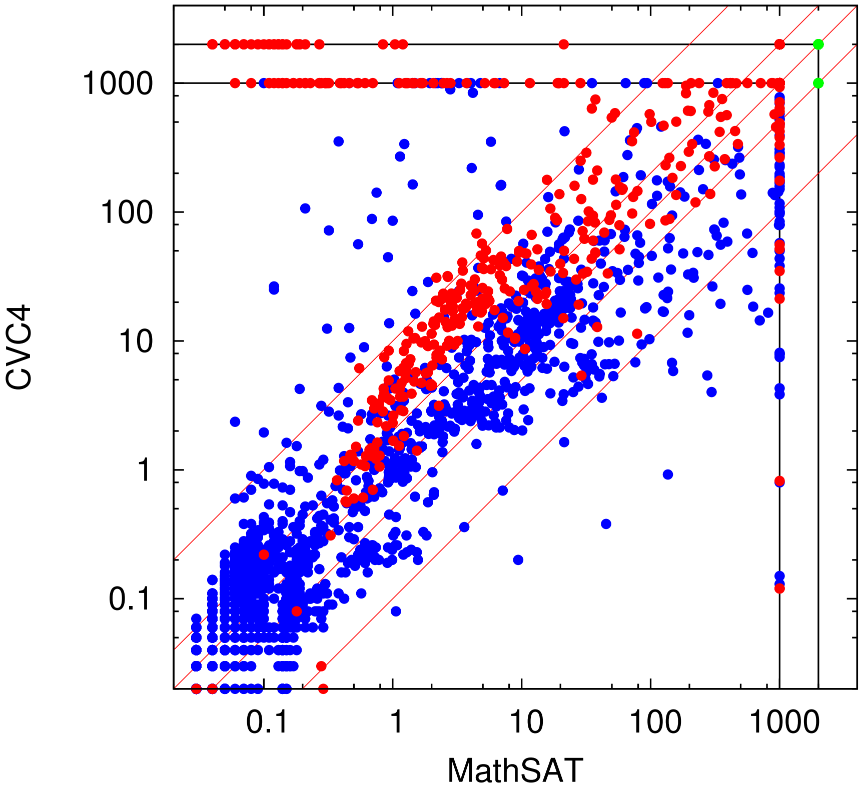
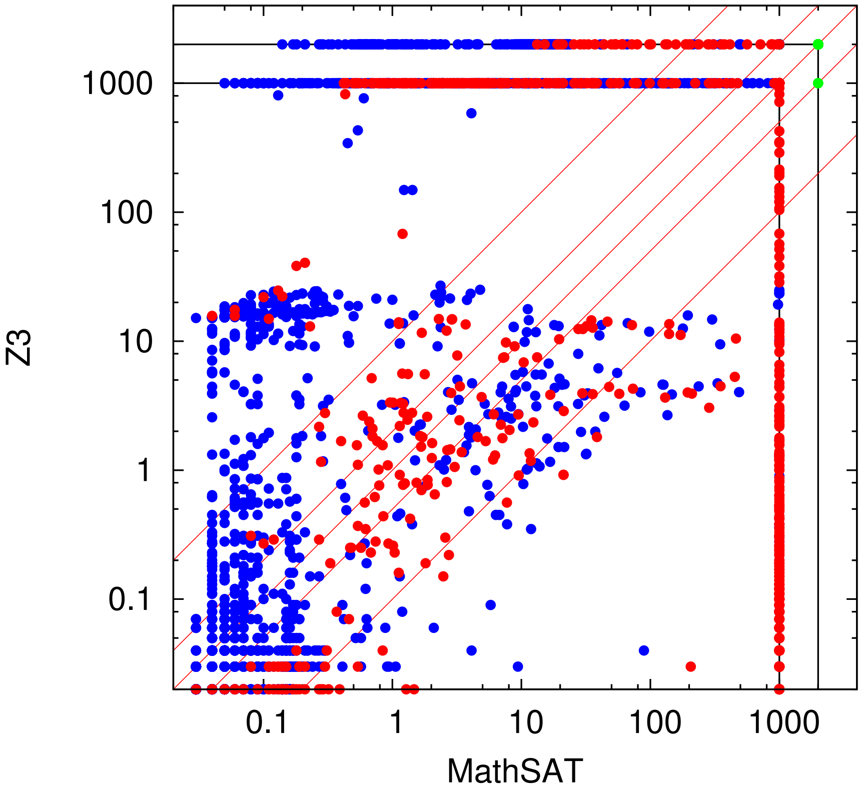


Figure 7.4: Survival plots for SMT($\mathcal{NRA}$) benchmarks excluding MetiTarski

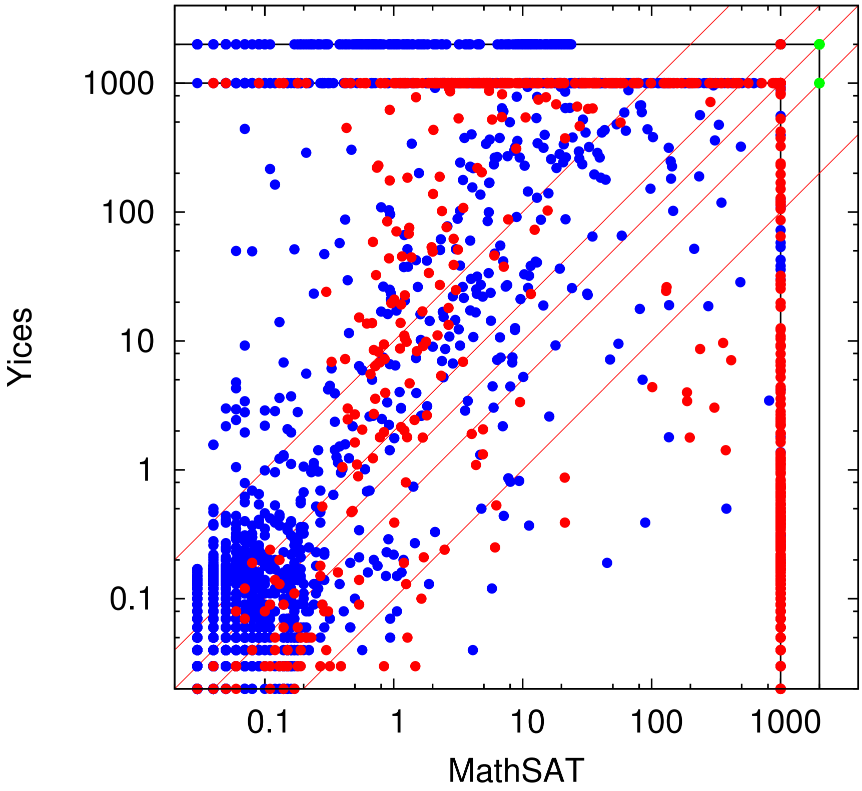


MathSAT solves 127 benchmarks that are not solved by CVC4, and 65 vice versa.

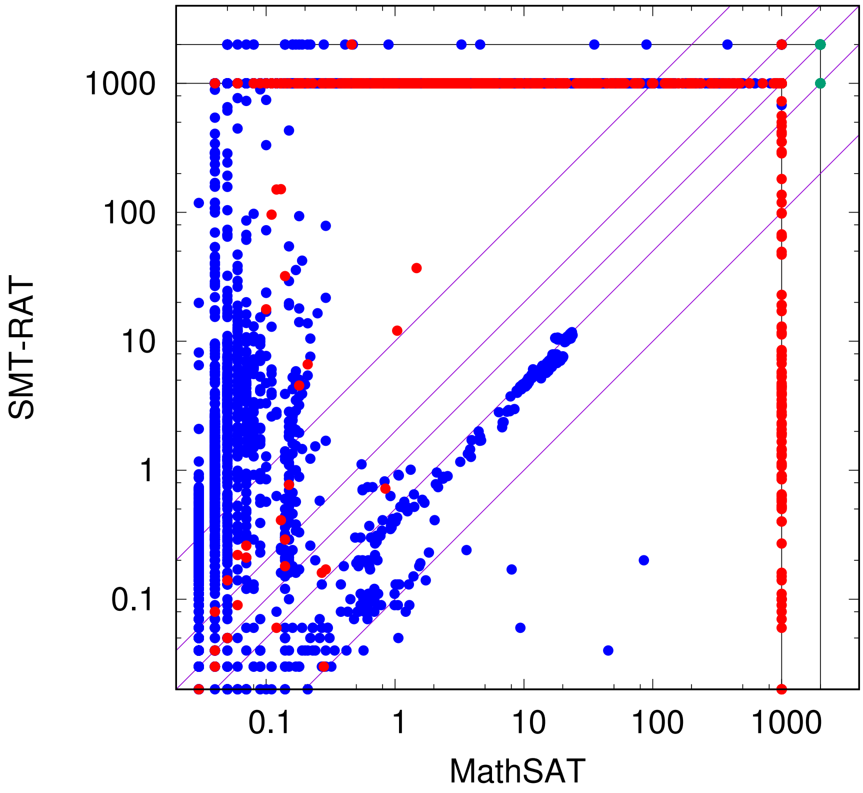


MathSAT solves 849 benchmarks that are not solved by z3, and 324 vice versa.

Figure 7.5: Scatters plots for SMT($\mathcal{NRA}$) benchmarks excluding MetiTarski

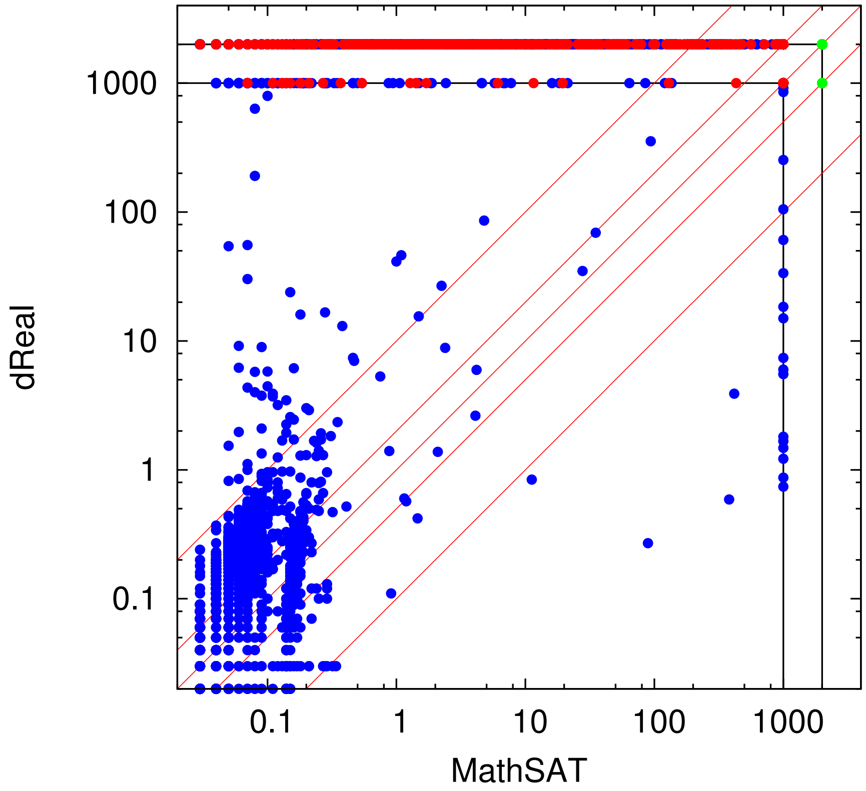


MathSAT solves 798 benchmarks that are not solved by Yices, and 282 vice versa.

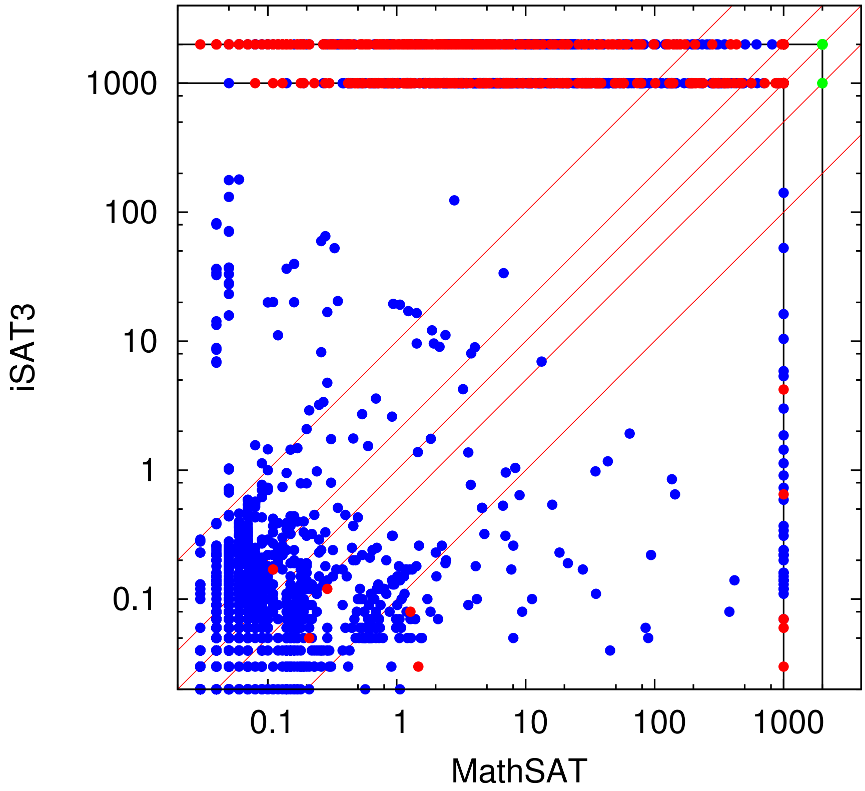


MathSAT solves 1462 benchmarks that are solved by SMT-RAT, and 107 vice versa.

Figure 7.5: Scatters plots for SMT($\mathcal{NRA}$) benchmarks excluding MetiTarski



MathSAT solves 1308 benchmarks that are not solved by dReal, 17 vice versa.



MathSAT solves 986 benchmarks that are not solved by iSAT3, and 37 vice versa.

Figure 7.5: Scatters plots for SMT($\mathcal{NRA}$) benchmarks excluding MetiTarski

Results for SMT($\mathcal{NTA}$)

The results on the SMT($\mathcal{NTA}$) benchmarks are reported in Table 7.2. We can see that MathSAT is able to solve more benchmarks than dReal and iSAT3. MetiTarski is unable to deal with benchmarks involving Boolean combinations, and thus could not be run on the BMC and dReal benchmarks. Similarly to the case of SMT($\mathcal{NRA}$), dReal is able to solve only unsatisfiable benchmarks, and returns MaybeSat in many situations that are in fact unsatisfiable. If we consider the scaled-down case, iSAT3 demonstrates better performance than dReal on BMC benchmarks, being able to prove more satisfiable benchmarks. Overall, however, it is still behind MathSAT. The scatters plots are reported in Fig. 7.7. The comparison between MathSAT and MetiTarski only considers the MetiTarski benchmarks. Despite MetiTarski being superior to MathSAT in terms of solved instances, we notice that MathSAT can be at times much faster than MetiTarski, and that it is not strictly dominated – MetiTarski runs out of memory in many benchmarks that MathSAT can solve. In fact, there is a substantial difference between the number of benchmarks solved by MetiTarski (536) and the virtual best solver (621). The scatters in Fig. 7.7 also confirm the complementarity between incremental linearization and interval constraint propagation solvers. It is possible to notice a stripe of points that are solved in nearly constant time by dReal and iSAT3 that are increasingly harder for MathSAT. On the bounded benchmarks, dReal and iSAT3 contribute with almost 200 instances solved to the virtual best (see Table 7.2). This confirms the potential benefits of integrating interval propagation within incremental linearization.

	Total **(2512)**	**BMC** **(887)**	**MetiTarski** **(681)**	**dReal** **(944)**
MathSAT	**58/1198**	**44/541**	0/299	**14/358**
dReal	0(1177)/761	0(294)/392	0(368)/267	0(528)/102
MetiTarski	0/536	-	0/**536**	-
VirtualBest	58/1588	44/546	0/621	14/421
MathSAT	**50/1139**	**39/547**	0/**313**	**11/279**
iSAT3	43(1316)/733	36(270)/475	0(455)/195	7(591)/63
dReal	0(1056)/782	0(267)/403	0(293)/269	0(496)/110
VirtualBest	70/1317	53/565	0/406	17/346

Table 7.2: Summary of SMT($\mathcal{NTA}$) experimental results (original problems above, bounded version below)

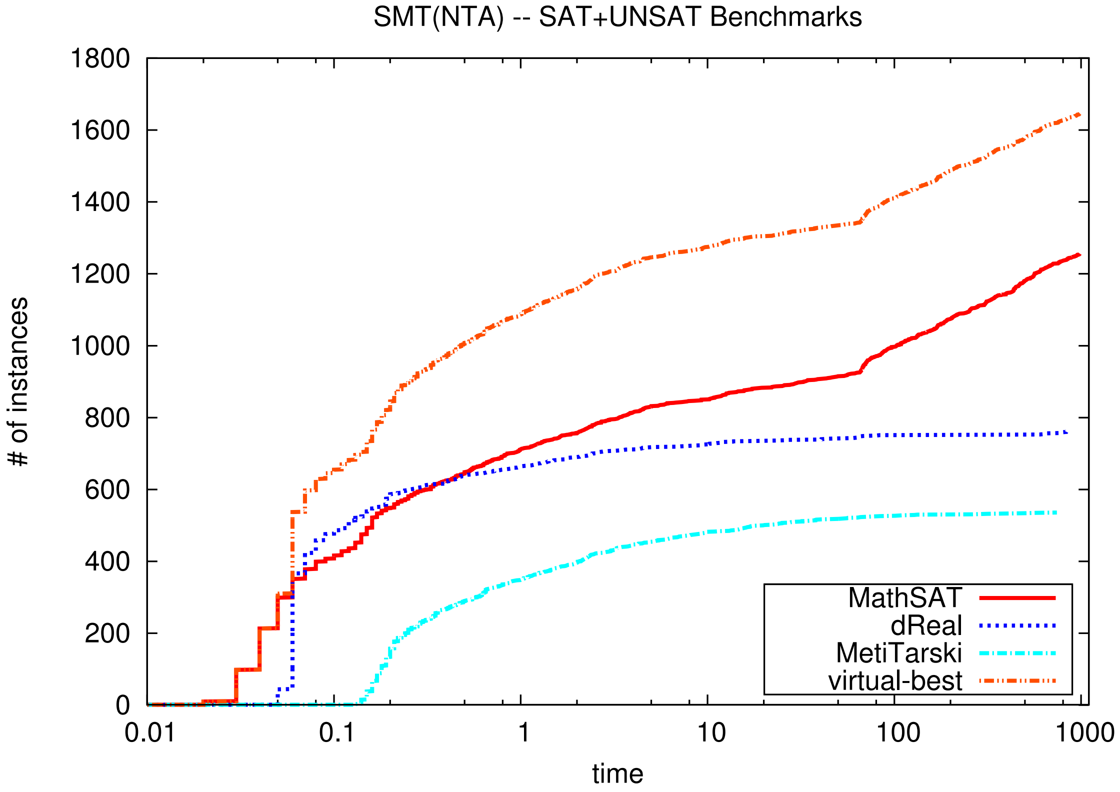
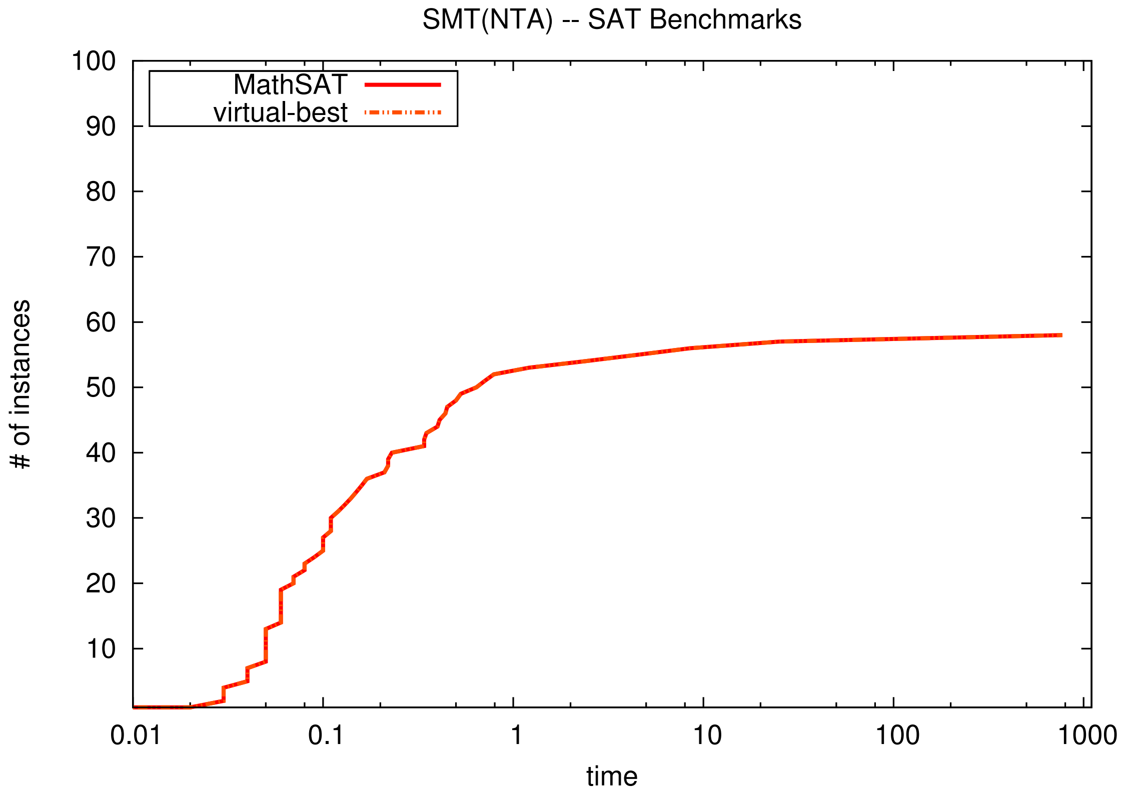


Figure 7.6: Survival plots for SMT($\mathcal{NTA}$) – unbounded benchmarks



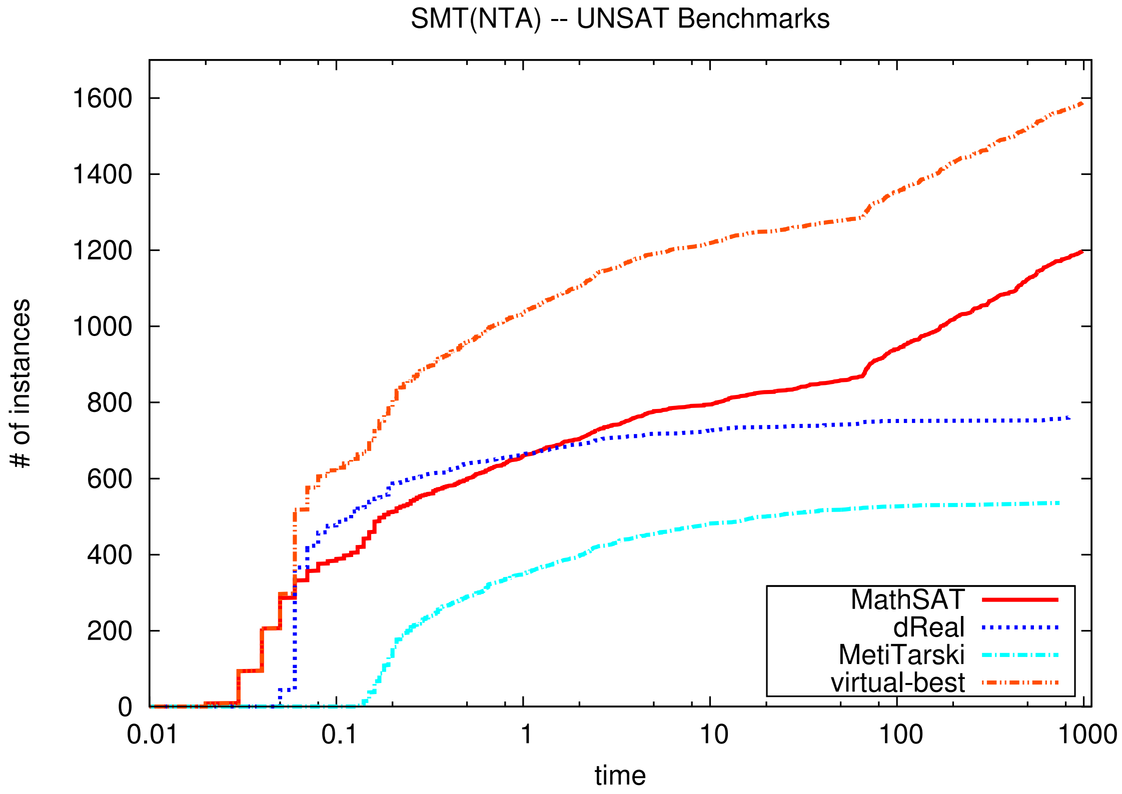
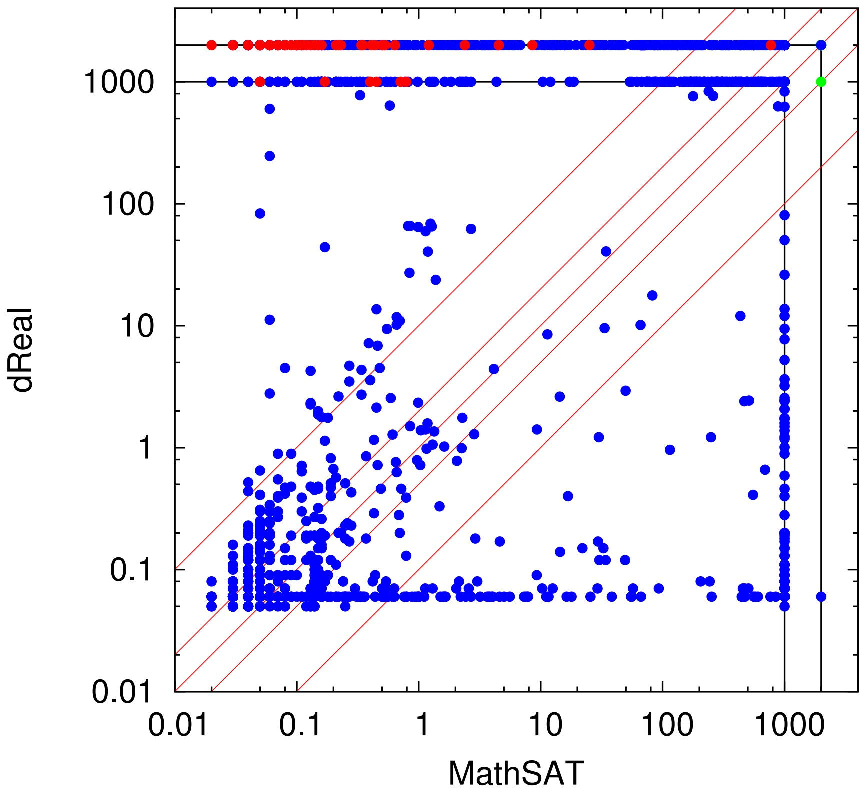
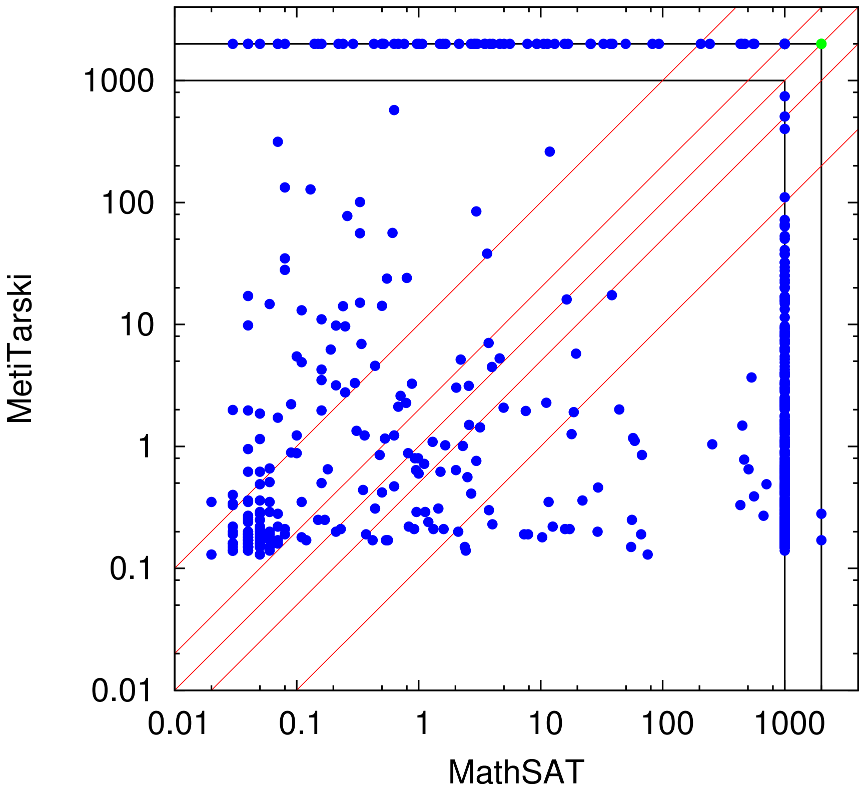


Figure 7.6: Survival plots for SMT($\mathcal{NTA}$) – unbounded benchmarks



MathSAT solves 636 benchmarks that are not solved by dReal, and 141 vice versa.



MathSAT solves 1018 benchmarks that are not solved by MetiTarski, and 298 vice versa.

Figure 7.7: Scatters plots for SMT($\mathcal{NTA}$) – unbounded benchmarks

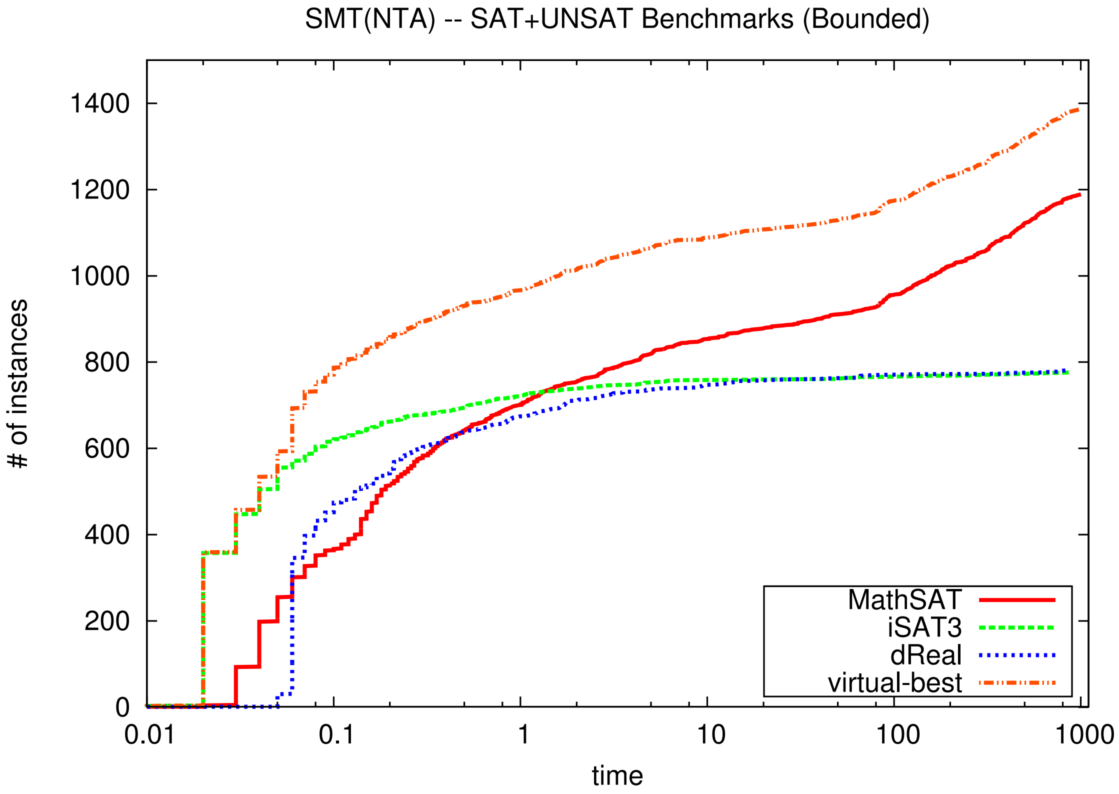


Figure 7.8: Survival plots for SMT($\mathcal{NTA}$) – bounded benchmarks

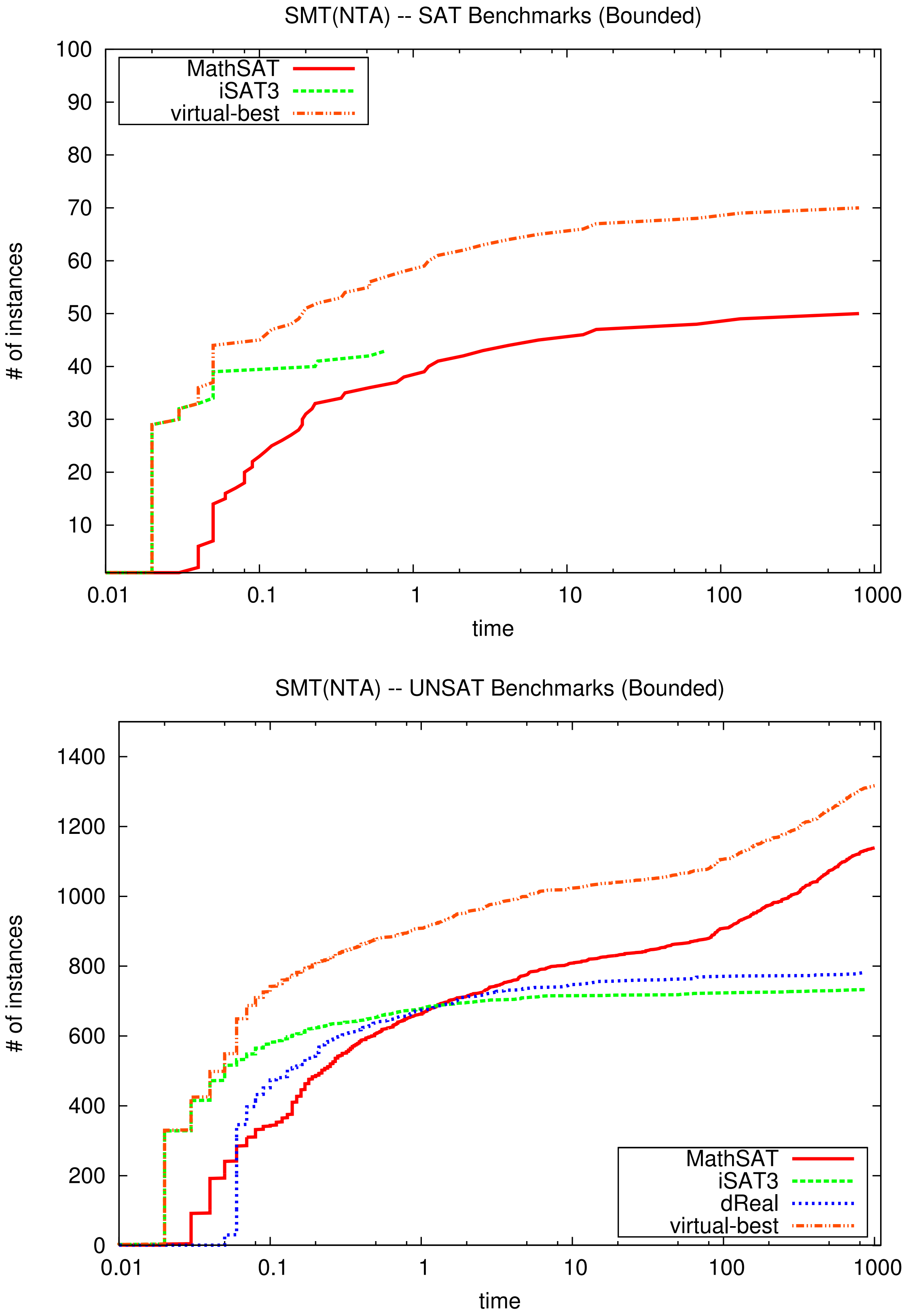
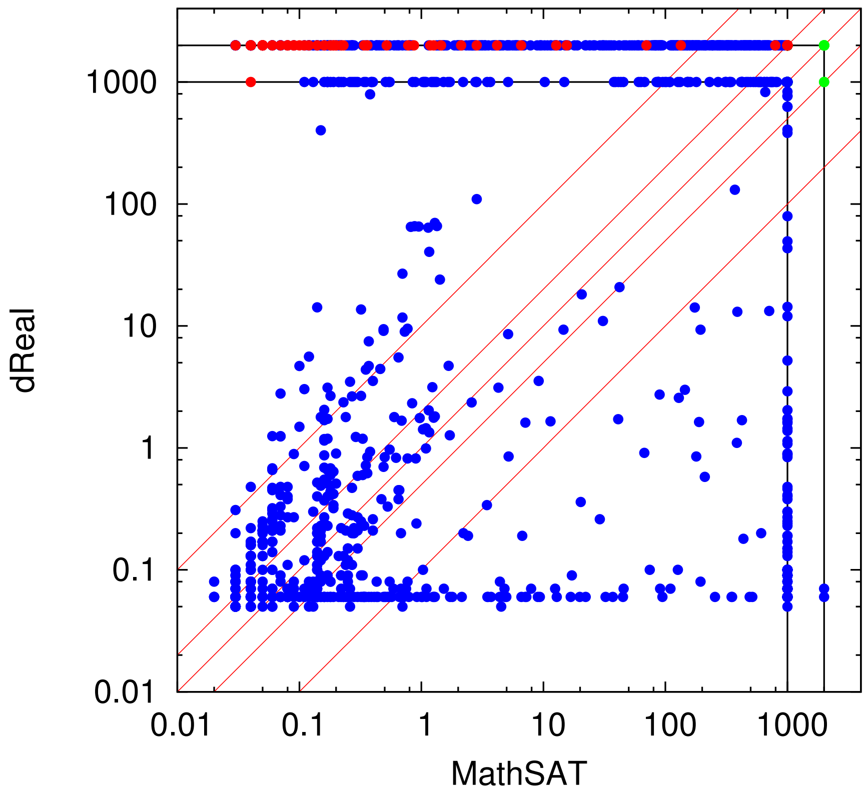
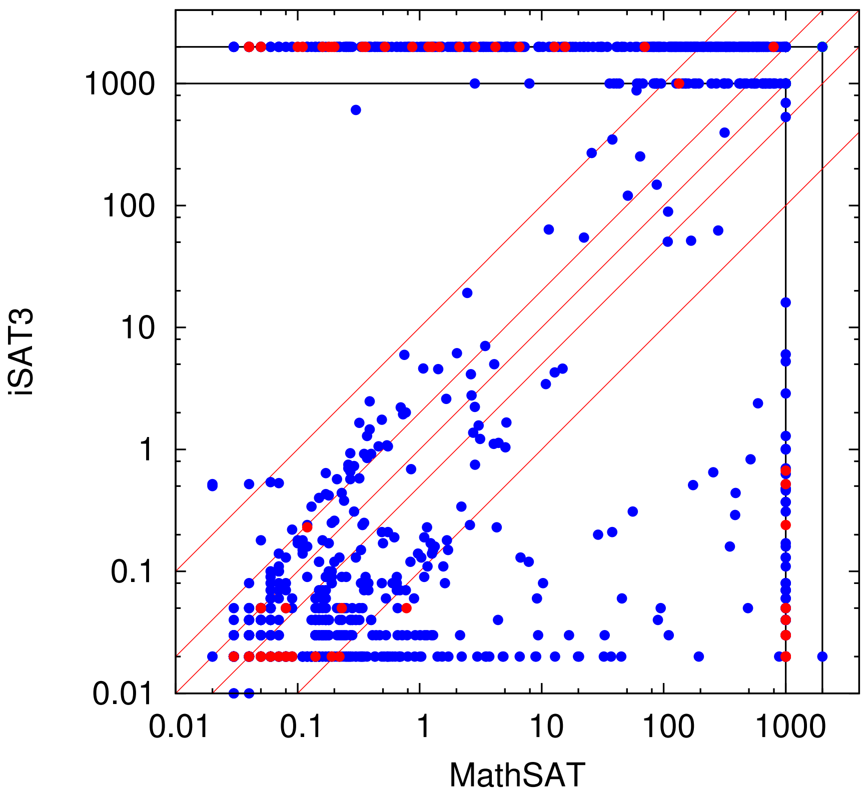


Figure 7.8: Survival plots for SMT($\mathcal{NTA}$) – bounded benchmarks



MathSAT solves 559 benchmarks that are not solved by dReal, and 152 vice versa.



MathSAT solves 506 benchmarks that are not solved by iSAT3, and 93 vice versa.

Figure 7.9: Scatters plots for SMT($\mathcal{NTA}$) – bounded benchmarks

Results for SMT($\mathcal{NIA}$)

The results of the SMT($\mathcal{NIA}$) benchmarks are reported in Table 7.3. MathSAT solves the highest number of satisfiable and unsatisfiable instances. It solves a total of 16716 problems; the closest to MathSAT is Yices which solves 15785 problems. z3, SMT-RAT, and CVC4 are able to solve less than 10000 benchmarks. Notice that Yices implements a combination of expensive $\mathcal{NRA}$ quantifier elimination (CAD) and branch-and-bound technique, while z3 and SMT-RAT rely on bit-blasting, and CVC4 is using a variant of incremental linearization. The difference between MathSAT and VirtualBest is around 2000 instances; that suggests some complementarity among the approaches. It is also evident from the survival and scatters plots, shown in Fig. 7.10 and Fig. 7.11 [5], respectively. The plots also show an interesting behavior: Despite Yices being able to solve fewer benchmarks than MathSAT, it is faster than MathSAT on some benchmarks. (This scenario can also be observed to a less extent when comparing MathSAT against the bit-blasting approaches.) This observation is another point in favor of complementarity among the approaches.

We notice that the simple model-finding technique (discussed in §6.3) is surprisingly useful on the $\mathcal{NIA}$ benchmarks. In fact, almost 60% of the benchmarks are known to be satisfiable – MathSAT solves 83% of them.

We notice a difference between the performance of MathSAT and CVC4. As mentioned earlier, CVC4 also implements a variant of incremental linearization, the difference between the solved satisfiable instances is similar to the trend we saw in the $\mathcal{NRA}$ benchmarks result, since CVC4 does not use a separate satisfiability detecting heuristic similar to what MathSAT implements. However, the difference in the solved

[5] For better readability, we also show separate scatters plots for the satisfiable and unsatisfiable cases.

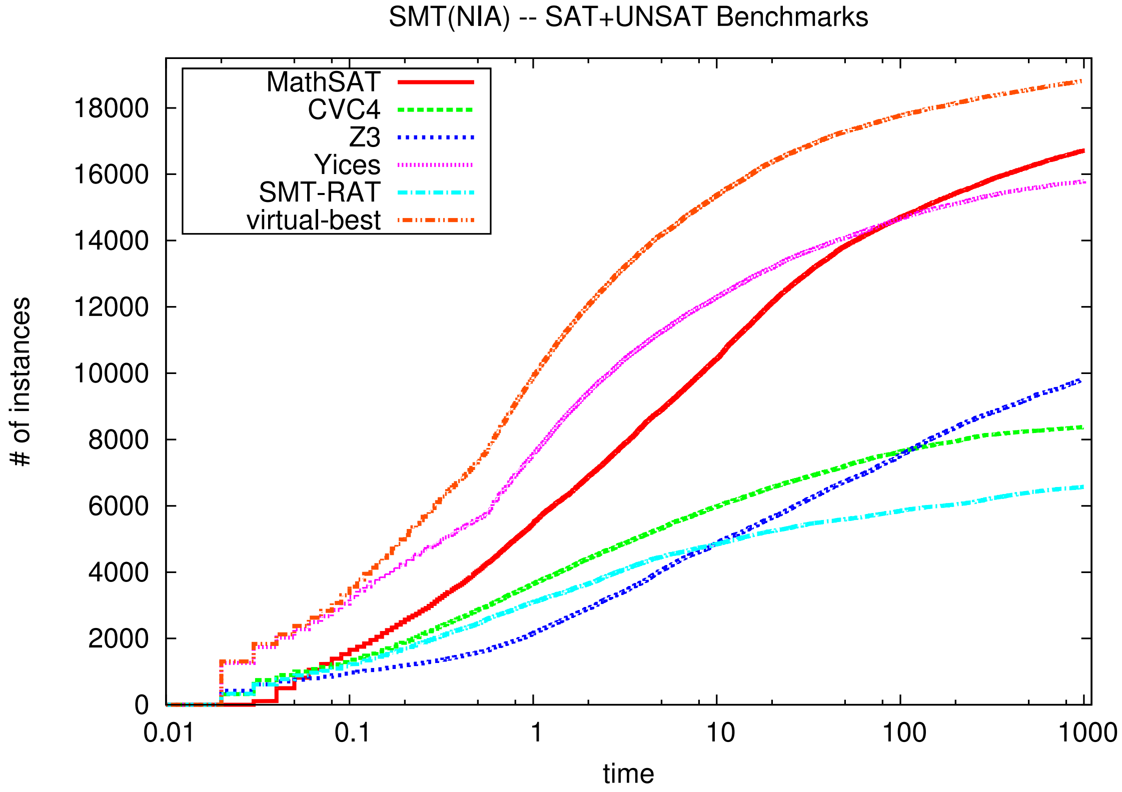
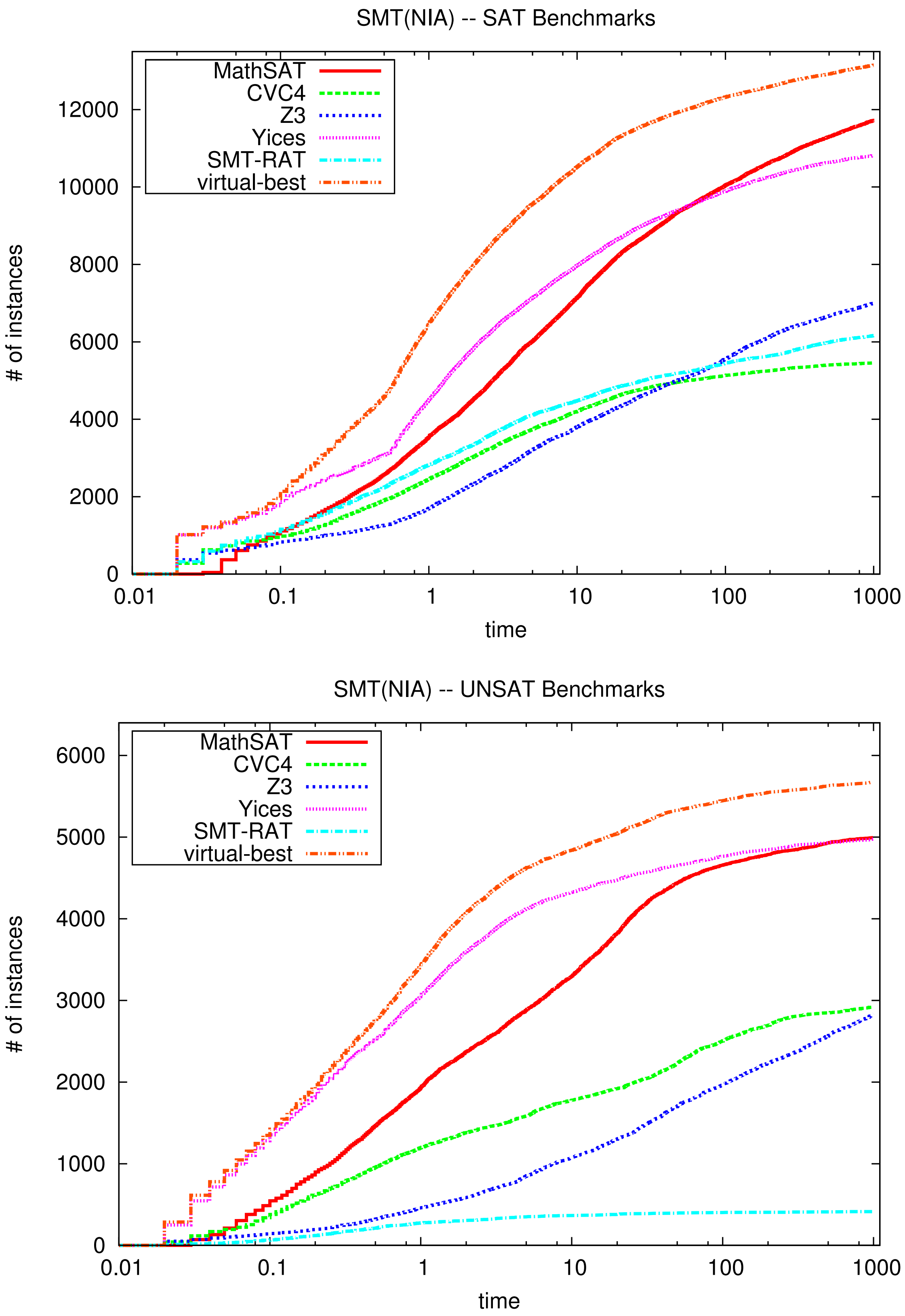


Figure 7.10: Survival plots for SMT($\mathcal{NIA}$) benchmarks



SMT(NIA) -- SAT Benchmarks

SMT(NIA) -- UNSAT Benchmarks

Figure 7.10: Survival plots for SMT($\mathcal{NIA}$) benchmarks

MathSAT solves 8672 (6479 SAT and 2193 UNSAT) benchmarks that are not solved by CVC4, and 327 (209 SAT and 118 UNSAT) vice versa.

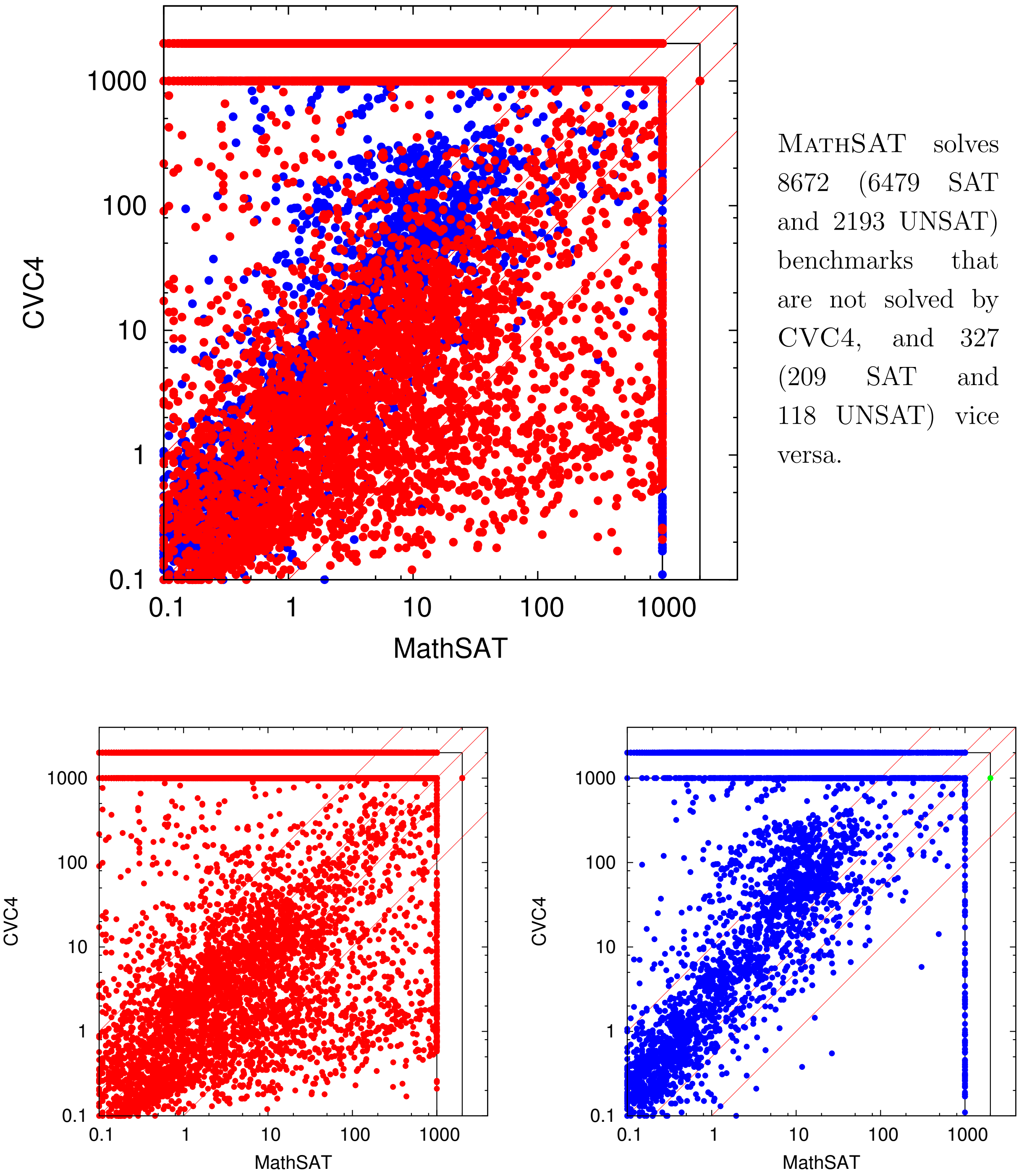


Figure 7.11: Scatters plots for SMT($\mathcal{NIA}$) benchmarks

MathSAT solves 7415 (5147 SAT and 2268 UNSAT) benchmarks that are not solved by z3, and 529 (417 SAT and 112 UNSAT) vice versa.

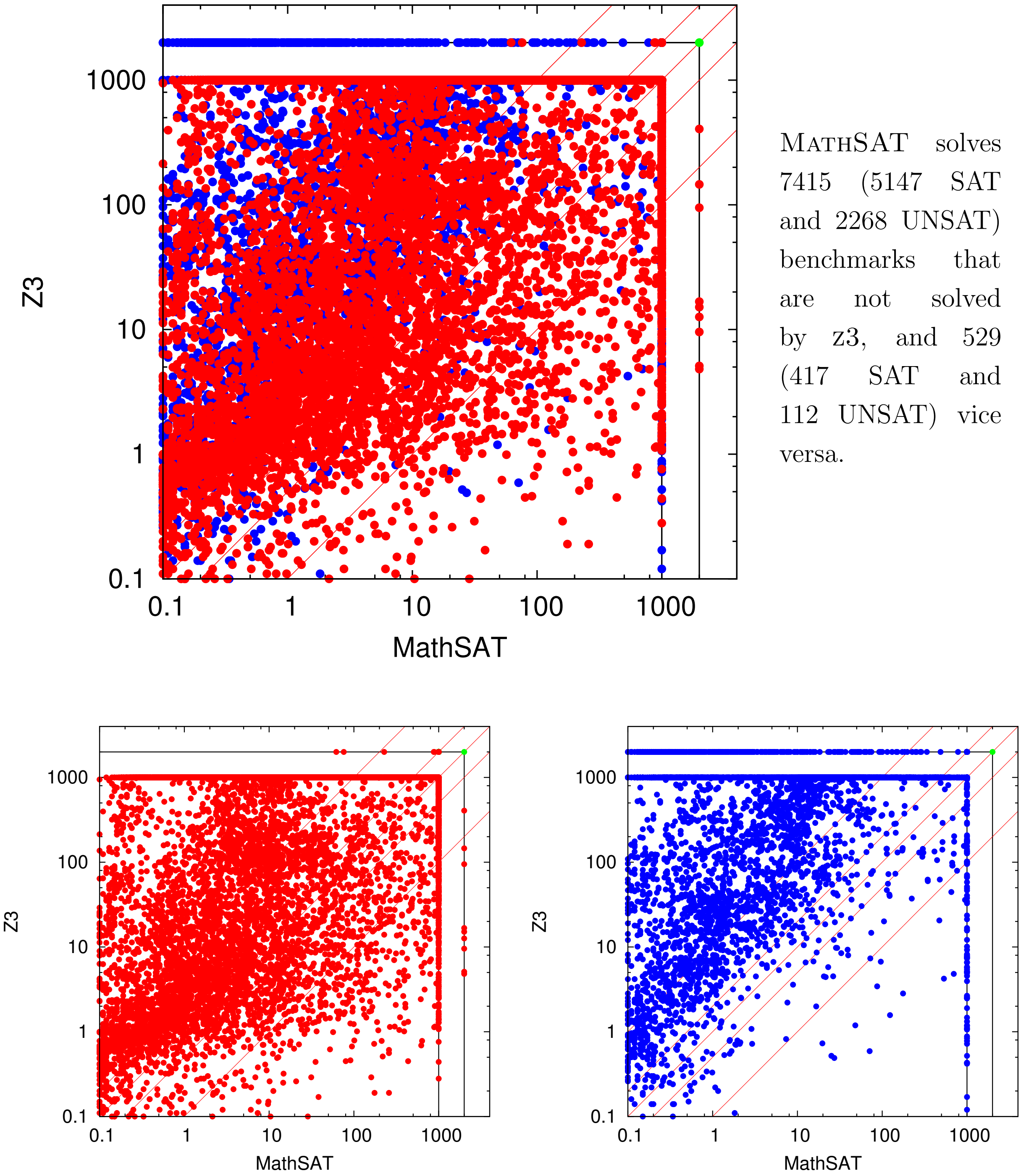


Figure 7.11: Scatters plots for SMT($\mathcal{NIA}$) benchmarks

MathSAT solves 2436 (1776 SAT and 660 UNSAT) benchmarks that are not solved by Yices, and 1505 (861 SAT and 644 UNSAT) vice versa.

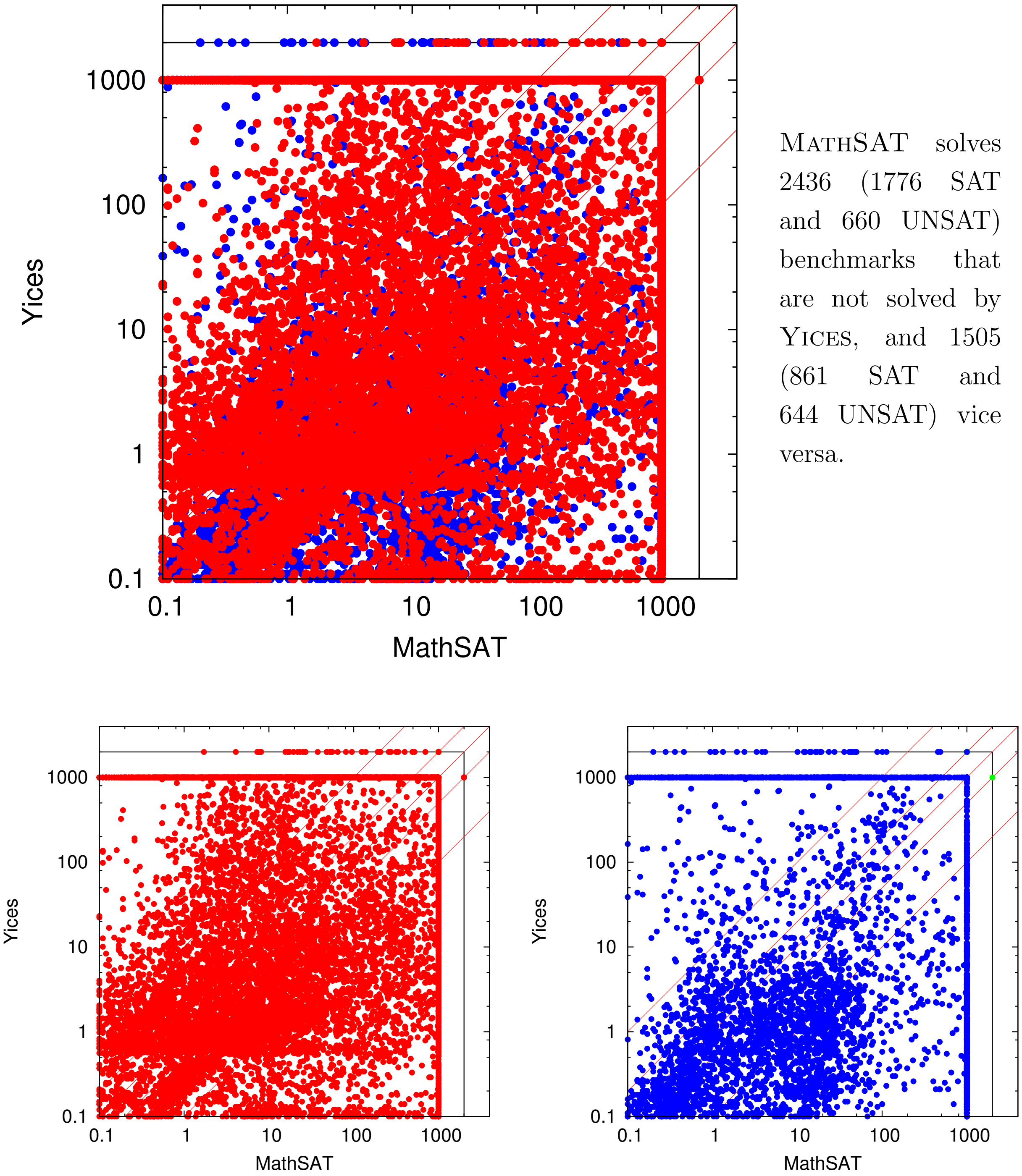


Figure 7.11: Scatters plots for SMT($\mathcal{NIA}$) benchmarks

MathSAT solves 10808 (6191 SAT and 4617 UNSAT) benchmarks that are not solved by SMT-RAT, and 667 (629 SAT and 38 UNSAT) vice versa.

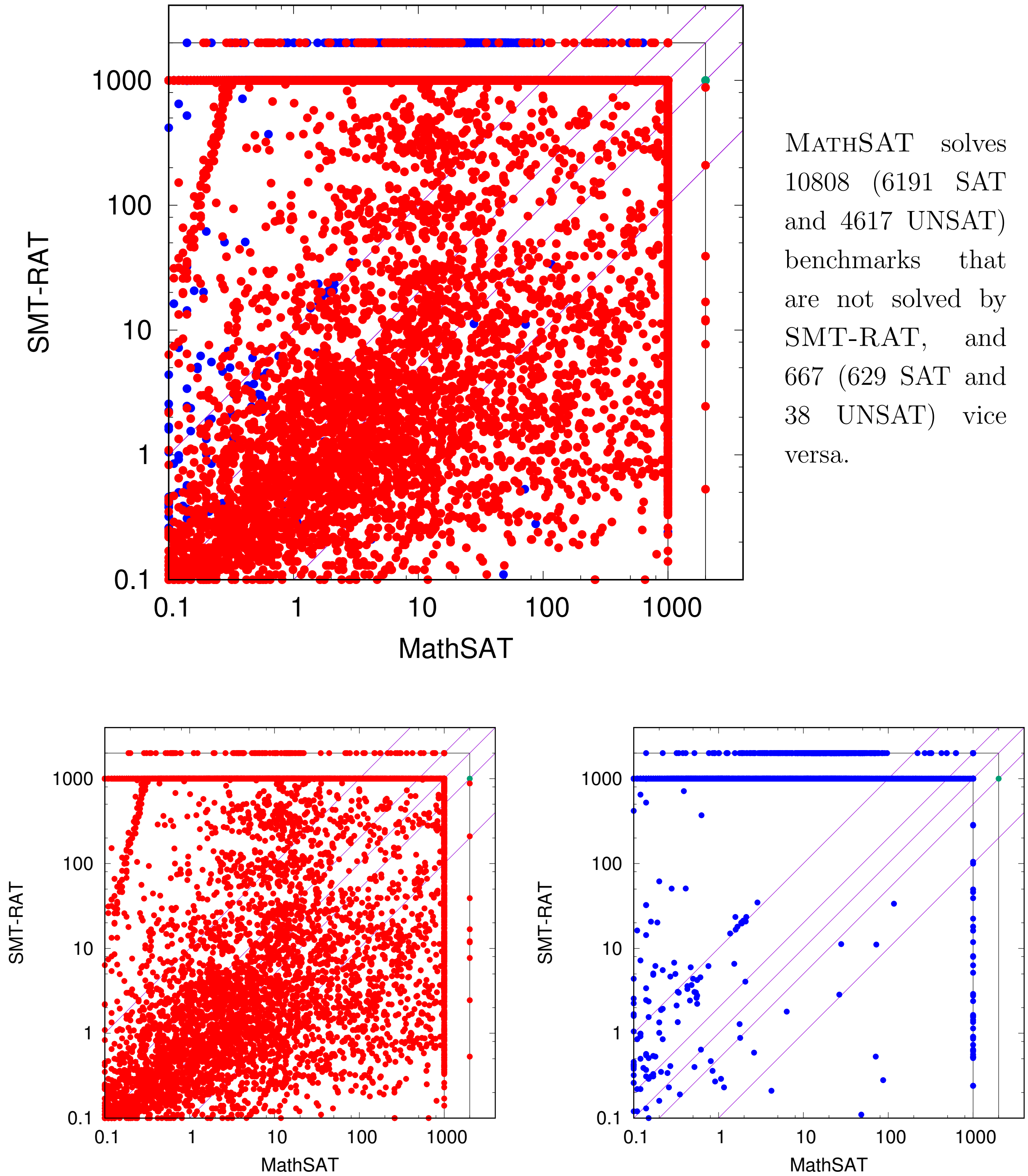


Figure 7.11: Scatters plots for SMT($\mathcal{NIA}$) benchmarks

	Total	**AProVE**	**Calypto**	**LassoRanker**	**LCTES**	**Leipzig**	**MCM**	**UltimateAutomizer**	**UltimateLassoRanker**	**VeryMax**
	(23876)	**(2409)**	**(177)**	**(106)**	**(2)**	**(167)**	**(186)**	**(7)**	**(32)**	**(20790)**
MathSAT	**11723/4993**	1642/561	**79**/89	**4/100**	0/**1**	126/**2**	12/0	0/**7**	**6/26**	**9854/4207**
CVC4	5453/2918	1309/610	63/89	**4**/90	0/**1**	83/**2**	14/**5**	0/6	**6/26**	3974/2089
z3	6993/2837	1656/325	78/96	**4**/92	0/0	**162**/0	20/1	0/**7**	**6/26**	5067/2290
Yices	10808/4977	1595/**708**	**79/97**	**4**/84	0/0	92/1	8/0	0/**7**	**6/26**	9024/4054
SMT-RAT	6161/414	**1663**/184	**79**/89	3/20	0/0	160/0	**21**/0	0/1	**6/26**	4229/94
VirtualBest	13148/5672	1663/724	79/97	4/101	0/1	162/2	25/6	0/7	6/26	11209/4708

Table 7.3: Summary of SMT($\mathcal{NIA}$) experimental results

unsatisfiable benchmarks is somewhat surprising. A more in-depth look at CVC4 reveals that the default configuration does not enable the tangent plane refinement. It suggests that the tangent plane refinement in incremental linearization is important for proving unsatisfiability of the $\mathcal{NIA}$ benchmarks.

It is worth to mention that MathSAT uniquely solves 2617 benchmarks with the exception of CVC4 (if we include CVC4 in the comparison, then the number becomes 2249). It also solves 2133 benchmarks with unknown status in SMT-LIB.[6]

In the VeryMax benchmarks family which contains 20790 instances, MathSAT dominantly solves most problems. The difference between MathSAT and Yices is 983, whereas the difference concerning the solvers based on bit-blasting approaches is between 6704 to 9783. These benchmarks are coming from a practical domain – termination analysis of software programs [BBL+17]. The relatively poor performance of the bit-blasting approaches on these problems suggests that they are not adequate for such application domains. On the other hand, incremental linearization and CAD with branch-and-bound are quite effective in this family.

[6] This value refers to the tagging of the SMT-LIB on 2017-06-17.

Summary

In this part, we addressed the SMT problem with respect to the theories of nonlinear arithmetic and transcendental functions. The key idea is to abstract nonlinear and transcendental functions as uninterpreted functions in the combined theories of $\mathcal{LA}$ and $\mathcal{UF}$, for which efficient solvers exist. The uninterpreted functions in the abstract domain, corresponding to nonlinear and transcendental functions in the original theory, are incrementally axiomatized through upper- and lower-bounding piecewise-linear constraints. The refinement is driven by the existence of spurious models. In the case of transcendental functions, the management of irrational values is particularly tricky, and care is required to ensure the soundness of the abstraction.

The approach is proved correct, and it has been implemented in the MathSAT SMT solver. We carried out an extensive experimental evaluation on a broad set of SMT benchmarks, and the results clearly demonstrate the effectiveness of incremental linearization. In particular we can make the following general remarks:

- Incremental linearization is highly competitive for $\mathcal{NRA}$. Given the maturity of the other solvers, we found it quite surprising to discover how well it compares on the SMT($\mathcal{NRA}$) benchmarks.

- Incremental linearization appears to be a very effective technique to deal with transcendental functions, a theory for which no complete methods are available. A key factor in the case of trigonometric functions appears to be the idea of reduction to the base interval. Incremental linearization could be improved further by integration with interval-based techniques.

- Incremental linearization remarkably outperform state-of-the-art SMT($\mathcal{NIA}$) techniques, in terms of total number of solved instances.
- The satisfiability-oriented methods presented in this part are often able to conclude Sat in cases where complete techniques bail out or do not exist.

Part III

Verification Modulo Nonlinear Arithmetic and Transcendental Functions

Overview

VMT is a fundamental research area (e.g., checking invariants for infinite-state transition systems). In general, given a theory, VMT is harder than SMT, due to the underlying notion of reachability: VMT is undecidable even for relatively simple theories such as $\mathcal{LRA}$ [HKPV98]. Yet, VMT($\mathcal{LA}$) tools are reasonably effective in practice, based on the power of SMT($\mathcal{LA}$) solvers and the ability to automatically construct abstractions. This is hardly the case for invariant checking for nonlinear transition systems. In fact, many real-world industrial design (e.g., aerospace, automotive) require modeling as transition systems over nonlinear arithmetic and transcendental functions.

In this part, we address the problem of checking invariants for transition systems over $\mathcal{NTA}$. Our main contribution lies in extending incremental linearization for the case of VMT, that is described in Chapter 8.

We have implemented the approach in the nuXmv VMT model checker, relying on the IC3 with implicit abstraction [CGMT16] engine. The implementation details and the experimental evaluation are presented in Chapter 9.

Remark 5. In principle, the procedure proposed later in this part can also work for $\mathcal{NIA}$. However, we skip the discussion here since currently, we do not have the infrastructure for evaluating it. This is left as a future work.

Chapter 8

VMT via Incremental Linearization

Similar to the SMT case, the idea of incremental linearization for VMT is to abstract transition systems in the $\mathcal{LA}$ and $\mathcal{UF}$ domain. The uninterpreted functions are used to model nonlinear and transcendental functions in transition systems. Then, we eliminate spurious counterexamples in VMT($\mathcal{UFLA}$) by incrementally adding linear constraints to tighten the piecewise-linear envelope around the (uninterpreted counterpart of the) abstract multiplication and transcendental functions.

The underlying rationale is that, for many practical problems, reasoning with full precision over nonlinear and transcendental functions may not be necessary. For example, constructing a piecewise-linear invariant may be sufficient to prove that the (nonlinear) transition system at hand satisfies a given property. The linearization is performed incrementally and only when and where needed, driven by the spurious counterexamples.

By means of Incremental linearization, we also tackle invariant checking for transition systems over $\mathcal{NRA}$ and transcendental functions. The key insight is to implement the Counterexample Guided Abstraction Refinement (CEGAR) [CGJ$^+$00] loop on top of an abstract version of the transition system expressed over $\mathcal{UFLRA}$. We leverage the characteristics of a recently introduced approach [CGMT16] based on the combination of

IC3 and predicate abstraction.

We remark that this approach based on incremental linearization has strong advantages over other VMT($\mathcal{NTA}$) approaches that could be obtained from traditional SMT-based algorithms by delegating the management of nonlinearity to an SMT($\mathcal{NTA}$) solver. BMC and k-induction [SSS00, ES03b] are be relatively simple to implement (given an SMT solver for the required theory) but have very limited effectiveness when it comes to proving properties over infinite-state transition systems. Extending other approaches (e.g., interpolation, IC3 [McM03, CGMT16]) to handle nonlinearities at the level of the solver would require the SMT($\mathcal{NRA}$) solver (or, worse, the SMT($\mathcal{NTA}$) solver), to carry out interpolation or quantifier elimination, and to proceed incrementally. These extra functions are usually not available, or they have a very high computational cost.

Structure of the Chapter. We describe the VMT($\mathcal{NTA}$) procedure based on incremental linearization in §8.1. In §8.2 we present the correctness proof of the procedure, and we discuss related work in §8.3.

8.1 Incremental Linearization for VMT($\mathcal{NTA}$)

We now consider the problem of VMT($\mathcal{NTA}$), exploring several approaches. A first direction is the direct use of an SMT($\mathcal{NTA}$) solver, like the one described in previous sections, to extend known SMT-based verification algorithms. This extension can be relatively easy in the case of BMC and k-induction, given that both techniques interact with the SMT($\mathcal{NTA}$) solver largely as if it were a "black box", as shown in Fig. 8.1.

Although effective in the finite-state case for safe instances in some practical cases, k-induction turns out to be quite weak even for $\mathcal{NRA}$.

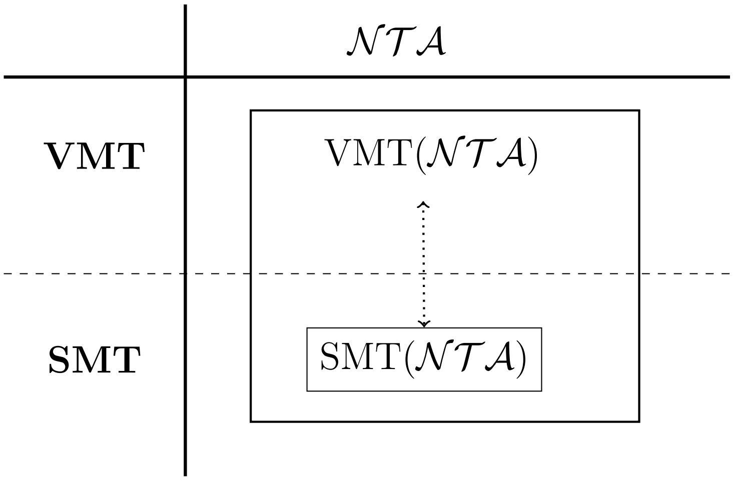


Figure 8.1: Solving VMT($\mathcal{NTA}$) via SMT($\mathcal{NTA}$)-based procedures

Consider for example the transition system $\langle I(X), T(X,X')\rangle$ s.t.

$$\begin{aligned} I(X) &\doteq x \geq 2 \wedge y \geq 2 \wedge z = x * y, \\ T(X,X') &\doteq x' = x + 1 \wedge y' = y + 1 \wedge z' = x' * y' \end{aligned}$$

The property $P(X) \doteq z \geq x + y$ is not k-inductive, not even for a very large value of k. Thus, the typical proving techniques based on k-induction using an SMT($\mathcal{NRA}$) solver will not be able to prove it.

Thus, one could explore the extension to $\mathcal{NTA}$ of other SMT-based approaches, such as interpolation-based verification or IC3 (e.g., [CGMT16, HB12b, KGC16b, BBW14]), that are often more effective than k-induction. However, there are substantial difficulties in lifting them to the case of $\mathcal{NTA}$. First, they require an interpolating SMT($\mathcal{NTA}$) solver, which is not available. Second, the effectiveness of IC3 depends on an efficient, incremental interaction with the underlying SMT engine, which is asked to solve a large number of relatively cheap and often satisfiable queries. The procedure of §6.1, however, can be very expensive, especially for satisfiable queries. Finally, some IC3 extensions require the ability to perform (approximated) quantifier eliminations, a feature not provided by the algorithm of Fig. 6.1.

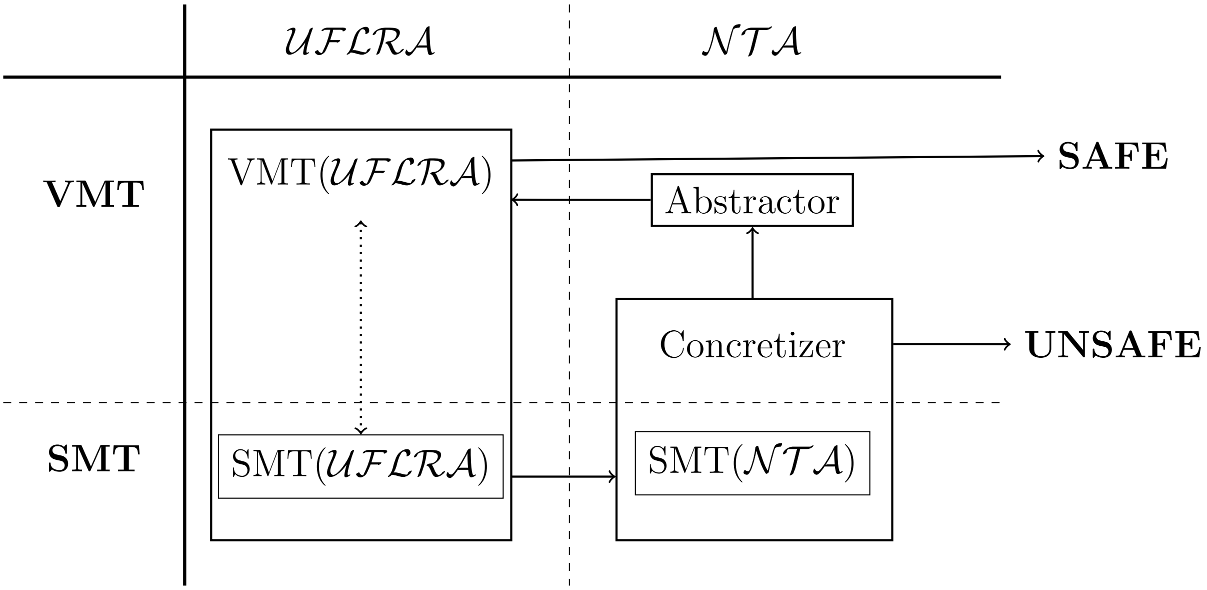


Figure 8.2: Solving VMT($\mathcal{NTA}$) via incremental linearization

Therefore, we propose a new direction, that radically differs from plugging a solver for SMT($\mathcal{NTA}$) into a known SMT-based algorithm. Instead, we adopt an abstraction-refinement approach, lifting the concepts of *incremental linearization* of $\mathcal{NTA}$ at the transition system level. In this approach, the VMT($\mathcal{NTA}$) problem is abstracted to a VMT($\mathcal{UFLRA}$) verification problem. A very efficient verification engine for VMT($\mathcal{UFLRA}$) is used as a black box, and the abstraction is refined in order to rule out spurious counterexamples. (See Fig. 8.2)

8.1.1 The Main Procedure

The pseudo-code of the main procedure is reported in Fig. 8.3. The main function IC3-NTA-PROVE takes as input a transition system $\mathcal{S}$ and a formula φ and checks if φ is an invariant property of $\mathcal{S}$. Notice that, as with SMT-NTA-CHECK in §6.1, IC3-NTA-PROVE is not guaranteed to terminate, so that we implicitly assume that it is stopped as soon as some given resource budget (e.g., time, memory, number of iterations) is

```
bool IC3-NTA-PROVE ($\mathcal{S}$ : transition system $\langle X, I, T\rangle$, $\varphi$ : invariant property):
1.  $\langle \mathcal{S}', \varphi' \rangle$ := VMT-PREPROCESS($\mathcal{S}, \varphi$)
2.  $\langle \widehat{\mathcal{S}}, \widehat{\varphi} \rangle$ := VMT-INITIAL-ABSTRACTION($\mathcal{S}', \varphi'$)
3.  while true:
4.      $\langle ok, \widehat{cex} \rangle$ := IC3-UFLRA-PROVE($\widehat{\mathcal{S}}, \widehat{\varphi}$)
5.      if ok:                  # property proved
6.          return true
7.      ⟨is_cex, $\Gamma$⟩ := SMT-NTA-CONCRETIZE-ABSTRACT-CEX($\mathcal{S}', \widehat{\mathcal{S}}, \widehat{\varphi}, \widehat{cex}$)
8.      if is_cex:              # counterexample found
9.          return false
10.     $\langle \Gamma_I, \Gamma_T \rangle$ := REFINE-TRANSITION-SYSTEM($\widehat{\mathcal{S}}, \Gamma$)
11.     $\widehat{\mathcal{S}} := \langle X, \widehat{I} \wedge \bigwedge \Gamma_I, \widehat{T} \wedge \bigwedge \Gamma_T \rangle$
```

Figure 8.3: Verification of $\mathcal{NTA}$ transition systems via abstraction to $\mathcal{UFLRA}$

exhausted.

We first apply a preprocessing step to $\langle \mathcal{S}, \varphi \rangle$ (function VMT-PREPROCESS) which is analogous to that of SMT-PREPROCESS in SMT-NTA-CHECK, producing $\langle \mathcal{S}', \varphi' \rangle$. Then we generate $\langle \widehat{\mathcal{S}}, \widehat{\varphi} \rangle$, an abstract $\mathcal{UFLRA}$ version of the input $\mathcal{NTA}$ transition system $\mathcal{S}$ and invariant φ, by invoking VMT-INITIAL-ABSTRACTION. As with the SMT case, the VMT-INITIAL-ABSTRACTION function replaces every non-linear multiplication and transcendental function in the transition system and property with the corresponding uninterpreted function symbol.

Then the procedure enters a loop (lines 3-11). At each iteration, the pair $\langle \widehat{\mathcal{S}}, \widehat{\varphi} \rangle$ is first checked by IC3-UFLRA-PROVE which implements IC3ia [1] [CGMT16], a procedure for VMT($\mathcal{UFLRA}$) that extends IC3 with implicit predicate abstraction [Ton09]. If the invariant property is verified in the abstract domain $\mathcal{UFLRA}$, then

[1] Notice that, other approaches, such as interpolation-based model checking, could in principle be used. Here we consider IC3ia because it is currently the most effective procedure for VMT($\mathcal{UFLRA}$), and also because it allows us to leverage incrementality.

```
⟨constraint set, constraint set⟩ REFINE-TRANSITION-SYSTEM ($\widehat{\mathcal{S}}$, $\Gamma$):
let $\langle X, \widehat{I}, \widehat{T} \rangle = \widehat{\mathcal{S}}$
$\langle \Gamma_I, \Gamma_T \rangle := \langle \emptyset, \emptyset \rangle$
for each $\gamma$ in $\Gamma$:
    if vars($\gamma$) $\subseteq X^{\langle 0 \rangle}$:
        $\Gamma_I := \Gamma_I \cup \{\gamma\{X^{\langle 0 \rangle} \mapsto X\}\}$
    else if there exists $i > 0$ s.t. vars($\gamma$) $\subseteq X^{\langle i \rangle}$:
        $\Gamma_T := \Gamma_T \cup \{\gamma\{X^{\langle i \rangle} \mapsto X\}, \gamma\{X^{\langle i \rangle} \mapsto X'\}\}$
    else there exists $i > 0$ s.t. vars($\gamma$) $\subseteq X^{\langle i \rangle} \cup X^{\langle i+1 \rangle}$:
        $\Gamma_T := \Gamma_T \cup \{\gamma\{X^{\langle i \rangle} \mapsto X\}\{X^{\langle i+1 \rangle} \mapsto X'\}\}$
return $\langle \Gamma_I, \Gamma_T \rangle$
```

Figure 8.4: Refinement of the $\mathcal{UFLRA}$ transition system

it is also verified in the original $\mathcal{NTA}$ domain, so that the whole procedure returns **true**. Otherwise, a counterexample is produced, and the SMT-NTA-CONCRETIZE-ABSTRACT-CEX primitive is used to check whether it is spurious. If not so, then the whole procedure returns **false**. If so, the linear constraints generated by SMT-NTA-CONCRETIZE-ABSTRACT-CEX are used to refine the abstraction of the transition system, and the procedure enters a new iteration.

8.1.2 Spuriousness Check and Abstraction Refinement

When IC3-UFLRA-PROVE returns a counterexample trace $\widehat{cex}$ for the abstract system $\widehat{\mathcal{S}}$, we use the dedicated routine SMT-NTA-CONCRETIZE-ABSTRACT-CEX to check for its spuriousness. The first step is to build a formula ψ whose unsatisfiability implies that $\widehat{cex}$ is spurious. The formula ψ is built by unrolling the transition relation of $\widehat{\mathcal{S}}$, and optionally adding constraints that restrict the allowed transitions to be compatible with the states in $\widehat{cex}$. If SMT-NTA-CONCRETIZE-ABSTRACT-CEX returns true, the property is violated. If SMT-NTA-CONCRETIZE-ABSTRACT-CEX returns false, we use the constraints Γ produced during search to refine the transition system $\widehat{\mathcal{S}}$, using the procedure shown in Fig. 8.4. Essentially, REFINE-TRANSITION-SYSTEM translates back the linearization constraints from their unrolled version (on variables $X^{\langle 0\rangle}, X^{\langle 1\rangle}, \ldots, X^{\langle k\rangle}$) to their "untimed" version (on variables X and X'). Each γ constraint is added either to the initial-states formula or to the transition relation formula, depending on the distance (in terms of steps) of the variables occurring in it. Care must be taken in order to deal with the case where γ spawns across multiple time points: The routine SMT-NTA-CONCRETIZE-ABSTRACT-CEX is similar in spirit to the SMT-NTA-CHECK (discussed in §6.1), but it is designed in such a way that the constraints in Γ never span more than a single transition step – for example, a monotonicity constraint over terms $\text{TF}(x^{\langle i\rangle})$ and $\text{TF}(y^{\langle j\rangle})$ is instantiated only if $j = i$ or $j = i + 1$. Note that, despite the restriction in the constraints instantiation, SMT-NTA-CONCRETIZE-ABSTRACT-CEX is as powerful as SMT-NTA-CHECK in detecting the spuriousness of a counterexample. This is because SMT-NTA-CONCRETIZE-ABSTRACT-CEX can basically instantiate a finite number of constraints restricted to a single transition step such that their conjunction is equivalent to any constraint instantiated by SMT-NTA-CHECK.

Example

We now demonstrate the execution of the procedure in Fig. 8.3 by an example.

Example 8.1. Consider the earlier mentioned transition system

$\mathcal{S} \doteq \langle I(X), T(X, X')\rangle$ with the property $P(X)$:

$$\begin{aligned} I(X) &\doteq x \geq 2 \wedge y \geq 2 \wedge z = x * y, \\ T(X, X') &\doteq x' = x + 1 \wedge y' = y + 1 \wedge z' = x' * y' \\ P(X) &\doteq z \geq x + y \end{aligned}$$

After the initial abstraction, line 1-2, we have $\langle \widehat{\mathcal{S}}, \widehat{P}(X)\rangle$ where:

$$\begin{aligned} \widehat{\mathcal{S}} &\doteq \langle \widehat{I}(X), \widehat{T}(X, X')\rangle \\ \widehat{I}(X) &\doteq x \geq 2 \wedge y \geq 2 \wedge z = f_*(x, y), \\ \widehat{T}(X, X') &\doteq x' = x + 1 \wedge y' = y + 1 \wedge z' = f_*(x', y') \\ \widehat{P}(X) &\doteq z \geq x + y \end{aligned}$$

In the loop (line 3-11), after execution of line 4, IC3-UFLRA-PROVE returns an abstract counterexample of length 2. Then SMT-NTA-CONCRETIZE-ABSTRACT-CEX tries to concretize the abstract counterexample: in simplest form it builds a BMC unrolling of length 2 (see §9.1 for more details). SMT-NTA-CONCRETIZE-ABSTRACT-CEX returns false (meaning that the abstract counterexample is spurious) and the following set of linear constraints:

$$\Gamma \doteq \{\ (x^{\langle 1\rangle} > 2 \wedge y^{\langle 1\rangle} > 2) \rightarrow f_*(x^{\langle 1\rangle}, y^{\langle 1\rangle}) > 2 * x^{\langle 1\rangle} + 2 * y^{\langle 1\rangle} - 4\ \}$$

Next, REFINE-TRANSITION-SYSTEM refines $\widehat{\mathcal{S}}$ using Γ, such that:

$$\begin{aligned} \widehat{I}(X) &\doteq x \geq 2 \wedge y \geq 2 \wedge z = f_*(x, y), \\ \widehat{T}(X, X') &\doteq x' = x + 1 \wedge y' = y + 1 \wedge z' = f_*(x', y') \wedge \\ &\quad (x > 2 \wedge y > 2) \rightarrow f_*(x, y) > 2 * x + 2 * y - 4 \ \wedge \\ &\quad (x' > 2 \wedge y' > 2) \rightarrow f_*(x', y') > 2 * x' + 2 * y' - 4 \end{aligned}$$

After the refinement, another iteration of the loop is performed. Now IC3-UFLRA-PROVE returns true (meaning that the property is safe in $\widehat{\mathcal{S}}$), which means $P(X)$ is safe in $\mathcal{S}$. Therefore, true is returned. △

8.2 Proof of Correctness

We prove that the IC3-NTA-PROVE procedure (Fig. 8.3) is sound. First we show that the procedure maintains an over-approximation of the input transition system. Consider a generic loop in IC3-NTA-PROVE (Line 10-11) and let $\widehat{\mathcal{S}} := \langle X, \widehat{I} \wedge \bigwedge \Gamma_I, \widehat{T} \wedge \bigwedge \Gamma_T \rangle$ at the end of the loop.

Lemma 8.2. *$\widehat{\mathcal{S}}$ is an over-approximation of $\mathcal{S}$.*

Proof. We need to show that for every path σ_k in $\mathcal{S}$, there is a path $\widehat{\sigma}_k$ in $\widehat{\mathcal{S}}$. This is true because:

- by definition of σ_k, we have $s_0 \models I(X)$ and $s_i \wedge s_{i+1}\{X \mapsto X'\} \models T(X, X')$ for $0 \leq i \leq k-2$, and
- similar to the the proof of Lemma 6.7, we can construct a path $\widehat{\sigma}_k$ in a such way that $\widehat{s}_0 \models \widehat{I} \wedge \bigwedge \Gamma_I$ and $\widehat{s}_i \wedge \widehat{s}_{i+1}\{X \mapsto X'\} \models \widehat{T}(X, X') \wedge \bigwedge \Gamma_T$ for $0 \leq i \leq k-2$.

Hence the statement holds. □

Lemma 8.3. *$\mathcal{S} \not\models \varphi$ implies $\widehat{\mathcal{S}} \not\models \widehat{\varphi}$.*

Proof. $\mathcal{S} \not\models \varphi$ means there is a path $\sigma_k = s_0, s_1, \ldots, s_{k-1}$ in $\mathcal{S}$ such that $s_{k-1} \models \neg\varphi$. Then by Lemma 8.2, we can also construct a path $\widehat{\sigma}_k = \widehat{s}_0, \widehat{s}_1, \ldots, \widehat{s}_{k-1}$ in $\widehat{\mathcal{S}}$ in a way that $\widehat{s}_{k-1} \models \neg\widehat{\varphi}$ (similar to the proof of Lemma 6.7). Hence the statement holds. □

Lemma 8.4. IC3-UFLRA-PROVE$(\widehat{\mathcal{S}}, \widehat{\varphi})$ *is sound, i.e., when it returns $\langle True, \ldots \rangle$ then $\widehat{\mathcal{S}} \models \widehat{\varphi}$, and when it returns $\langle False, \widehat{cex} \rangle$ then $\widehat{\mathcal{S}} \not\models \widehat{\varphi}$ where $\widehat{cex}$ is a counterexample.*

Proof. Proof omitted – see [CGMT16]. □

Theorem 8.5. IC3-NTA-PROVE$(\mathcal{S}, \varphi)$ *is sound, i.e., when it returns True then $\mathcal{S} \models \varphi$ and when it returns False then $\mathcal{S} \not\models \varphi$.*

Proof. IC3-NTA-PROVE($\mathcal{S}, \varphi$) return true only when IC3-UFLRA-PROVE($\widehat{\mathcal{S}}, \widehat{\varphi}$) returns $\langle True, \ldots \rangle$. By Lemma 8.2 and Lemma 8.3, $\mathcal{S} \models \varphi$. IC3-NTA-PROVE($\mathcal{S}, \varphi$) return false only when the simulation of the abstract counterexample $\widehat{cex}$ (Lemma 8.4) by the SMT-NTA-CONCRETIZE-ABSTRACT-CEX succeeds, and by Theorem 6.9 $\mathcal{S} \not\models \varphi$. □

8.3 Related Work

As mentioned earlier, there are not many tools that work on symbolic transition systems over $\mathcal{NTA}$. iSAT3 and dReach are two tools which can return "safe" or "unsafe" or "maybe unsafe" result. They use nonlinear solvers based on interval constraint propagation. In contrast, our approach is based on linearization. Unlike iSAT3 and dReach, we avoid numerical approximation, and thus provide definite answers – safe or unsafe. Notice that dReach is a bounded model checker, and it can not prove unbounded properties; iSAT3 uses interpolation-based approach. In contrast, we are not only able to use BMC and k-induction, but also a more powerful IC3-based technique, i.e., IC3ia.

In the context of $\mathcal{NRA}$, the work in [CGKT16, MFK$^+$16] performs linearization statically at the beginning of the analysis, whereas our linearization technique is done incrementally. Moreover, they use an $\mathcal{LRA}$ solver under the hood. Instead we rely on a $\mathcal{UFLRA}$ solver.

Chapter 9

Implementation and Experimental Evaluation

The VMT procedures presented in the previous chapter have been implemented within the nuXmv VMT model checker [CCD^{+}14]. The description of the procedures leaves some flexibility for different heuristics regarding refinement strategies and implementation choices. In the next section, we describe the most important ones. We remark that these choices do not affect the soundness of the approach, but they can have an important impact on performance.

The nuXmv VMT model checker has been extended to deal with non-linear multiplications and with the transcendental functions supported by its underlying SMT solver MathSAT. The verification engines of nuXmv have been extended to deal with invariant checking based on Bounded Model Checking, k-induction, and incremental linearization over IC3. The implementation of BMC and k-induction extends the algorithms already implemented in nuXmv for the $\mathcal{LRA}$ case. This extension was relatively simple, given the augmented capabilities of the underlying MathSAT solver. We also extended nuXmv with the capability to output BMC and k-induction proof obligations in SMT-LIB format. [1] The implementation

[1] More precisely, with the extension of the SMT-LIB format to transcendental function symbols.

of the VMT($\mathcal{NTA}$) algorithm based on incremental linearization was more involved. It builds on top of the best nuXmv engine for VMT($\mathcal{UFLRA}$), i.e., IC3 with Implicit Abstraction [CGMT16], leveraging its stateful, incremental nature. We chose not to implement the incremental linearization loop on top of an induction-based engine. We have no reason to believe that it would bring significant added value. Given the complementarity between IC3 and interpolation-based methods demonstrated in the finite-state case, an incremental linearization loop over an interpolation-based solver for VMT($\mathcal{UFLRA}$) might yield additional solving capability, and it is currently under consideration.

9.1 Implementation Details

Counterexample Checking and Refinement

Different heuristics can be considered for implementing the abstract counterexample check routine of Fig. 8.3 (function SMT-NTA-concretize-abstract-cex), trading generality for complexity. In particular, the unrolling of the transition system to check the feasibility of the abstract counterexample could be fully constrained by the states in $\widehat{cex}$ (thus checking only one abstract counterexample path per iteration); it could be only partially constrained (e.g., by considering only the Boolean variables and/or the state variables occurring only in linear constraints); or it could be left unconstrained, considering only the length of the abstract counterexample. In our current implementation, we only consider the length of $\widehat{cex}$ to build a BMC formula that checks for any counterexample of the given length, leaving the investigation of alternative strategies to future work.

```
$\langle \text{constraint set}, \text{constraint set} \rangle$ REDUCE-CONSTRAINTS ($\langle X, \widehat{I}, \widehat{T} \rangle$, $\widehat{cex}$, $\langle \Gamma_I, \Gamma_T \rangle$):
1.  $\psi := I^{\langle 0 \rangle} \wedge \Gamma_I^{\langle 0 \rangle} \wedge \widehat{cex}[0]^{\langle 0 \rangle} \wedge \bigwedge_{i=0}^{|\widehat{cex}|-1} \left( T^{\langle i \rangle} \wedge \Gamma_T^{\langle i \rangle} \wedge \Gamma_T^{\langle i+1 \rangle} \wedge \widehat{cex}[i+1]^{\langle i+1 \rangle} \right)$
2.  sat := SMT-UFLRA-CHECK($\psi$)
3.  assert not sat
4.  let $C$ be an unsatisfiable core of $\psi$
5.  $\Gamma_I = \{\gamma \in \Gamma_I \mid \gamma\{X \mapsto X^{\langle 0 \rangle}\} \in C\}$
6.  $\Gamma_T = \{\gamma \in \Gamma_T \mid \exists j > 0 \text{ s.t. } \gamma\{X \mapsto X^{\langle j \rangle}\}\{X' \mapsto X^{\langle j+1 \rangle}\} \in C\}$
7.  return $\langle \Gamma_I, \Gamma_T \rangle$
```

Figure 9.1: Reducing the constraints needed for refinement

Reduction in the Number of Constraints

In general, not all the constraints generated during a call to SMT-NTA-CONCRETIZE-ABSTRACT-CEX are needed to successfully block a counterexample, especially when using eager constraint instantiation strategies at the SMT level and when considering (like described above) all possible counterexample traces of a given length at each call to SMT-NTA-CONCRETIZE-ABSTRACT-CEX. In the long run, having a large number of redundant constraints can be quite harmful for performance.

In order to mitigate this problem, we apply a filtering strategy to the set of constraints, before adding them to the transition system. The strategy is based on the use of unsatisfiable cores, and it is shown as pseudo-code in Fig. 9.1. The function REDUCE-CONSTRAINTS takes as input the current abstract transition system, the current abstract counterexample $\widehat{cex}$, and the sets of refinement constraints Γ_I and Γ_T returned by the function REFINE-TRANSITION-SYSTEM of Fig. 8.4. Then it builds an abstract BMC formula constrained by $\widehat{cex}$ (line 1 of Fig. 9.1). This formula is unsatisfiable, and we can extract a reduced set of constraints that is still sufficient for blocking the abstract counterexample by removing all the constraints that are not in the unsatisfiable core produced by the SMT solver (lines

5–6).

9.2 Experimental Setup

We now experimentally evaluate the proposed approaches for VMT. The experiments were run on a cluster of identical machines equipped with 2.6GHz Intel Xeon X5650 processors. The memory limit was set to 6 GB. We used 3600 seconds for the VMT experiments. (The time out for VMT was dictated by restrictions in the computing power available, but appears to be reasonable given the number of benchmarks.) The results of the various solvers were automatically cross checked, and no discrepancies were reported.

We present the data using tables, survival plots, and scatters plots. The tables present detailed results: each column represents a benchmark family, each entry shows the number of sat/unsat results reported, for tools reporting MaybeUnsafe the number is shown in parentheses, the best performer for each family is highlighted in boldface, and the overall best is underlined. The survival plots compare the performance of multiple approaches. The x-axis shows the solving time in log-scale, and the y-axis shows the number of instances solved within the corresponding time. (Notice that we may use different scales on different plots.) The scatters plots compare pairwise solvers S_1 and S_2 on individual benchmarks: each point (t_1, t_2) in a scatters represents a benchmark problem that was solved in time t_i by solver S_i. We adopt a logarithmic scale, and report time out and memory out as separate lines. Red [blue, respectively] dots are for unsafe [safe, resp.] instances. Green dots indicate unknown instances. Diagonal lines mark 2x and 10x performance differences. Points on the inner edges indicate timeouts, those on the outer edges indicate other errors (memory outs or aborts). In the comparison, by VirtualBest we mean the results of a virtual portfolio solver that performs on each benchmark as the best of the solvers in the portfolio.

Benchmarks

We collected 114 benchmarks over VMT($\mathcal{NRA}$) and 126 benchmarks over VMT($\mathcal{NTA}$). The VMT($\mathcal{NRA}$) benchmarks set consists of the following families:

- *Handcrafted:* 14 (13 safe and 1 unsafe) benchmarks.
- *HyComp:* 7 (3 safe, 4 unsafe) benchmarks from [CMT12] and converted to VMT($\mathcal{NRA}$) using HyComp [CGMT15].
- *HYST:* 65 benchmarks generated from the hybrid systems examples in the HYST [BBJ15] distribution, by approximating the continuous time with a fixed rate sampling. This process is done automatically using an extended version of HYST. Since the generated benchmarks are approximations, their status is unknown.
- *iSAT3 and iSAT3-CFG:* 11 (7 safe, 4 unsafe) benchmarks from [MSN+16] and the iSAT3 examples available online.
- *nuXmv:* 2 safe benchmarks, with complex Boolean structure, from the nuXmv users' mailing list.
- *SAS13:* 13 benchmarks are generated from the C programs used in [BDG+13], but interpreted over $\mathcal{NRA}$ instead of the theory of IEEE floating-point numbers. This makes some of the instances unsafe.
- *TCM:* 2 safe benchmarks from the Simulink models (taken from the case study [BBD+15]), and converted them to VMT($\mathcal{NRA}$) via the Simulink to nuXmv flow (see Chapter 10).

The VMT($\mathcal{NTA}$) benchmarks set consists of the following families:

- *Handcrafted:* 3 (safe) benchmarks.
- *HARE, HYST, and WBS:* 50, 57, and 12 benchmarks obtained from the discretization (using HyComp [CGMT15]) of the hybrid systems benchmarks from [RPV17, BBJ15, CMS16] (unknown status).
- *iSAT3:* 4 benchmarks from the iSAT3 webpage.

Similar to the case of SMT($\mathcal{NTA}$), we included a scaled-down version of the VMT($\mathcal{NTA}$) benchmarks, with real-valued variables constrained to the $[-300, 300]$ interval, in order to include iSAT3 in the comparison.

The benchmarks of the comparison are expressed in the VMT-LIB language [2], that extends the SMT-LIB language so that the SMT formulae are interpreted as the initial condition and the transition relation of the transition system. Cross-translation scripts were developed for the iSAT3 and dReal benchmarks.

Other Approaches

We compared the incremental linearization algorithm implemented in nuXmv (in the following referred to as IncreLin-nuXmv) against the static abstraction approach proposed (for the case of $\mathcal{NRA}$) in [CGKT16] (referred to as StaticLin-nuXmv). In the case of $\mathcal{NTA}$, StaticLin-nuXmv also uses some basic constraints to limit the interpretation of transcendental functions. We also compared IncreLin-nuXmv against various approaches based on the direct use of an SMT($\mathcal{NRA}$) or SMT($\mathcal{NTA}$) solver.

[2] Information available at `http://www.vmt-lib.org/`. A complete specification of the VMT-LIB language can be found in the nuXmv User Manual available at `https://nuxmv.fbk.eu/downloads/nuxmv-user-manual.pdf`.

- BMC: BMC-NRA-Z3, based on z3 (limited to $\mathcal{NRA}$); BMC-NRA-dReal and BMC-NTA-dReal, based on dReal; BMC-NRA-MathSAT and BMC-NTA-MathSAT, based on MathSAT.

- k-induction: K-induction-NRA-Z3, based on z3 (limited to $\mathcal{NRA}$); K-induction-NRA-dReal and K-induction-NTA-dReal, based on dReal; K-induction-NRA-MathSAT and K-induction-NTA-MathSAT, based on MathSAT.

- The interpolation-based iSAT3 engine [MSN+16], with two different levels of precision – Interpolation-iSAT3$_{[1e-1]}$ and Interpolation-iSAT3$_{[1e-9]}$.

Remark 6. The BMC and k-induction solvers using dReal are based on a script developed with the specific objective of this evaluation. When dReal returns a MaybeSat result on a BMC query (or a base case query) the script returns MaybeUnsafe. When dReal returns MaybeSat on an inductive step query the script considers it a failed induction and increases k. This does not hamper the correctness of Safe results. However, we notice that the loop is slightly different from the one that is implemented in nuXmv on top of MathSAT and of z3, in that they may run out of resources trying to prove that an inductive query is satisfiable. The ability to "bail out" from hard satisfiable inductive checks gives dReal a slight advantage.

9.3 Results

Results for VMT($\mathcal{NRA}$)

The results for VMT($\mathcal{NRA}$) are summarized in Table 9.1. The experimental results for VMT($\mathcal{NRA}$) clearly demonstrate the merits of incremental linearization compared to BMC and k-induction approaches based on the

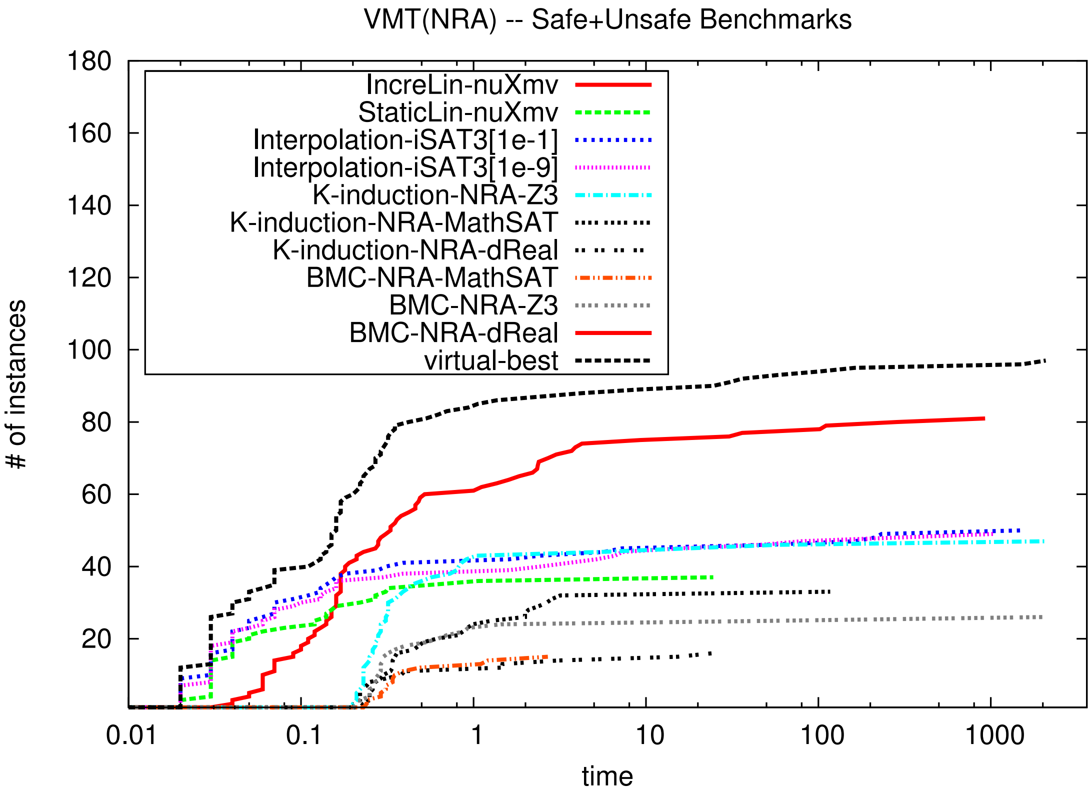
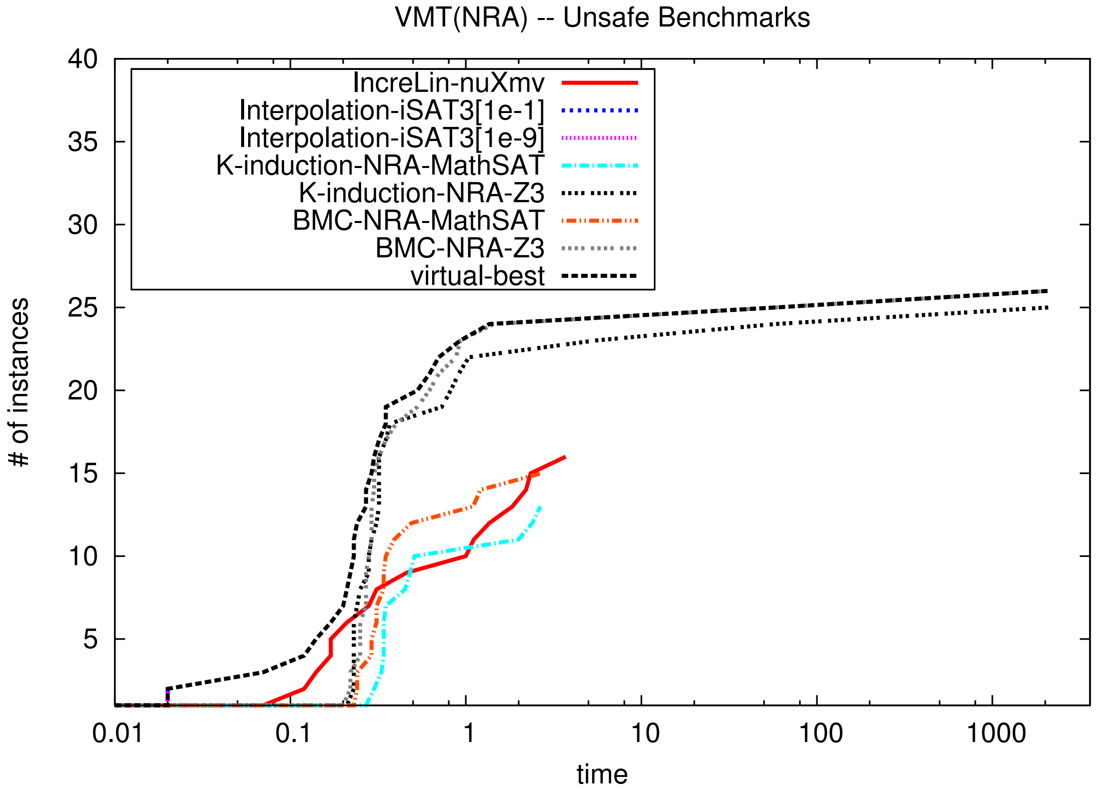


Figure 9.2: Survival plots for VMT($\mathcal{NRA}$) benchmarks

direct usage of SMT($\mathcal{NRA}$). IncreLin-nuXmv is by far the best solver among all the available ones, with 81 benchmarks solved against the 50 solved by the runner-up Interpolation-iSAT3$_{[1e-1]}$. The purely-static approach of StaticLin-nuXmv only solves 37, thus confirming the importance of incremental refinement of the abstraction. The scatters plot in Fig. 9.3 reports the comparison on VMT($\mathcal{NRA}$) problems. We notice that IncreLin-nuXmv dominates the approaches based on k-induction: every safe instance solved by k-induction is also solved by incremental linearization (with the exception of one instance). Interpolation has some complementarity wrt. incremental linearization: it solves 5 benchmarks where IncreLin-nuXmv (as well as all the other engines) times out. We conjecture that incremental linearization for SMT($\mathcal{NRA}$) could be extended to produce interpolants applicable to verification. We also see that the BMC



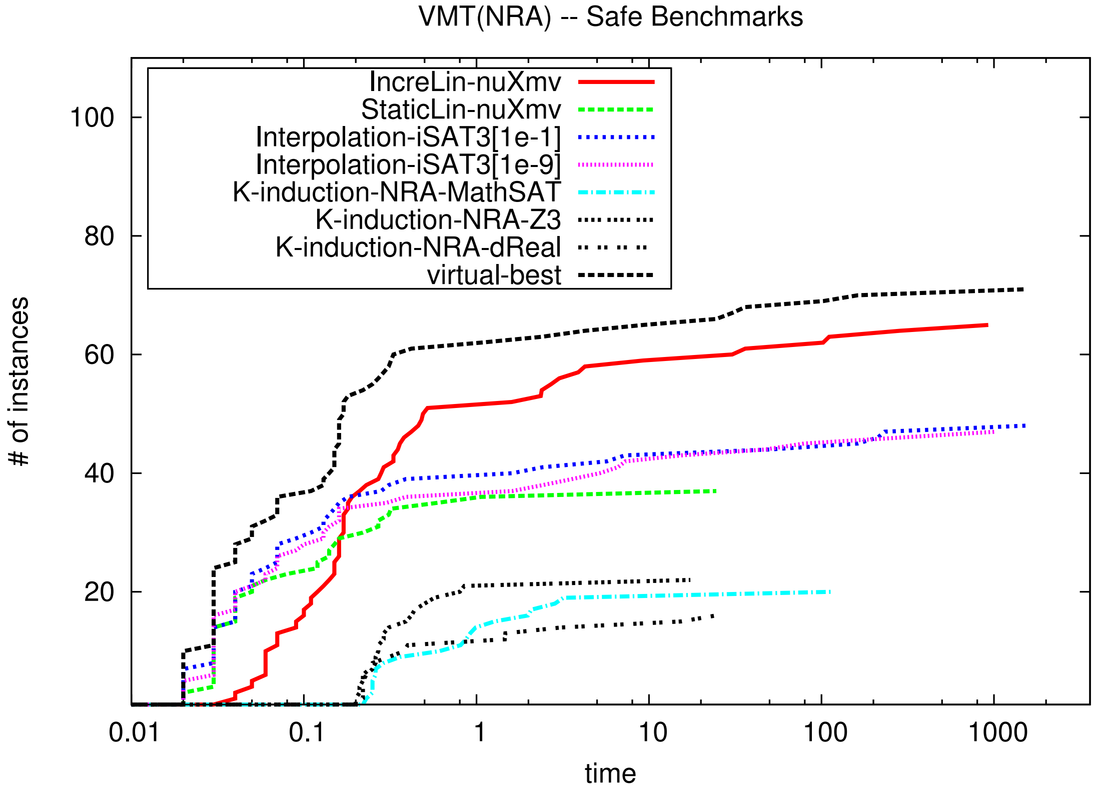
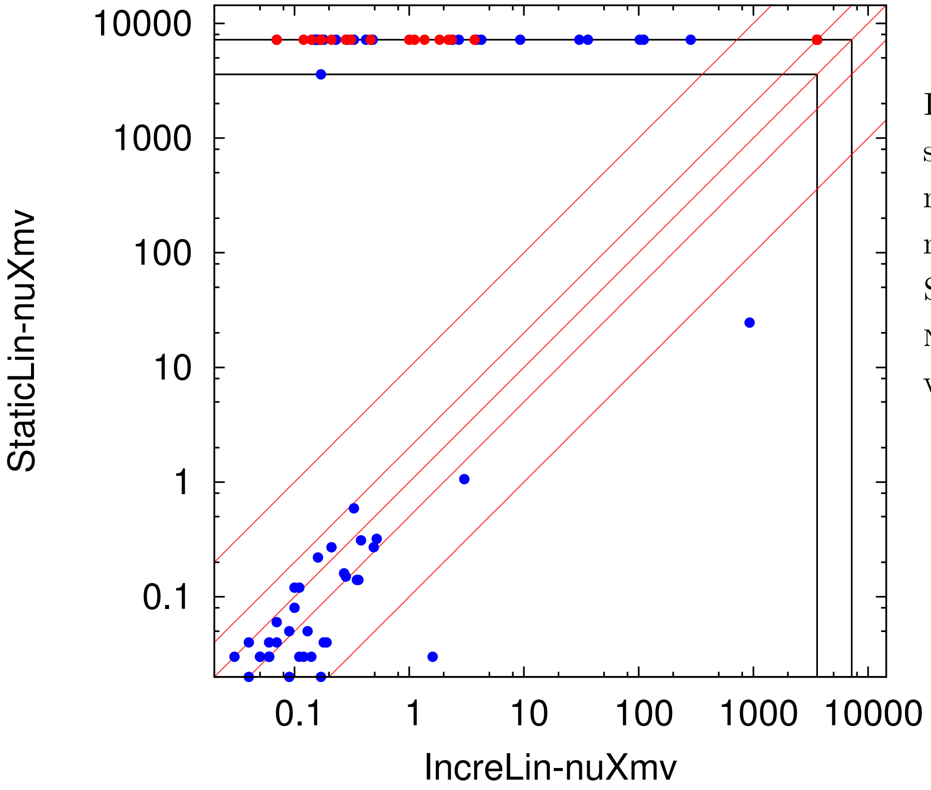
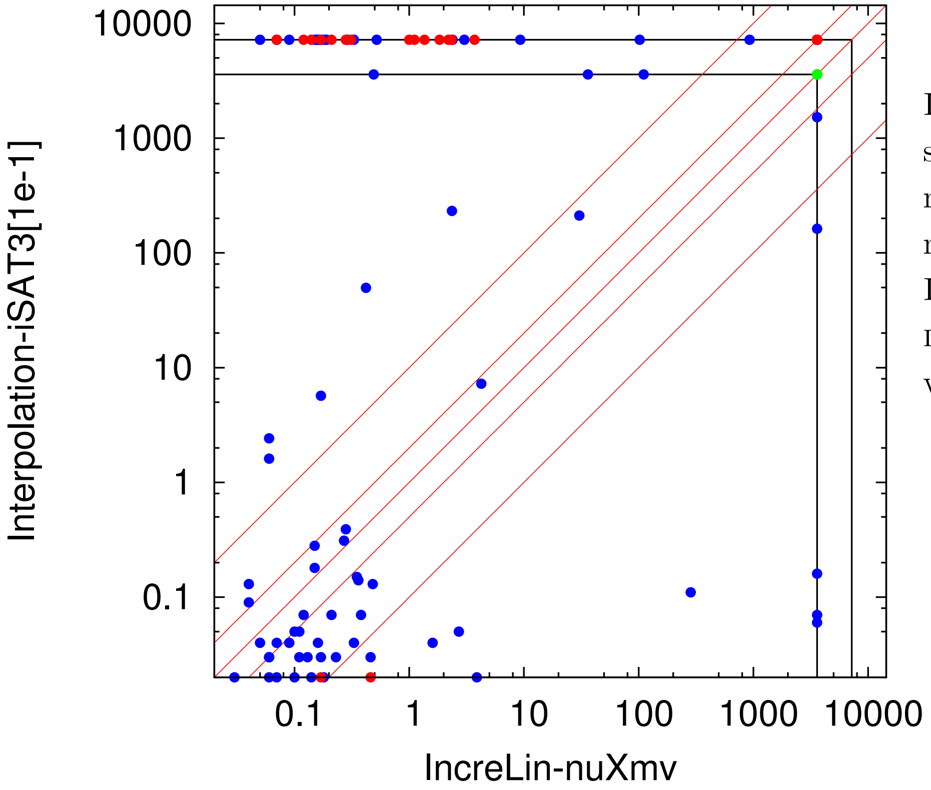


Figure 9.2: Survival plots for VMT($\mathcal{NRA}$) benchmarks

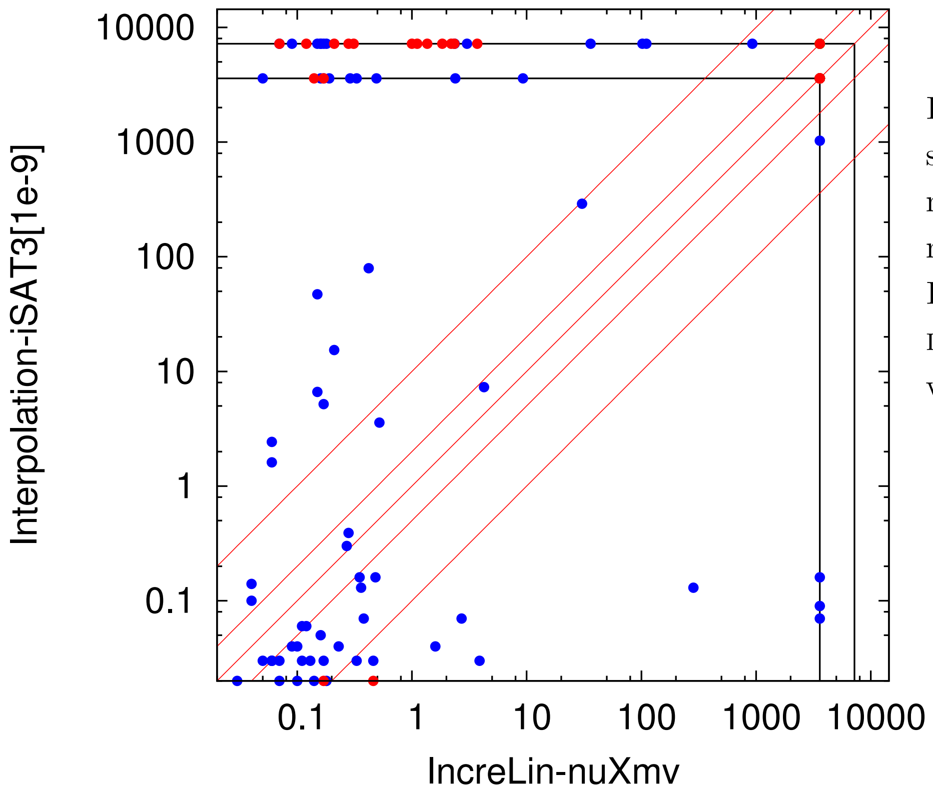


IncreLin-nuXmv solves 44 benchmarks that are not solved by StaticLin-nuXmv, and 0 vice versa.

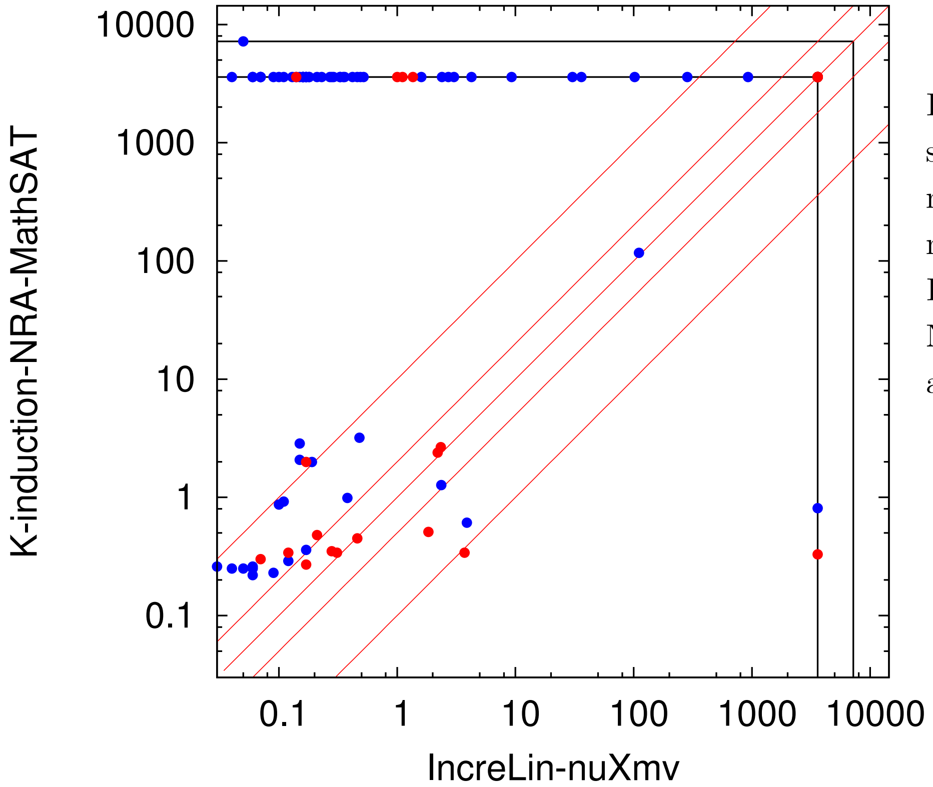


IncreLin-nuXmv solves 36 benchmarks that are not solved by Interpolation-iSAT3$_{[1e-1]}$, and 5 vice versa.

Figure 9.3: Scatters plots of VMT($\mathcal{NRA}$) benchmarks

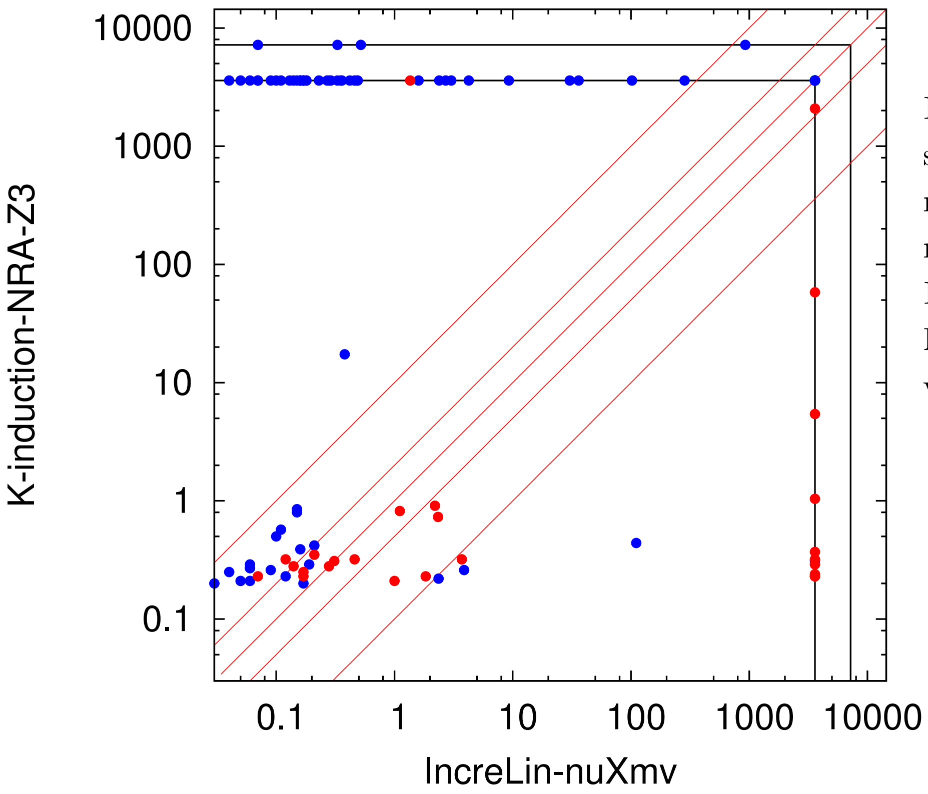


IncreLin-nuXmv solves 36 benchmarks that are not solved by Interpolation-iSAT3$_{[1e-9]}$, and 4 vice versa.

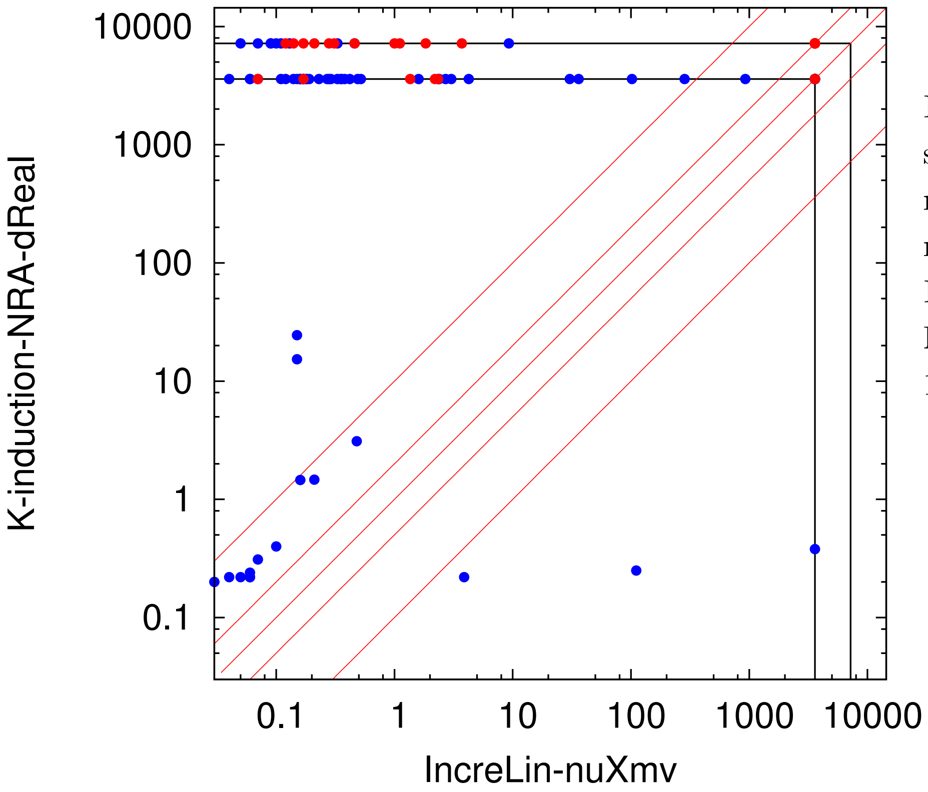


IncreLin-nuXmv solves 50 benchmarks that are not solved by K-induction-NRA-MathSAT, and 2 vice versa.

Figure 9.3: Scatters plots of VMT($\mathcal{NRA}$) benchmarks

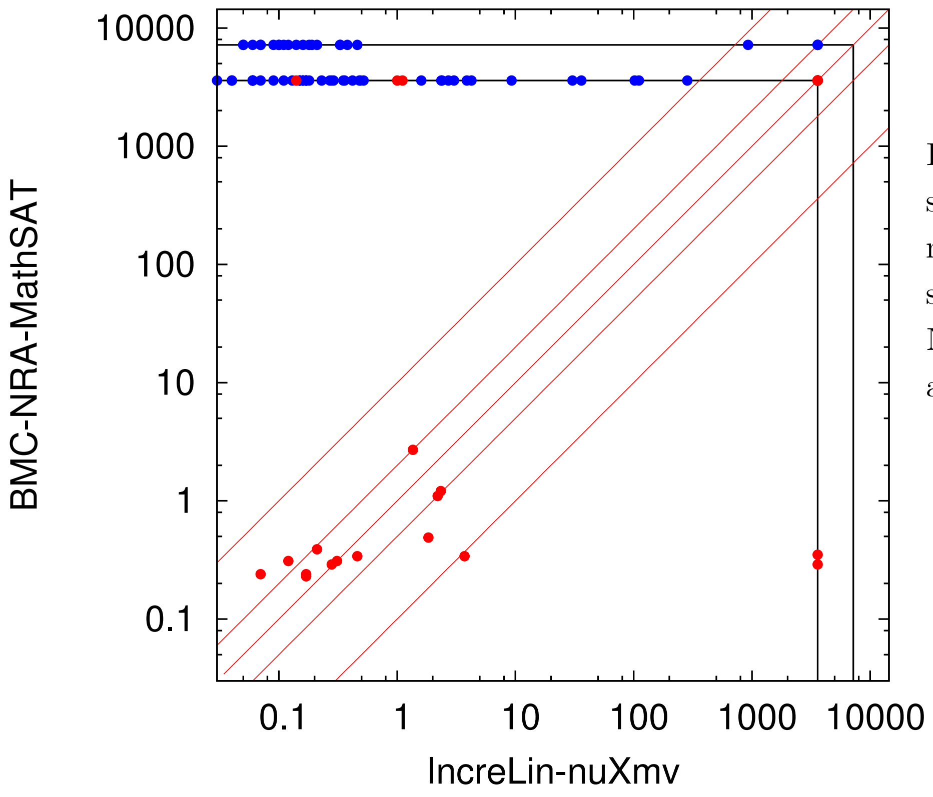


IncreLin-nuXmv solves 45 benchmarks that are not solved by K-induction-NRA-Z3, and 11 vice versa.

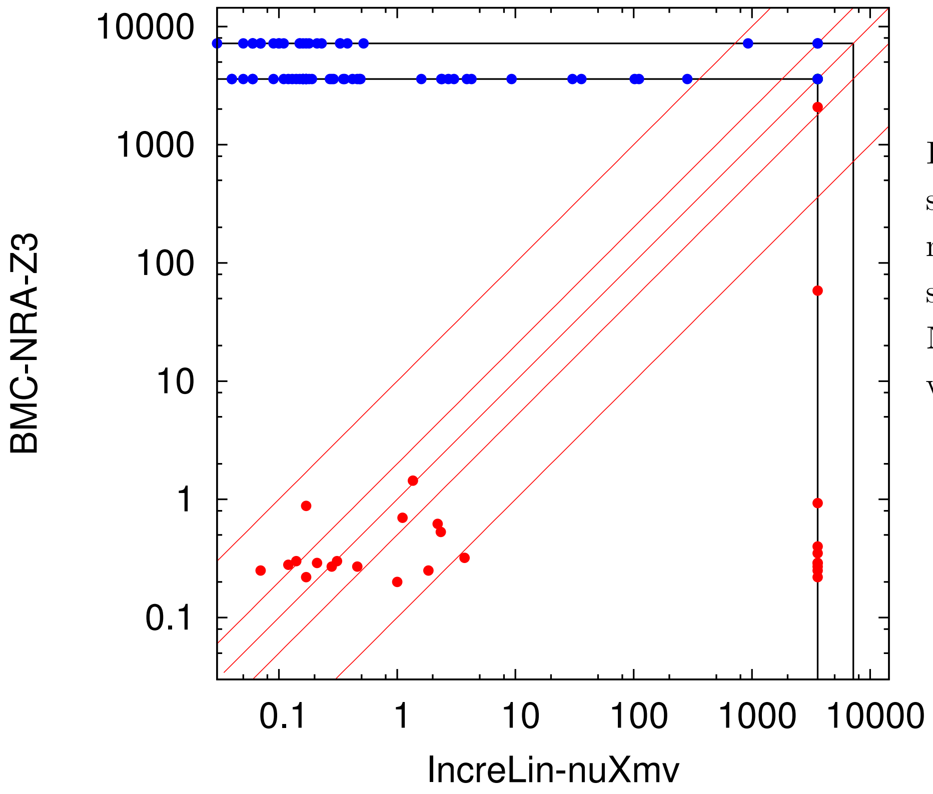


IncreLin-nuXmv solves 66 benchmarks that are not solved by K-induction-NRA-dReal , and 1 vice versa.

Figure 9.3: Scatters plots of VMT($\mathcal{NRA}$) benchmarks

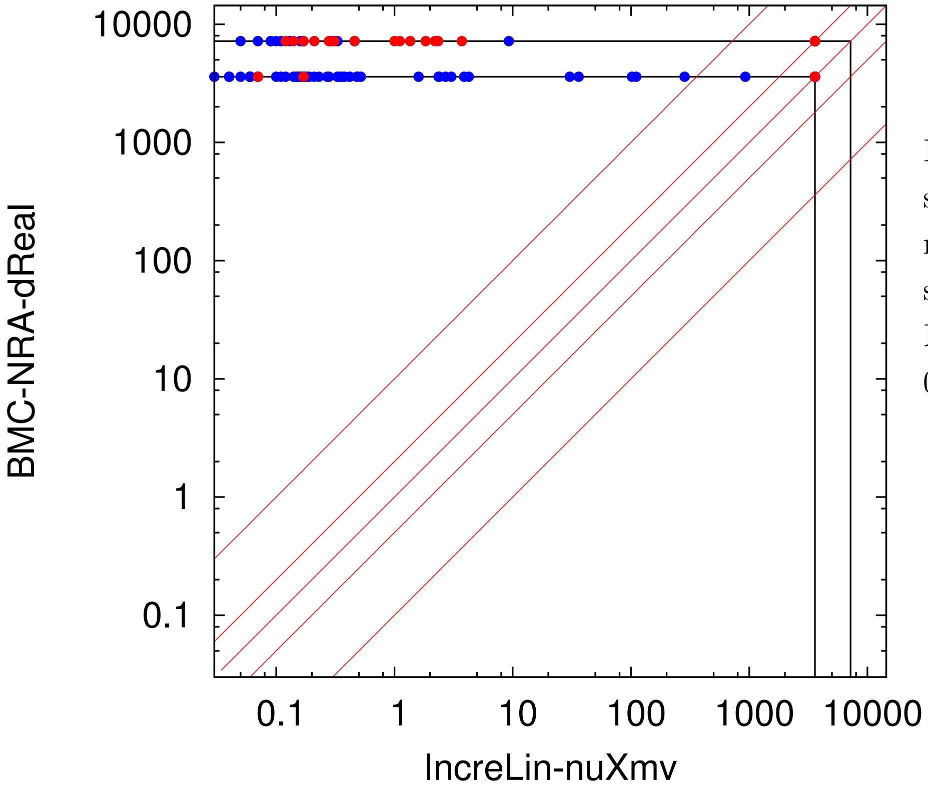


IncreLin-nuXmv solves 68 benchmarks that are not solved by BMC-NRA-MathSAT , and 2 vice versa.



IncreLin-nuXmv solves 65 benchmarks that are not solved by BMC-NRA-Z3, and 10 vice versa.

Figure 9.3: Scatters plots of VMT($\mathcal{NRA}$) benchmarks



IncreLin-nuXmv solves 81 benchmarks that are not solved by BMC-NRA-dReal , and 0 vice versa.

Figure 9.3: Scatters plots of VMT($\mathcal{NRA}$) benchmarks

	Total **(114)**	**Handcrafted** **(14)**	**HyComp** **(7)**	**HYST** **(65)**	**ISAT3** **(1)**	**ISAT3-CFG** **(10)**	**nuXmv** **(2)**	**SAS13** **(13)**	**TCM** **(2)**
IncreLin-nuXmv	16/**65**	**1/13**	1/**3**	7/**34**	0/0	2/**6**	0/**2**	**5/5**	0/**2**
StaticLin-nuXmv	0/37	0/4	0/1	0/19	0/0	0/4	0/**2**	0/**5**	0/**2**
Interpolation-iSAT3$_{[1e-1]}$	2(47)/48	0(8)/2	0(3)/0	2(23)/**34**	0/0	0(4)/**6**	0/0	0(9)/4	0/**2**
Interpolation-iSAT3$_{[1e-9]}$	2(19)/47	0(3)/2	0(2)/0	2(3)/32	0/0	0(3)/**6**	0/0	0(8)/**5**	0/**2**
K-induction-NRA-MathSAT	13/20	**1**/2	0/0	6/12	0/0	1/4	0/0	**5**/0	0/**2**
K-induction-NRA-Z3	25/22	**1**/2	**2**/0	**15**/12	0/0	2/**6**	0/0	**5**/0	0/**2**
K-induction-NRA-dReal	0(32)/16	0(4)/2	0(2)/0	0(19)/9	0/0	0(2)/5	0/0	0(5)/0	0/0
BMC-NRA-MathSAT	15/0	**1**/0	0/0	7/0	0/0	2/0	0/0	**5**/0	0/0
BMC-NRA-Z3	**26**/0	**1**/0	**2**/0	**15**/0	0/0	**3**/0	0/0	**5**/0	0/0
BMC-NRA-dReal	0(39)/0	0(8)/0	0(2)/0	0(19)/0	0/0	0(3)/0	0/0	0(7)/0	0/0
VirtualBest	26/71	1/13	2/3	15/39	0/0	3/7	0/2	5/5	0/2

Table 9.1: Summary of VMT($\mathcal{NRA}$) experimental results

approach based on complete SMT($\mathcal{NRA}$) techniques is the best in terms of unsafe instances. This can be due to the fact that we are adopting a concretization approach that is based on incremental linearization and not a complete solver.

Results for VMT($\mathcal{NTA}$)

The results for VMT($\mathcal{NTA}$) are reported in Table 9.2. Overall, the experimental results clearly demonstrate the merits of incremental linearization at the level of VMT. IncreLin-nuXmv is by far the best solver among all the available ones, confirming the trends of the VMT($\mathcal{NRA}$) case. In fact, incremental linearization dominates even more in VMT($\mathcal{NTA}$), given the lack of complete techniques: the performance of IncreLin-nuXmv is very close to the virtual best (see Table 9.2). The survival plots reported in Fig. 9.4 and 9.6, and the scatters plots reported in Fig. 9.5 and 9.7 corroborate these observations. Compared to VMT($\mathcal{NRA}$), we notice many more memory-outs, suggesting that the solvers may proceed by brute force and generate useless lemmas.

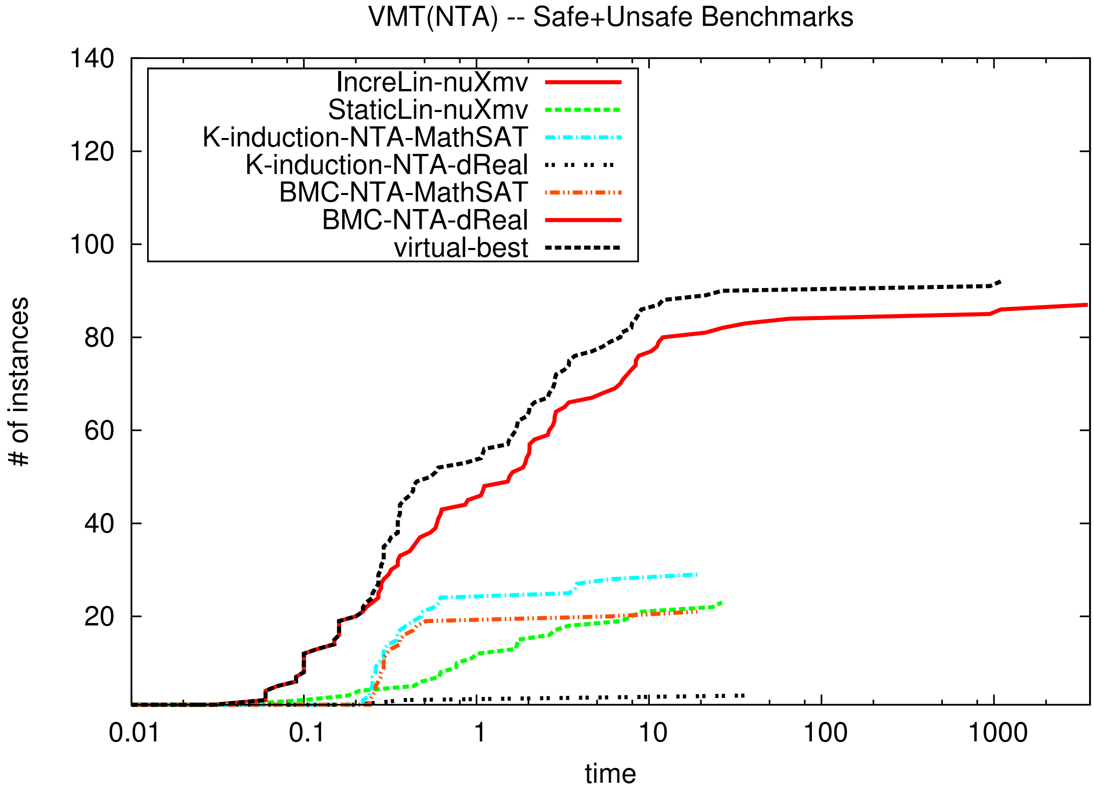
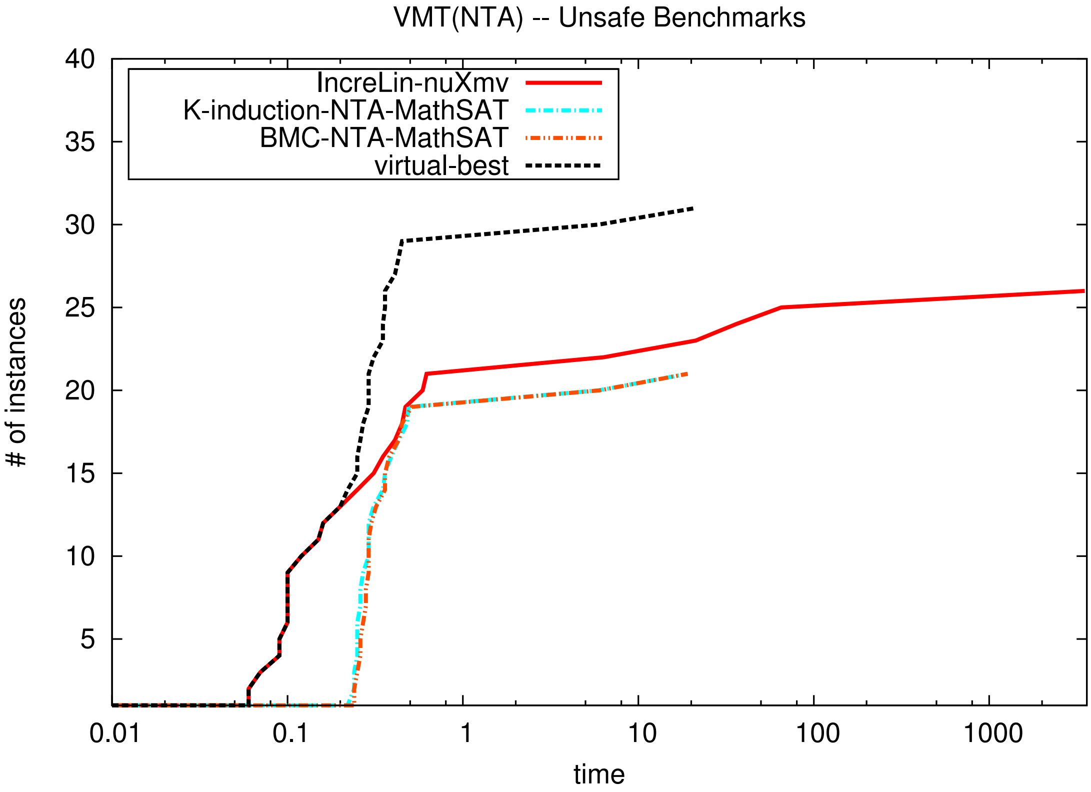


Figure 9.4: Survival plots for VMT($\mathcal{NTA}$) – unbounded benchmarks



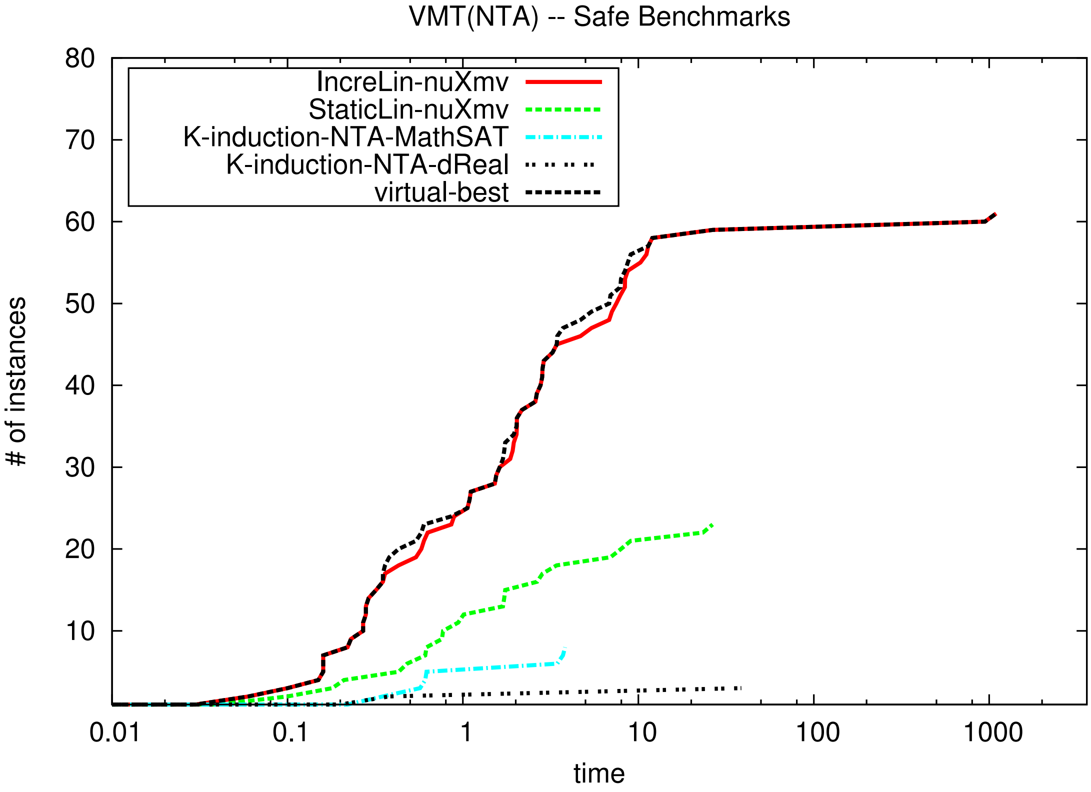
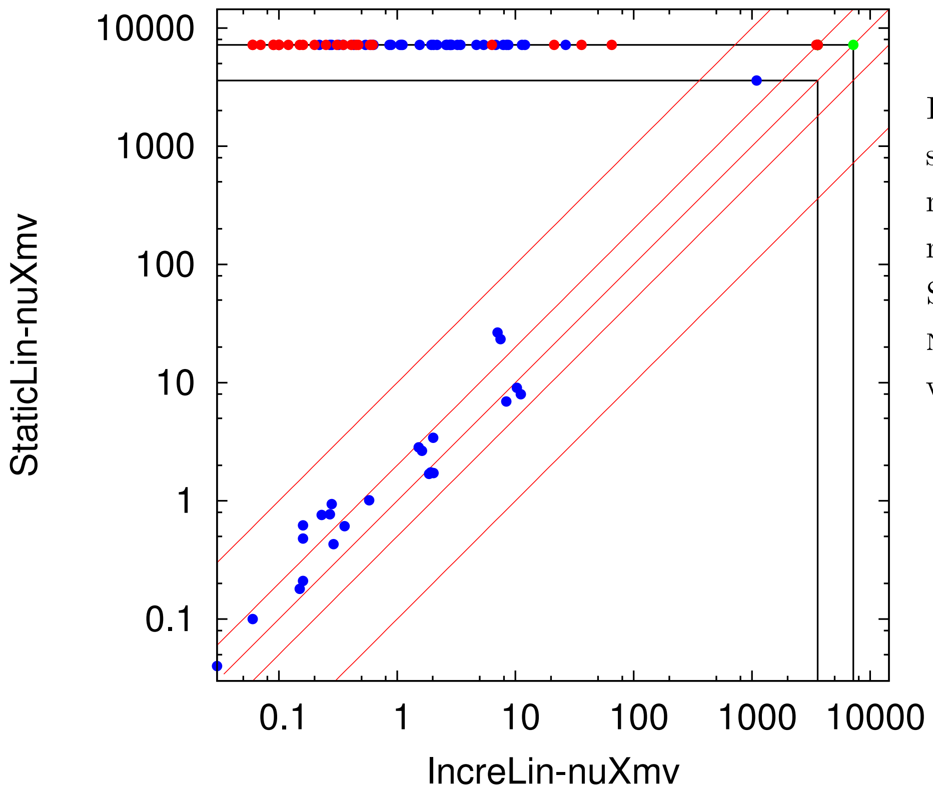
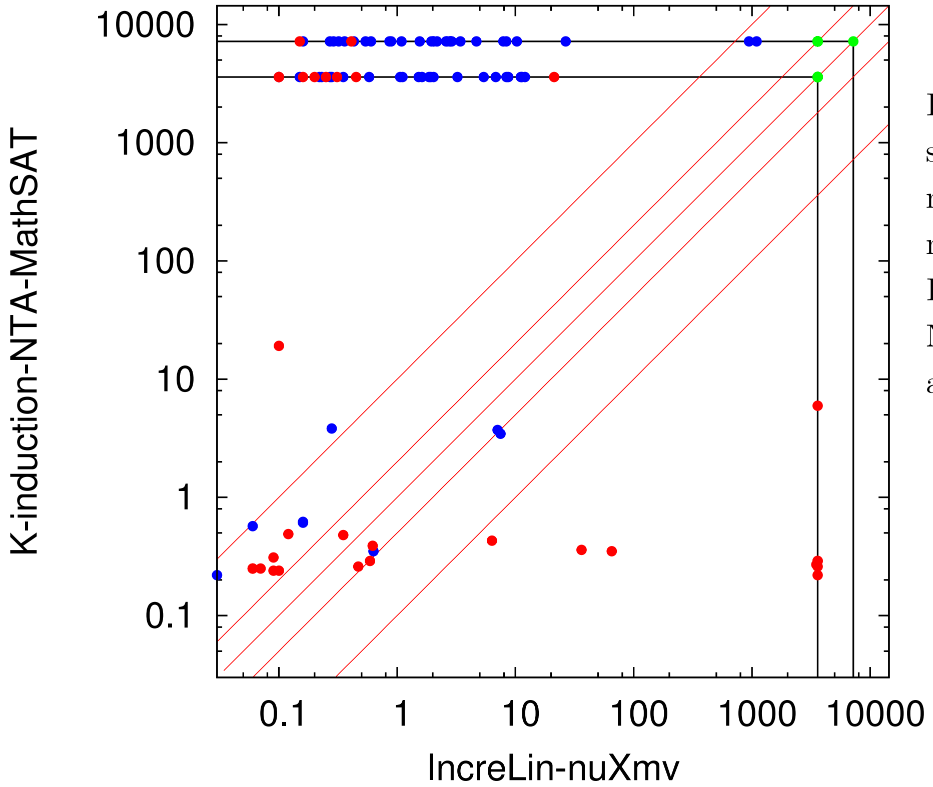


Figure 9.4: Survival plots for VMT($\mathcal{NTA}$) – unbounded benchmarks

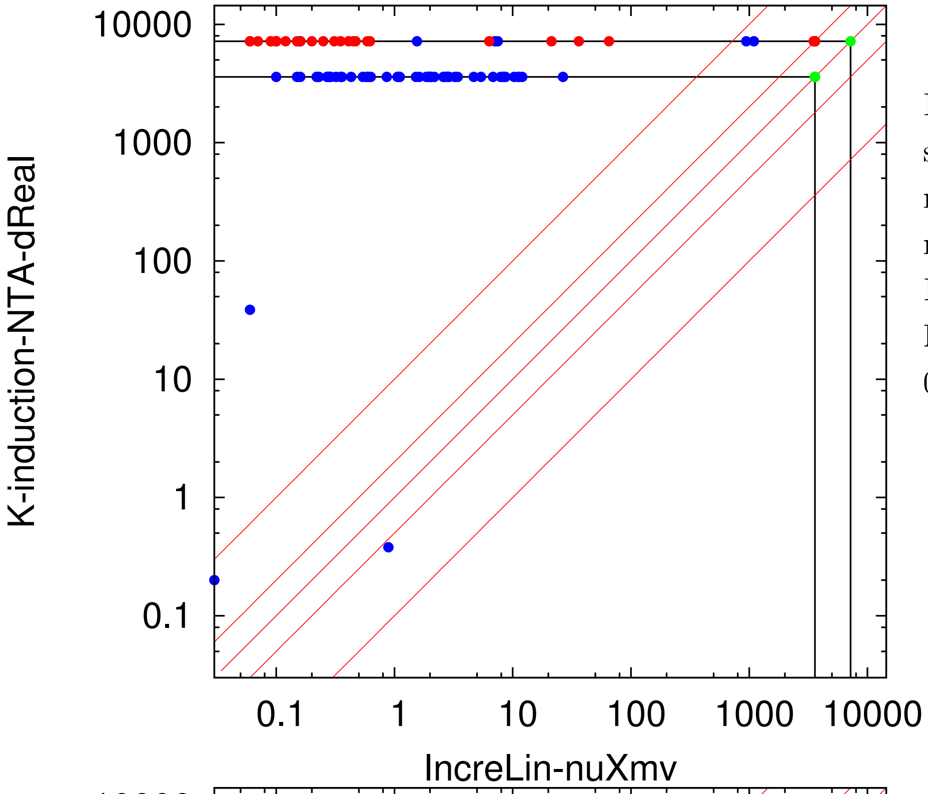


IncreLin-nuXmv solves 64 benchmarks that are not solved by StaticLin-nuXmv, and 0 vice versa.

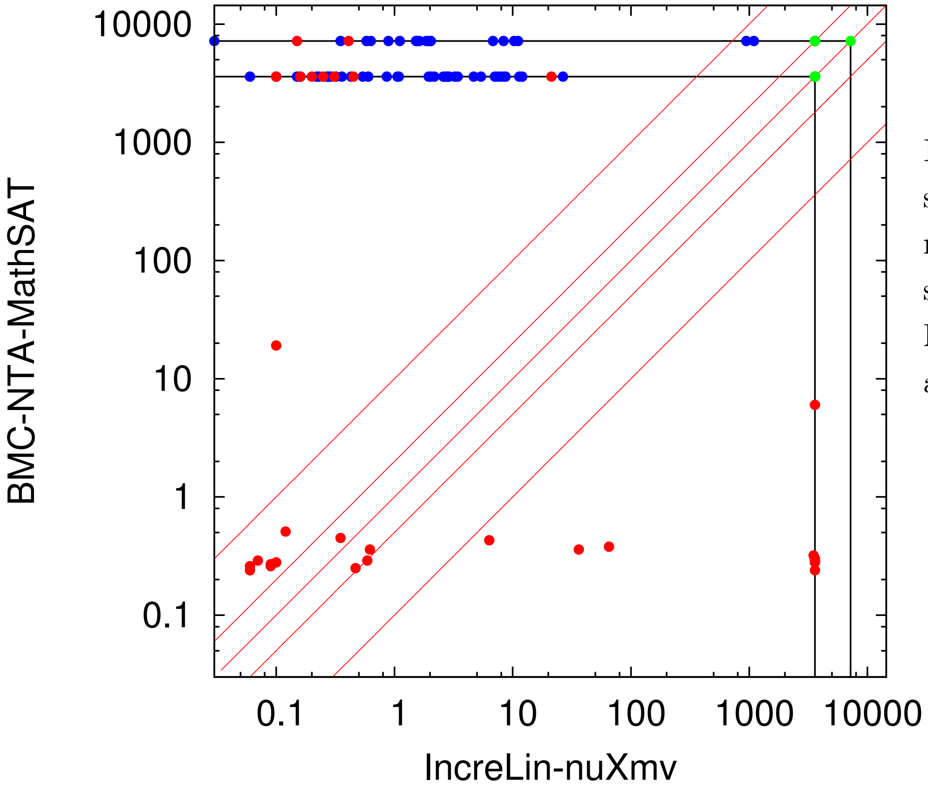


IncreLin-nuXmv solves 63 benchmarks that are not solved by K-induction-NTA-MathSAT, and 5 vice versa.

Figure 9.5: Scatters plots of VMT($\mathcal{NTA}$) – unbounded benchmarks

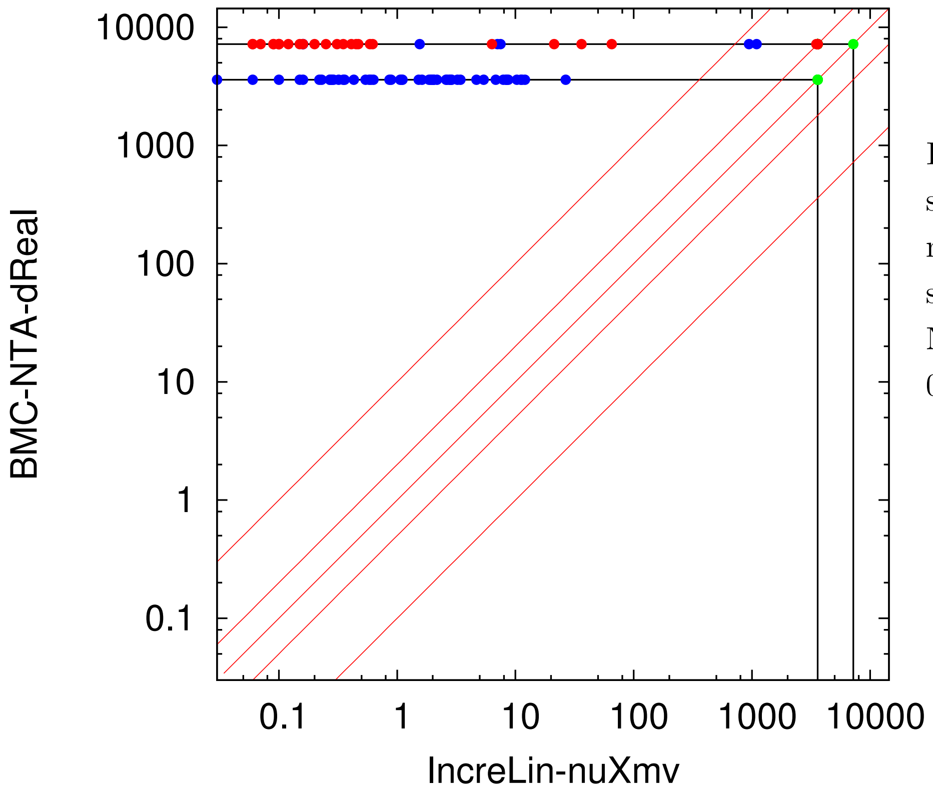


IncreLin-nuXmv solves 84 benchmarks that are not solved by K-induction-NTA-dReal, and 0 vice versa.



IncreLin-nuXmv solves 71 benchmarks that are not solved by BMC-NTA-MathSAT, and 5 vice versa.

Figure 9.5: Scatters plots of VMT($\mathcal{NTA}$) – unbounded benchmarks



IncreLin-nuXmv solves 87 benchmarks that are not solved by BMC-NTA-dReal, and 0 vice versa.

Figure 9.5: Scatters plots of VMT($\mathcal{NTA}$) – unbounded benchmarks

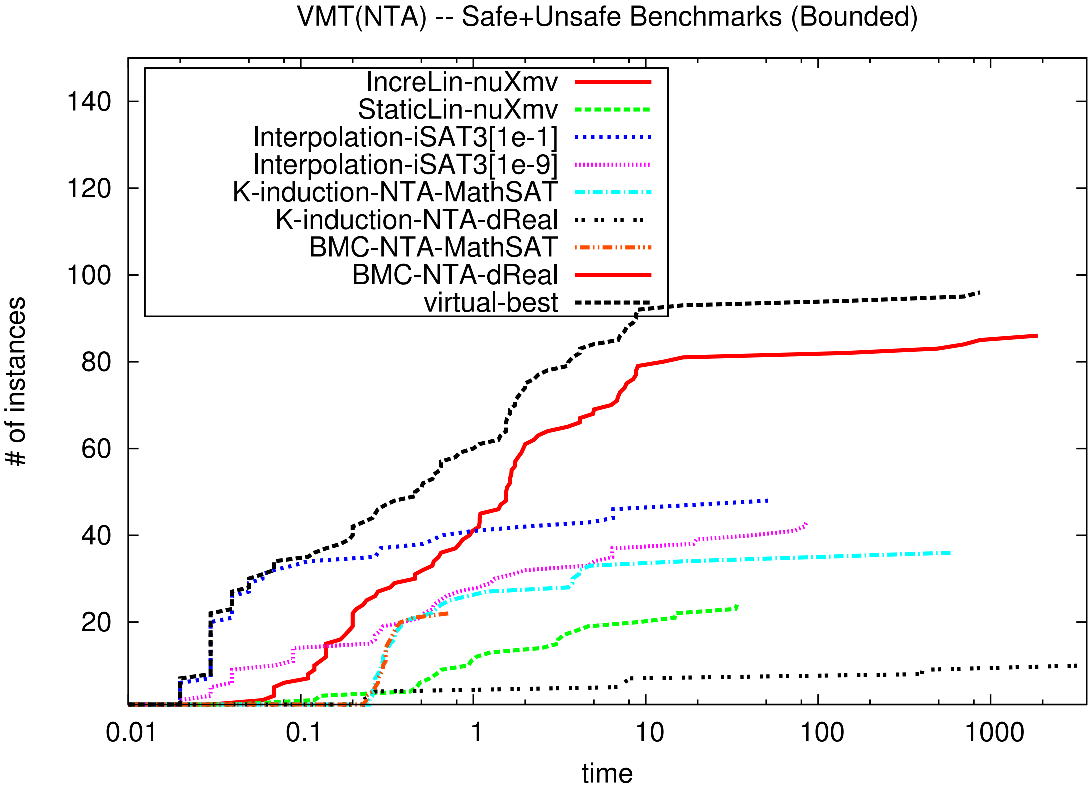
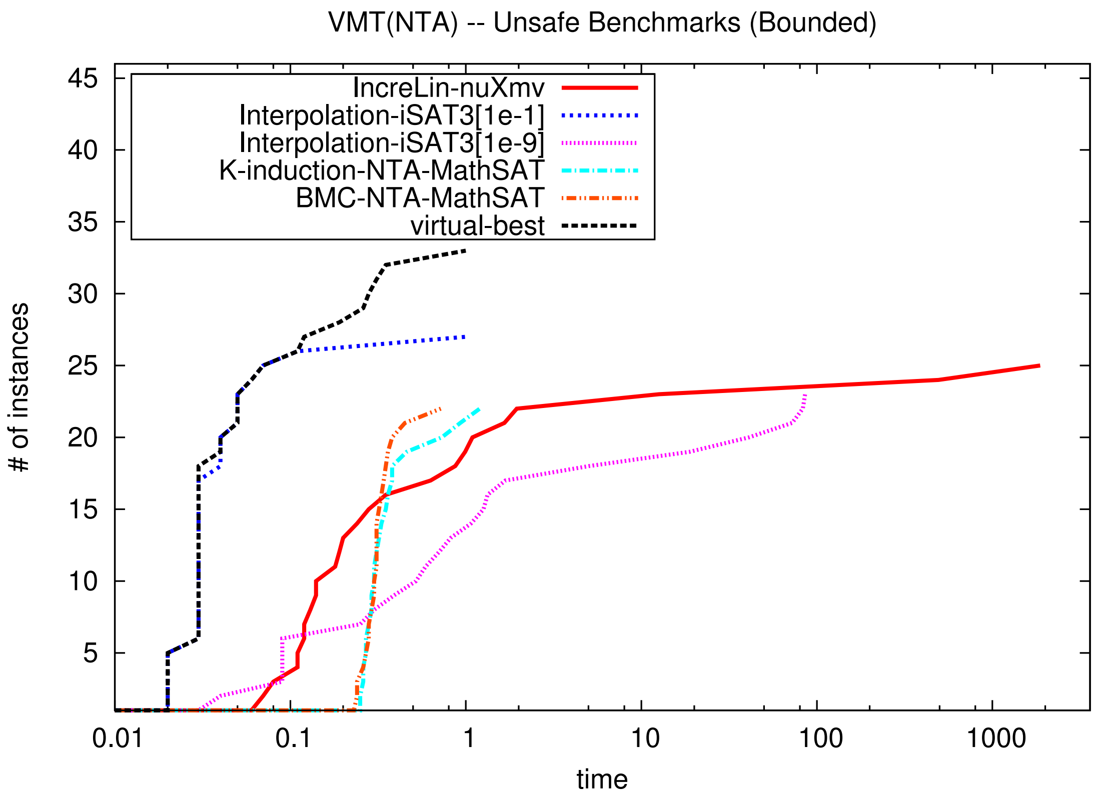


Figure 9.6: Survival plots for VMT($\mathcal{NTA}$) – bounded benchmarks



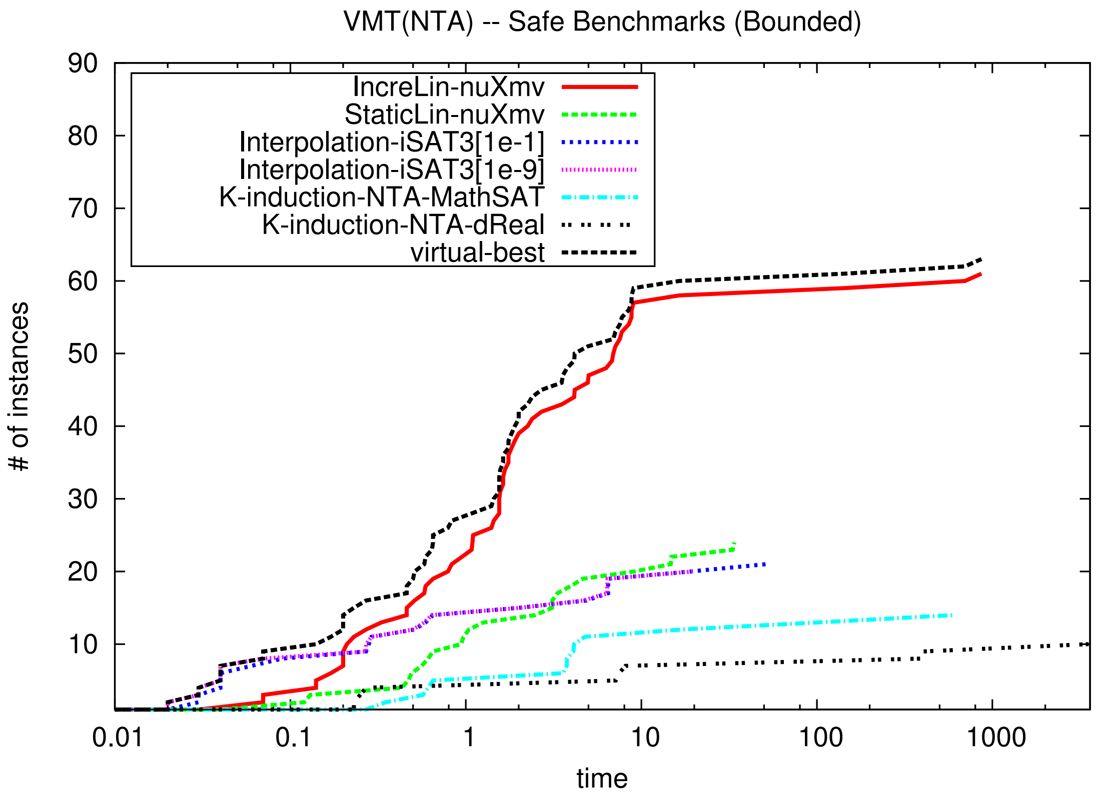
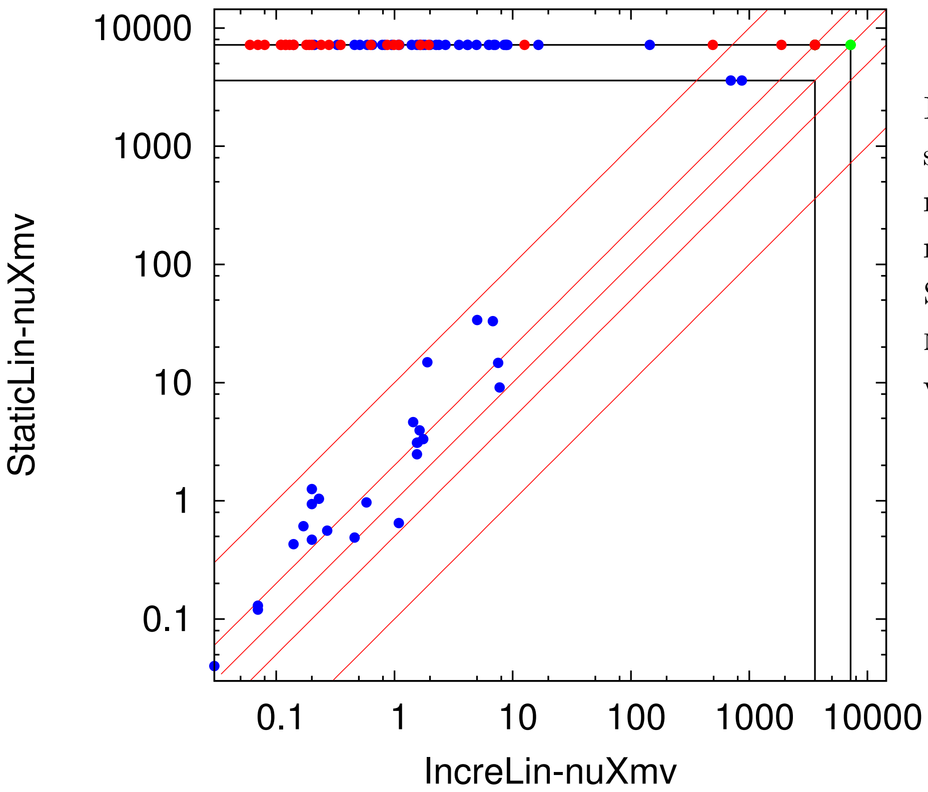
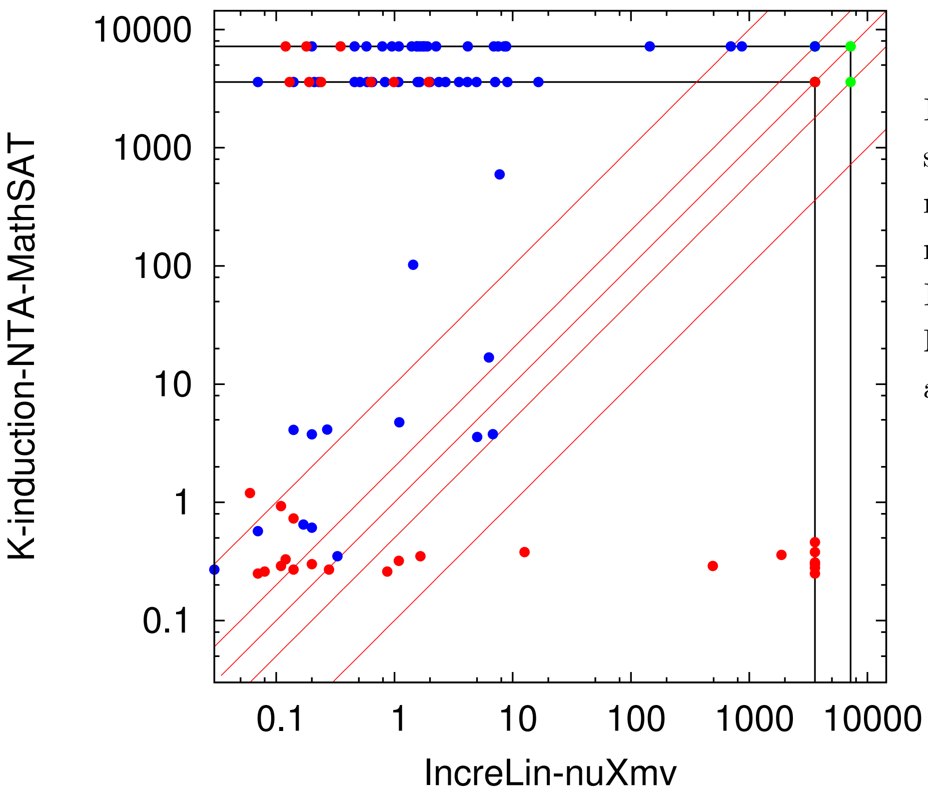


Figure 9.6: Survival plots for VMT($\mathcal{NTA}$) – bounded benchmarks

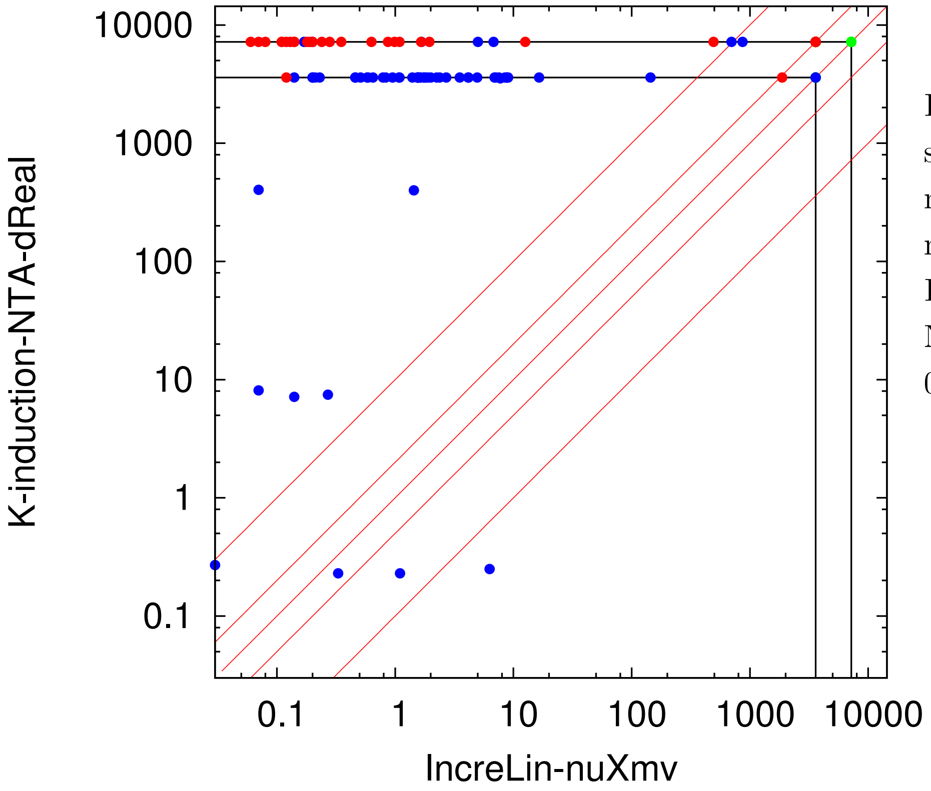


IncreLin-nuXmv solves 62 benchmarks that are not solved by StaticLin-nuXmv, and 0 vice versa.

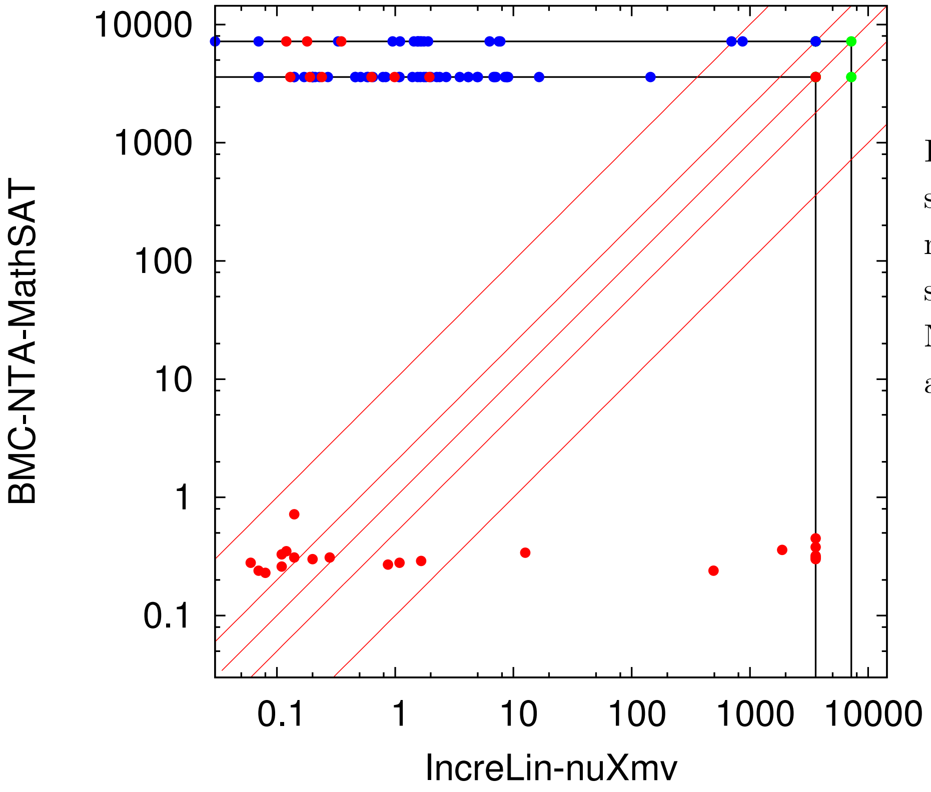


IncreLin-nuXmv solves 56 benchmarks that are not solved by K-induction-NTA-MathSAT, and 6 vice versa.

Figure 9.7: Scatters plots of VMT($\mathcal{NTA}$) – bounded benchmarks

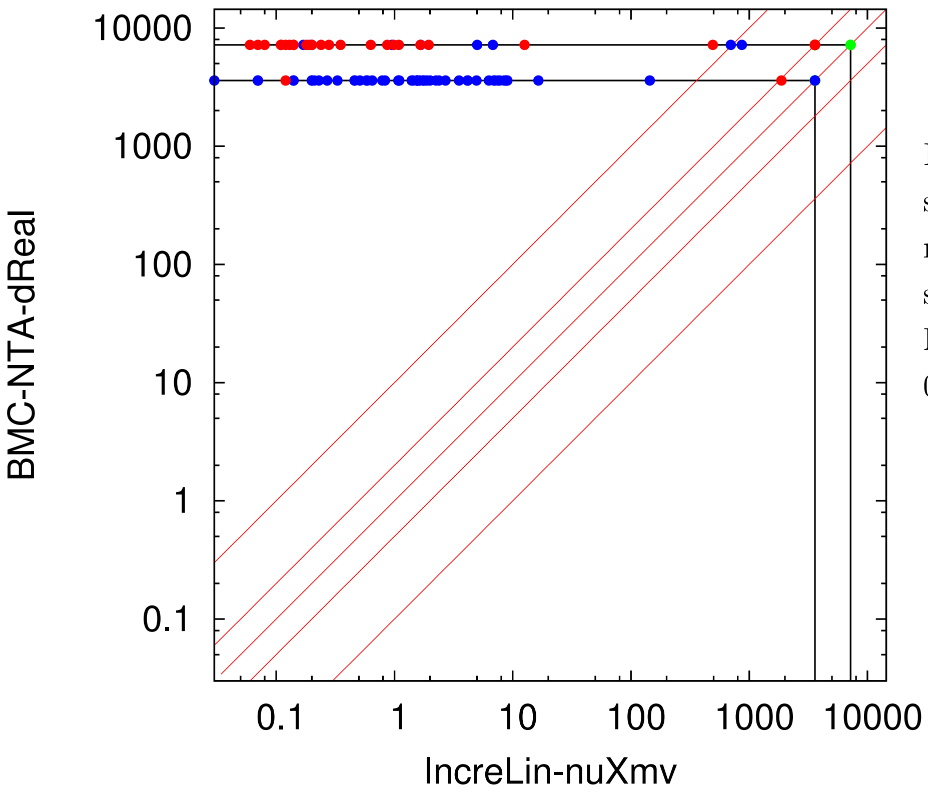


IncreLin-nuXmv solves 76 benchmarks that are not solved by K-induction-NTA-dReal, and 0 vice versa.

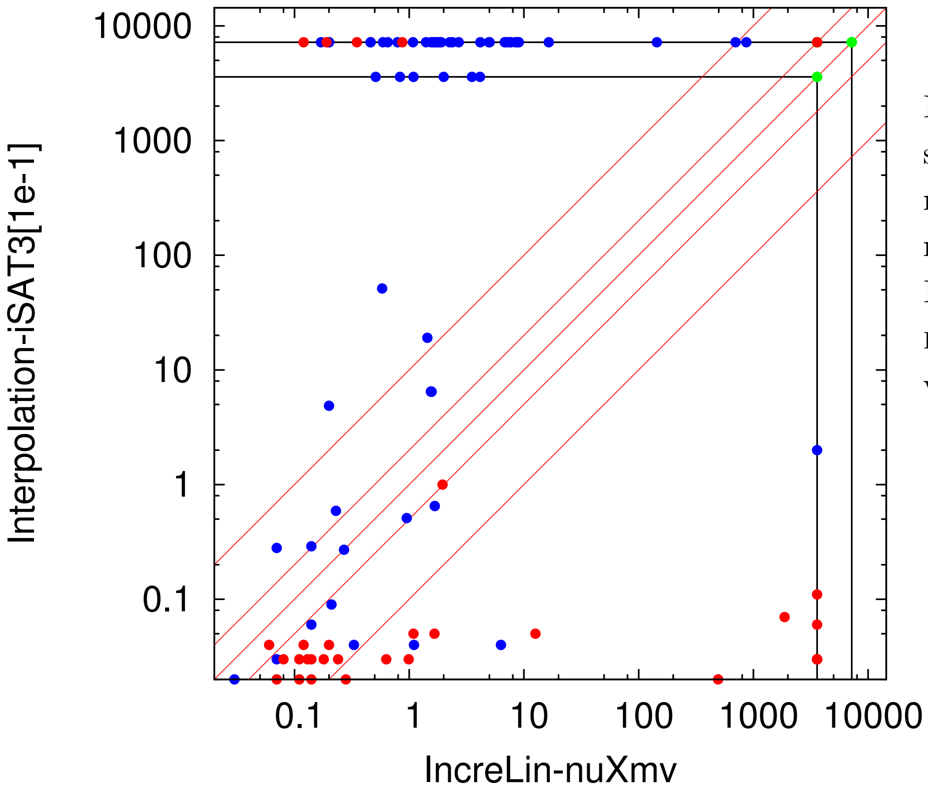


IncreLin-nuXmv solves 70 benchmarks that are not solved by BMC-NTA-MathSAT, and 6 vice versa.

Figure 9.7: Scatters plots of VMT($\mathcal{NTA}$) – bounded benchmarks

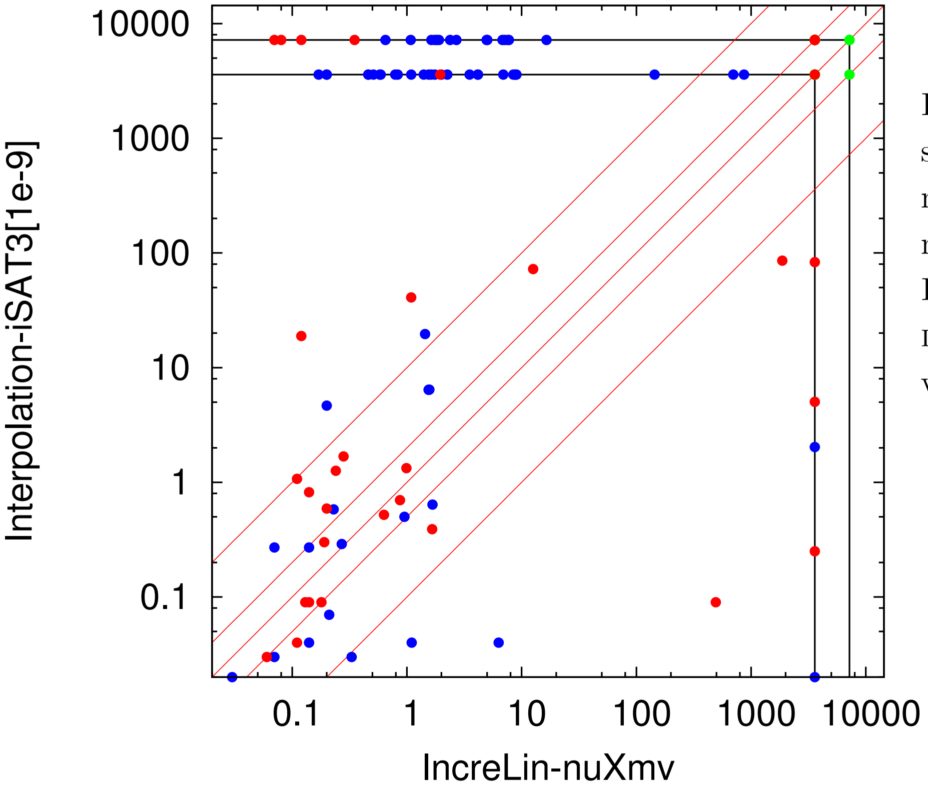


IncreLin-nuXmv solves 86 benchmarks that are not solved by BMC-NTA-dReal, and 0 vice versa.



IncreLin-nuXmv solves 46 benchmarks that are not solved by Interpolation-iSAT3$_{[1e-1]}$, and 8 vice versa.

Figure 9.7: Scatters plots of VMT($\mathcal{NTA}$) – bounded benchmarks



IncreLin-nuXmv solves 48 benchmarks that are not solved by Interpolation-iSAT3$_{[1e-9]}$, and 5 vice versa.

Figure 9.7: Scatters plots of VMT($\mathcal{NTA}$) – bounded benchmarks

	Total **(126)**	**Handcrafted** **(3)**	**HARE** **(50)**	**HYST** **(57)**	**ISAT3** **(4)**	**WBS** **(12)**
IncreLin-nuXmv	**26/61**	0/**3**	0/**43**	**24/4**	0/**2**	**2/9**
StaticLin-nuXmv	0/23	0/0	0/15	0/1	0/0	0/7
K-induction-NTA-MathSAT	21/8	0/1	0/0	20/1	0/0	1/6
K-induction-NTA-dReal	0(66)/3	0(0)/0	0(3)/0	0(54)/2	0(0)/0	0(9)/1
BMC-NTA-MathSAT	21/0	0/0	0/0	19/0	0/0	1/0
BMC-NTA-dReal	0(67)/0	0(0)/0	0(4)/0	0(54)/0	0(0)/0	0(9)/0
VirtualBest	31/61	0/3	0/43	29/4	0/2	2/9
IncreLin-nuXmv	25/**61**	0/**3**	0/**43**	23/4	0/2	**2/9**
StaticLin-nuXmv	0/24	0/0	0/15	0/1	0/1	0/7
Interpolation-iSAT3$_{[1e-1]}$	**27**(36)/21	0(0)/**3**	0(4)/8	**27**(21)/**5**	0(1)/**3**	0(10)/2
Interpolation-iSAT3$_{[1e-9]}$	23(27)/20	0(0)/**3**	0(2)/7	23(19)/**5**	0(0)/**3**	0(6)/2
K-induction-NTA-MathSAT	22/14	0/**3**	0/4	22/1	0/0	0/6
K-induction-NTA-dReal	0(55)/10	0(0)/**3**	0(4)/4	0(42)/1	0(0)/1	0(9)/1
BMC-NTA-MathSAT	22/0	0/0	0/0	22/0	0/0	0/0
BMC-NTA-dReal	0(55)/0	0(0)/0	0(4)/0	0(42)/0	0(0)/0	0(9)/0
VirtualBest	33/63	0/3	0/43	31/5	0/3	2/9

Table 9.2: Summary of VMT($\mathcal{NTA}$) experimental results (original problems above, bounded version below)

Summary

In this part, we tackled the VMT problem (invariant checking) with respect to the theories of nonlinear arithmetic and transcendental functions. We lifted the idea of incremental linearization from the SMT level to the VMT level. The main underlying idea is to over-approximate the transition system by abstracting nonlinear and transcendental functions as uninterpreted functions. The uninterpreted functions corresponding to the multiplication function and transcendental functions are axiomatized incrementally, driven by sprurious counterexample traces.

We have proved the correctness of the approach. The approach is now tightly integrated in the nuXmv VMT model checker. Moreover, using incremental linearization, we provide a robust IC3-based verification method for $\mathcal{NRA}$ and $\mathcal{NTA}$. We have evaluated our approach on benchmarks coming from various practical domains and compared against other available tools. The experimental results clearly show that the strength of our approach. From the results, we can also make the following remarks:

- Incremental linearization advances the state of the art in VMT($\mathcal{NRA}$) and VMT($\mathcal{NTA}$).

- Incremental linearization shows complete dominance in the safe benchmarks (both in the case of $\mathcal{NRA}$ and $\mathcal{NTA}$) – clearly, nuXmv is very close the virtual best solver in the survival plots.

- The satisfiability detecting techniques presented for incremental linearization perform reasonably well in the case of VMT($\mathcal{NRA}$), and even the strongest in VMT($\mathcal{NTA}$).

Part IV

Verification in Systems Design Automation

Overview

One of the primary motivations of the work we presented earlier in Part II and III is to improve the capabilities of system design verification. Research in this area has focused on two orthogonal directions: verification techniques and design methodologies. Often, there is a gap between the two. For instance, SIMULINK designs can have nonlinear behavior, but the main verification tool, i.e., SIMULINK DESIGN VERIFIER cannot check them. In this part, we focus on integrating our proposed verification technique and other state-of-the-art verification methods in design tools – thus, attempting to bridge the gap.

We present approaches to integrate NUXMV with SIMULINK and with VERILOG. We chose SIMULINK and VERILOG as they are widely used in industry. Moreover, each of them represents a different class of tools from the perspective of the verification engine integration.

We follow a compilation-based approach, in which a compiler is used to convert the input model into something that can be easily fed to a verification engine (via some transformations). This is a typical flow to connect verification backends. However, developing a compiler that can handle the design language features and semantic complexities is a tough challenge. We tackle it by utilizing the code generation capabilities (one of the benefits of model-based design approaches) of the design tools. In case of SIMULINK, C code can be generated via a black-box compiler – the internal structure of the compiler is not accessible. For the VERILOG case, several industrial and academic synthesizers are available that are mostly white-box compilers – the internal structure of the compiler is available.

Our contributions are the following:

- Integration of nuXmv with Simulink – using a black-box compiler (Chapter 10). We give the first completely automatic approach to verify invariant properties of discrete-time synchronous Simulink designs with nonlinear dynamics.
- Integration of nuXmv with Verilog – using a white-box compiler (Chapter 11). Our work also allows to generate verification benchmarks at a high-level of abstraction from Verilog designs.
- Implementation and experimental evaluation of the approaches (Chapter 12) – Simulink to nuXmv is evaluated on an industrial-level Simulink model containing nonlinear behavior and Verilog to nuXmv is compared against other Verilog verification tools on a collection of Verilog benchmarks.

Remark 7. The ideas presented here can also be used for integrating any infinite-state synchronous model checker.

Chapter 10

Simulink to nuXmv

Simulink models can be classified based on the blocks types, timing properties, and the properties of the simulation solver.

Discrete vs. Continuous vs. Hybrid. A Simulink model is a *discrete model* if it does not contain any continuous type block. A discrete model can have *single-rate sampling* or *multirate sampling* (model whose blocks have different sampling rate). A model is *continuous* if it contains continuous blocks but no discrete block. A model that uses both discrete and continuous type blocks is called *hybrid model.*

Simulation Solver. A simulation solver defines how the Simulink model is solved for simulating the model dynamics. In a way, choosing a solver tells what simulation semantics will be followed for the given Simulink model. Solvers are broadly classified using the following criteria:

- The type of *step size* used in the computation, also shown in Fig. 10.1.
  - Fixed-step solvers solve the model at step sizes (constant) for a given simulation time.
  - Variable-step solvers can vary the step sizes during the simulation.

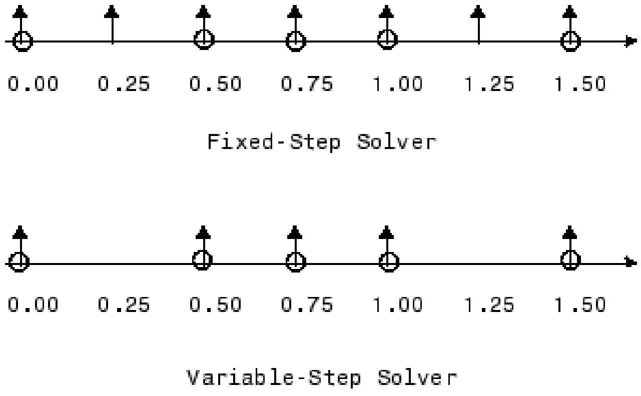
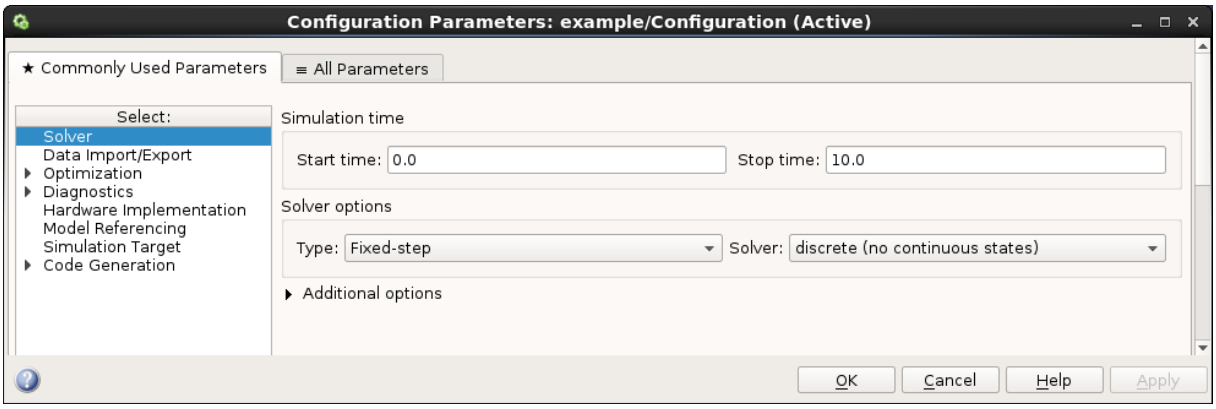


Figure 10.1: Difference between fixed-step and variable-step [sima]



Configuration Parameters: example/Configuration (Active)

Simulation time

Start time: 0.0 Stop time: 10.0

Solver options

Type: Fixed-step Solver: discrete (no continuous states)

Additional options

Figure 10.2: SIMULINK simulation solver configuration

- The *nature* of the solver (see Fig. 10.2).
  - *Discrete solvers* are used for solving discrete models.
  - *Continuous solvers* compute the continuous states of a model by using numerical methods. They are used for simulating continuous or hybrid models.

10.1 Problem Definition

We address the problem of formally verifying invariant properties of SIMULINK models. There are some challenges in dealing with SIMULINK models:

Challenge I. Simulink is used for modeling and simulating various kinds of systems. For that purpose, in addition to the built-in blocks library, it also offers several add-on products with extended blocks library – Stateflow being one of them. These extra add-on products add complexity in handling Simulink models.

Challenge II. Simulink does not have a standard formal semantics; instead, it is given informally as "what the simulator does."

Challenge III. There is also a challenge of maintaining the tool for the future releases of Simulink: it can be the case that the functionality (simulation semantics) of some block is changed.

Environment Model and Assumptions

We focus on Simulink models that use built-in Simulink blocks library, Stateflow extended add-on, and custom blocks built using the former sets. Within those Simulink models, we restrict the input problem to a particular class that is:

- discrete model with
- fixed-step simulation and
- single-rate sampling for each block.

For the Stateflow and Matlab-code blocks, we assume that there are no infinite loops. Moreover, the bounds of the loops are known apriori. Regarding the semantics issue, we follow the simulation semantics of Simulink and focus on discrete systems with one global clock. In this way, we address the first two challenges. About the last challenge, we will see that our approach is indeed simpler to maintain.

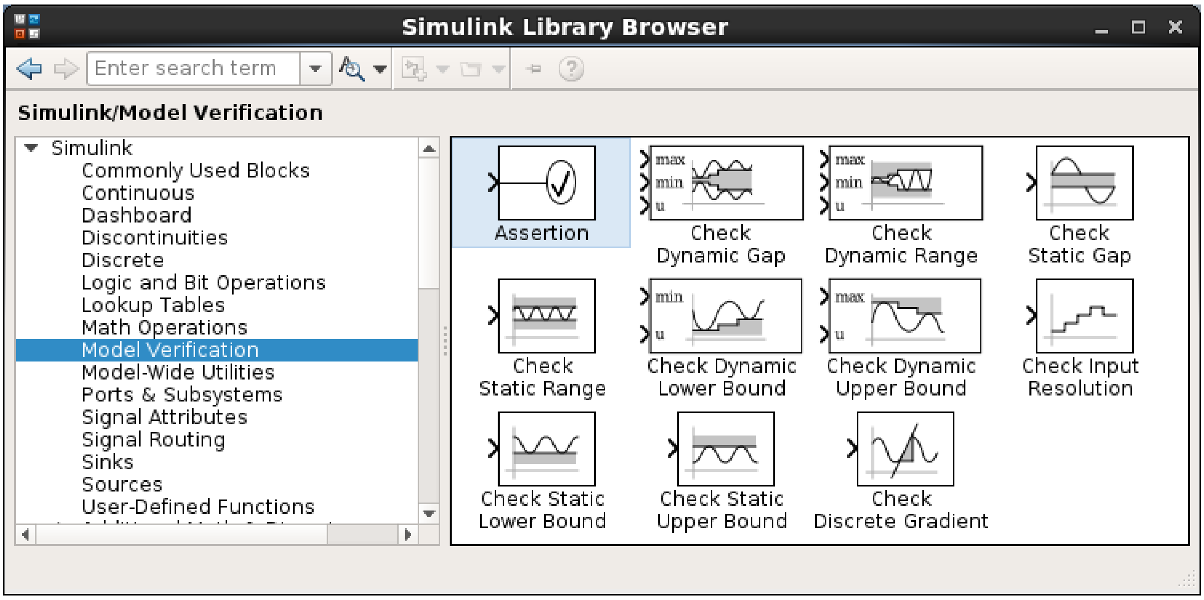


Figure 10.3: SIMULINK assertion block

The SIMULINK assertion block (see Fig. 10.3) is used to specify an invariant property in the SIMULINK model.

10.2 Proposed Solution

In this section, we present our proposed solution: The solution has three main phases: (i) *forward phase*, (ii) *analysis phase*, (iii) *backward phase*. The forward phase is about how the translation from SIMULINK to NUXMV is performed. In the analysis phase, the translated model is analyzed using the NUXMV VMT model checker. The backward phase covers how the result of the analysis is reported back to the MATLAB environment. In particular, the counterexample found by the analysis tool is used to create a SIMULINK test case which can be simulated within the MATLAB environment.

The approach uses the SIMULINK code generation technology, specifically SIMULINK EMBEDDED CODER (a black-box compiler), to generate C code from SIMULINK models. For translating the C program to NUXMV,

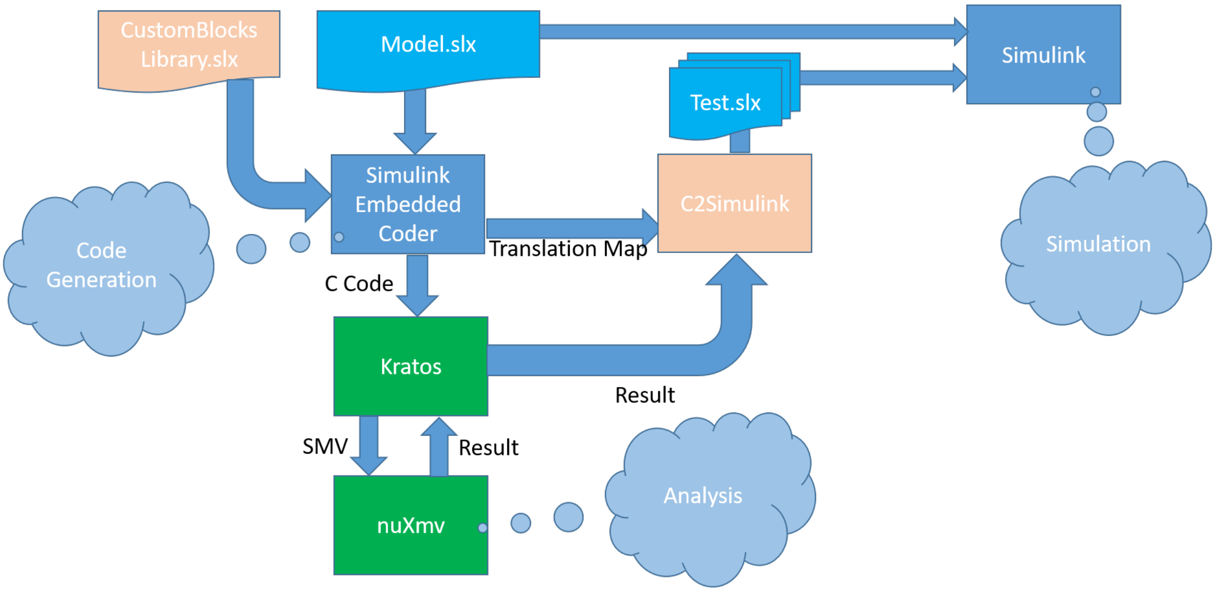


Figure 10.4: SIMULINK to SMV via EMBEDDED CODER– detailed flow

we use the KRATOS. KRATOS [CGM$^+$11] is a software model checker for sequential and (cooperative) threaded C programs. It is used to parse the code, apply some code-to-code transformations, and output a NUXMV program. The NUXMV program is then analyzed using the NUXMV model checker. When NUXMV finds a counterexample, it can be used to generate a test case SIMULINK model. In turn, the test case can then be simulated using a fixed-step simulation solver provided with the SIMULINK package. The complete flow of the approach is shown in Fig. 10.4.

The generated C code from EMBEDDED CODER has a particular form, shown in Fig. 10.5. The `init()` function in the C code can be interpreted as a formula specifying the initial value of the state variables, and similarly, the `step()` function in the code can be seen as a formula over current and next state variables specifying the transition relation of the model. For the translation, we rely on single static assignment (SSA) transformations as provided by KRATOS. Assertions (i.e., properties) appear in the C code in the proper location where they have to be interpreted in the original

```
int main()
{
    init();
    while (1) {
        /* read inputs */
        step();
    }
}
```

Figure 10.5: Simulink Embedded Coder: Form of the C code

program. Thus, the SSA will adequately take them into account. The C code also contains three C structs that specify inputs, outputs, and state variables. Besides generating the C code, Embedded Coder also produces a Matlab structure comprising the translation information about the names and types of the Simulink and the C code. Using that information we can go back from the C code level to Simulink level for generating test cases for the failed assertions.

For generating the C code, Embedded Coder imposes certain assumptions on the kinds of the Simulink model it can handle. Most importantly, between the discrete-time and continuous-time models, it can handle only discrete-time models. Then it can not generate code for all the Simulink blocks. However, it can manage most of the discrete blocks. A complete list of blocks that support code generation is available at `http://www.mathworks.com/help/rtw/ug/supported-products-and-block-usage.html`.

Advantages and Disadvantages

The approach can handle a wide-range of Simulink and Stateflow blocks. Moreover, it can also support custom blocks with Matlab code. This flow is easy to maintain with the newer releases of Simulink since we mainly rely on the Embedded Coder which is also a product of MathWorks®. We can also leverage on the optimization provided by Embedded Coder that may help in getting a nuXmv model which is efficient for the verification process. Another added value is the possibility to use software model checking techniques for the analysis.

The possible drawbacks of the method are that the translated nuXmv model does not preserves the hierarchy of the Simulink model and is not very readable, which may be required for some other activities, e.g., safety analysis.

10.3 Related Work

Several translations from Simulink have been proposed to different formal languages and verification tools, e.g., the CoCoSim [coc, BGG+17] framework, BIP [STS+10], LUSTRE [TSCC05], Boogie [RG14], SL2SX [MF16]. These approaches use a tailor-made compiler for the translation, and because of that, the support for the kinds (in terms of the used Simulink blocks) of Simulink models they can translate is limited. In contrast, we rely on Embedded Coder for compilation, and our approach supports a wide range of Simulink blocks.

Chapter 11

Verilog to nuXmv

Note. *The work presented in this chapter was done as a part of the GRC Research Project 2012-TJ-2266 WOLF – funded by Semiconductor Research Corporation (SRC).*

VERILOG models describe the behavior of hardware designs at a high-level, that are (automatically) synthesized into Register Transfer Level (RTL) designs. The RTL specification contains the word-level information of a design. At RTL memories present in a design are either treated as a whole or expanded into individual elements. RTL designs are synthesized into gate-level logic designs and further synthesized into physical-level logic designs. In general, formal verification is performed at the gate level, where the structural information of the original design is almost lost. There are successful attempts to perform formal verification directly at RTL, though these approaches use verification techniques at the Boolean-level.

11.1 Problem Definition

Our goal is to lift the verification of hardware designs from gate level to RTL, exploiting recent VMT techniques, like those provided by the NUXMV VMT model checker [CGMT16, CCD$^+$14]. In particular we aim at handling efficiently designs with memories.

Our contributions are the following: First, we provide a verification tool-chain, based on Yosys [Wol]– a VERILOG synthesizing tool, and on NUXMV [CCD$^+$14]. We have extended Yosys to generate VMT problems from VERILOG and SystemVerilog assertions, in the NUXMV format, to be then analyzed with the advanced model checking algorithms provided by NUXMV. Moreover, the tool-chain allows also to generate VMT problems in other target languages to experiment with different verification backends.

11.2 Proposed Solution

We use Yosys as a white-box compiler, whose job is to simplify the complex structures of VERILOG language, flatten the hierarchy, and synthesize a word-level RTL design. Then the simplified design is translated into the input language of NUXMV.

Tool Architecture

The architecture of VERILOG2SMV is depicted in Fig. 11.1, which also shows VERILOG2SMV + NUXMV tool-chain. VERILOG2SMV takes as input a VERILOG design, written in VERILOG IEEE standard 2005 [ver06]. We assume the VERILOG design falls in the synthesizable subset of VERILOG [ver05]. We provide two complementary ways to specify the properties to be checked within the VERILOG design. Properties can be specified within the VERILOG model using the SystemVerilog `assert` statement. The `assert` statement can appear in the procedural block or in the module body of the VERILOG design (see `assert` at lines 10-11 in the example in Fig. 11.2). Properties can also be specified with a conventional notation by means of a single-bit output wire, whose name starts with `safety` (see

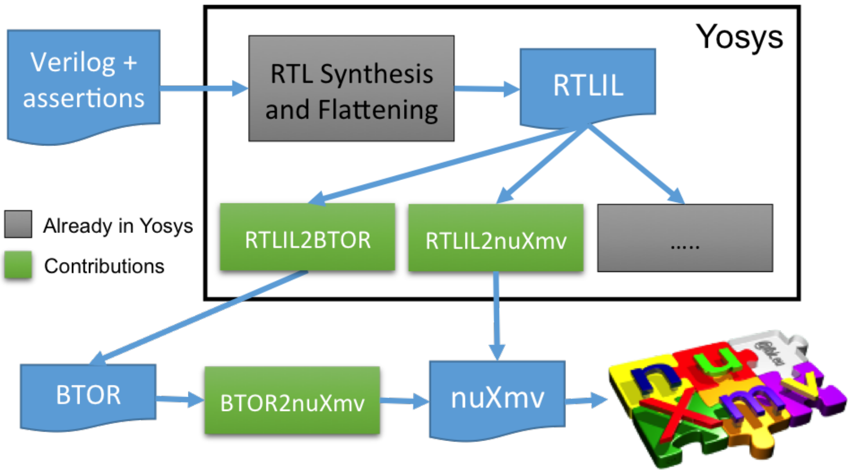


Figure 11.1: Verilog2SMV architecture and verification tool-chain

e.g., the `safety1` wire in Fig. 11.2). The value of this output wire needs to be driven high when the property is met.

The flow of the translation is the following: We leverage on Yosys to first flatten the Verilog high-level design, and then to synthesize the RTL circuit from the result of the flattening process. The RTL circuit is stored in the Yosys internal representation RTLIL. Yosys follows the IEEE Verilog standard synthesis semantics [ver05]. The RTL circuit goes into a new Yosys module that translates the input RTL circuit into a corresponding nuXmv and BTOR [IW15] problem. The translation preserves all the names of signals, registers, and memories of the original Verilog design. This makes it easy to interpret back in Verilog, possible counterexamples produced by the back-end model checkers. We explicitly model the clock as a Boolean input variable. This enables us to faithfully model flip-flops, memories, and latches, whereas it is not possible using the traditional translations since they do model clock implicitly. The translation does not currently handle multiple clocks, multi-dimensional arrays, and combinational logic loops. Note that the translation module only considers the parts of the design that are on the circuit path to the specified properties.

```
module array(input clk, output safety1);
   reg [7:0] counter;
   reg [7:0] mem [7:0];

   always @(posedge clk) begin
      counter <= counter + 8'd1;
      mem[counter] <= mem[counter] + 8'd1;
   end

   assert property (!(counter > 8'd0) || mem[counter - 8'd1] == counter - 8'd1);

   assign safety1 = (counter > 8'd0);
endmodule
```

Figure 11.2: Specifying property in VERILOG design

The two target verification languages, BTOR and NUXMV, allow to specify and reason about transition systems. The NUXMV language allows to express transition systems using all the finite data types available in NUSMV [CCG+02] (Booleans, enumerative, bounded integers), plus bit-vectors, reals, integers, and (finite and infinite) arrays, with no restriction on the specification of the initial values of the variables. On the other hand, BTOR is limited: 1) it can only deal with bit-vectors and one-dimensional arrays; 2) registers are implicitly initialized to the zero value, while arrays are uninitialized. In both cases we represent VERILOG registers of width greater than one, with corresponding variables of type bit-vectors within NUXMV and BTOR. However registers with width one are treated differently: NUXMV treats them as Boolean, while BTOR treats them as bit-vectors of width one. Memories in both cases are encoded with arrays of bit-vectors.

Notice that the conversion into BTOR is not complete in the sense that it supports zero-initialized registers and uninitialized memories.

Fig. 11.3 shows the NUXMV file generated with VERILOG2SMV starting from VERILOG module `array` as described in Fig. 11.2. We see that

```
MODULE main
IVAR
"clk" : boolean;

VAR
"counter" : word[8];
mem : array word[3] of word[8];

DEFINE
__expr1 := resize(0ub8_11111111, 1);
__expr2 := bool(__expr1);
__expr3 := "counter"[2:0];
__expr4 := READ(mem, __expr3);
__expr5 := (__expr4 + 0ub8_00000001);
__expr6 := ("clk");
__expr7 := (__expr6 & __expr2);
__expr8 := WRITE(mem, __expr3, __expr5);
__expr9 := ("counter" + 0ub8_00000001);
__expr10 := ("clk");
__expr11 := (__expr10 ? __expr9 : "counter");
__expr12 := next("counter") = __expr11;
__expr13 := (case __expr7: __expr8; TRUE: mem; esac);
__expr14 := next(mem) = __expr13;
__expr15 := (__expr12 & __expr14);
__expr16 := ("counter" > 0ub8_00000000);
__expr17 := word1(__expr16);
__expr18 := (0ud1_0 = __expr17);
__expr19 := ("counter" - 0ub8_00000001);
__expr20 := __expr19[2:0];
__expr21 := READ(mem, __expr20);
__expr22 := ("counter" - 0ub8_00000001);
__expr23 := (__expr21 = __expr22);
__expr24 := (__expr18 | __expr23);
__expr25 := bool(0ub1_1);
__expr26 := (__expr25 -> __expr24);
__expr27 := ("counter" > 0ub8_00000000);

INIT TRUE;
TRANS __expr15;
INVARSPEC __expr26;
INVARSPEC __expr27;
```

Figure 11.3: nuXmv translation for the Verilog design shown in Fig. 11.2

the memory `mem` is retained in the NUXMV file, as an array (declaration `mem : array word[3] of word[8]` at line 7). In the translation, we also introduce explicitly the clock and we model it as an input Boolean variable (see the `clk` input variable at line 3). Initial blocks in a VERILOG design converted into `INIT` constraints in the NUXMV file. In this example, since there is no initial block, the `INIT` constraint is simply the constant `TRUE`. The assignments to registers and memories are translated into `TRANS` constraint. (For details about the NUXMV syntax, we refer the reader the NUXMV user-manual [BCC+16].) Properties are simply translated into corresponding `INVARSPEC`. The `assert` command in Fig. 11.2 is translated into the first `INVARSPEC`, while the property corresponding to wire `satisfy1` is encoded into the second and last `INVARSPEC`.

As future work, we would like to extend the support to full SystemVerilog properties. A rather ambitious future direction would be extending VERILOG2SMV to convert VERILOG designs into threaded software program. This conversion will help verification (using techniques like ESST [CNR12]) of high-level VERILOG designs.

11.3 Related Work

We mention some tools that can be used to convert VERILOG designs into verification problems. (We omit considering tools which are no more maintained, like e.g., vl2mv [VSS+96].)

The following tools are publicly available. V3 [WWH] reads VERILOG designs and produces word-level BTOR designs using QuteRTL [YWH12] as a VERILOG frontend. ABC [BM10] has its VERILOG frontend, which transforms the designs into gate-level designs for verification. It cannot produce word-level MC problems. EBMC [KP] takes VERILOG designs with assertions and produces SMT formulas by applying BMC and/or k-induction. It can also output Boolean-level MC problem in SMV format. (EBMC is the successor of VCEGAR [JKSC07], which is no more maintained.) Yosys [Wol] is a freely-available synthesis tool from high-level VERILOG to RTL and gate-level VERILOG. A very-recent version can also produce SMT formulas representing combinatorial circuit designs. Cadence-SMV [McM00] can take VERILOG design as input, generating Boolean-level SMV design (in its publicly-available version). AVERROES [LS14] is a verification tool which takes input VERILOG designs and invariant properties. With the exception of EBMC, they all cannot handle memories without abstracting or blasting them.

The following tools are not publicly available. AiPG [Sug] is a verification tool built on top of Boolector [BB09]. It takes VERILOG designs and assertions as input. It can also produce BTOR designs. Reveal [ALS08] is a tool for the verification of Verilog designs against assertions. SIXTH-SENSE [Bau06] is a verification tool by IBM which can handle VERILOG designs.

Chapter 12

Implementation and Experimental Evaluation

12.1 Simulink to nuXmv

12.1.1 Implementation

We have implemented the SIMULINK to NUXMV approach in MATLAB, and C++. The process of generating C code for a SIMULINK model and interacting with KRATOS is automated in a MATLAB script. The script also includes SIMULINK test case model generation for the counterexample produced by NUXMV/KRATOS. We also extended KRATOS frontend to parse the extra details given in the C code, e.g., input variables, state variables, etc.

12.1.2 Experimental Evaluation

We evaluated our approach on an industrial-level SIMULINK model of a twin-engine aircraft simulation called Transport Class Model (TCM) [Hue11]. The TCM is a publicly available SIMULINK model with nonlinear dynamics. Due to the fact the model is nonlinear, it can not be analyzed using the verification tool SIMULINK DESIGN VERIFIER, made by MathWorks®.

The TCM model consists of approximately 5700 Simulink blocks. It models an aircraft that can be controlled manually (using ailStick and elevStick) or by autopilot (mode logic) – see Fig. 12.1. The TCM's autopilot controls altitude and maintains desired flight path angle (FPA), desired heading, and speed. The requirements are coming from pilot training manuals and the Federal Aviation Regulations for commercial aircraft. In [BBD+15], these requirements were formalized and verified by PKind [KT11] model checker. The Simulink model was automatically converted into a LUSTRE model, which was then fed to PKind for the verification task. Like Simulink Design Verifier, PKind cannot handle systems with nonlinear dynamics. Therefore, the LUSTRE model was approximated via linearization of nonlinear dynamics.

As a case study, we try to use our flow on G-120 and G-130 properties which were verified using compositional reasoning in the paper [BBD+15]. We chose them because they were explained in the paper. Here we describe G-120 since G-130 is very similar to it.

The assumptions for G-120 are G-180, A1, A2, FPA1:

G-120: The original requirement for G-120 is that guidance shall be capable of climbing at a defined rate, to be limited by minimum and maximum engine performance and airspeeds.

G-180: The FPA control shall engage when the altitude control mode is selected, and when there is no manual pitch or manual roll command from the stick.

A1: If not in Altitude control mode, the Altitude control module is not engaged.

A2: If Altitude control module is not engaged, the Altitude control module will not send commands.

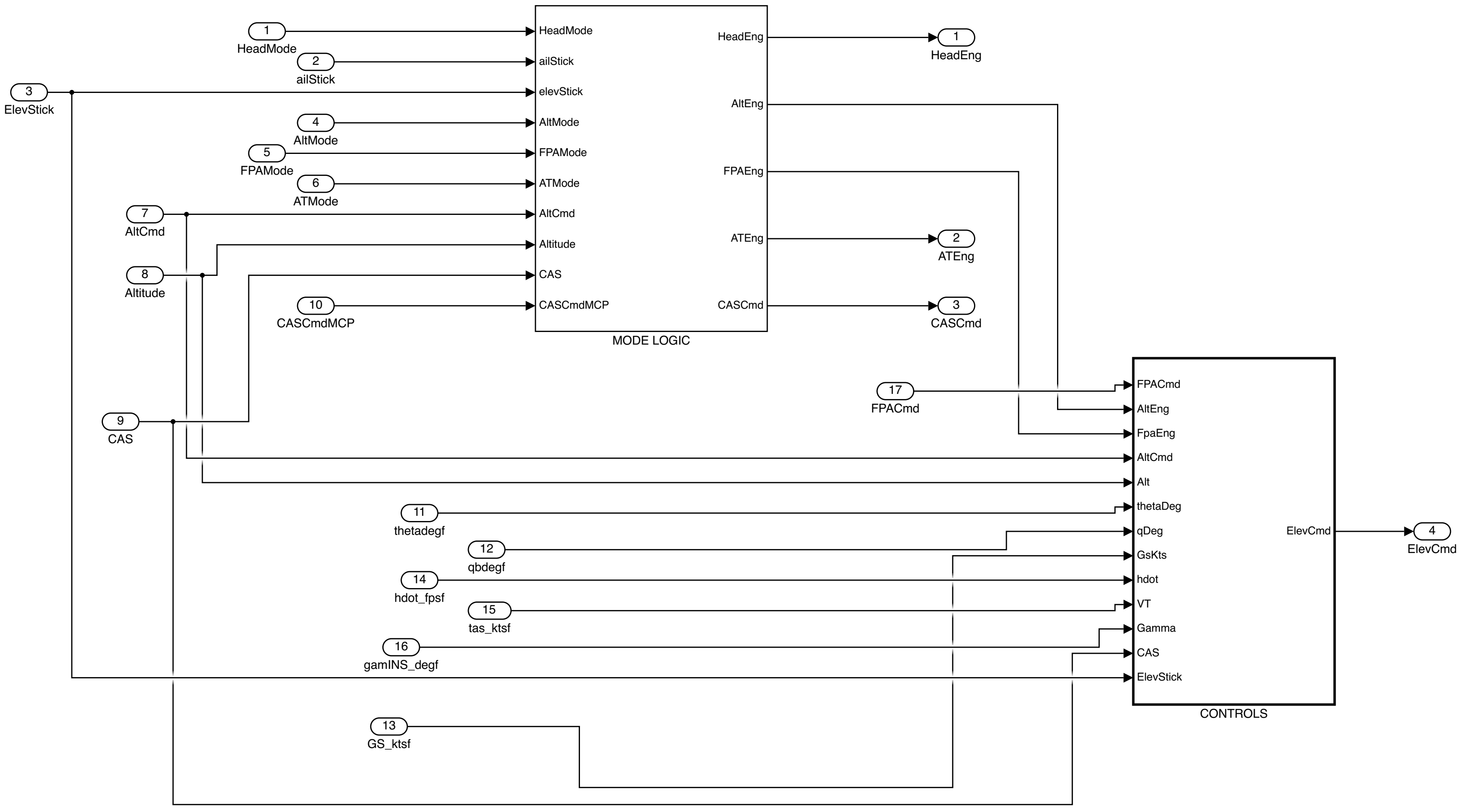


Figure 12.1: TCM: top level

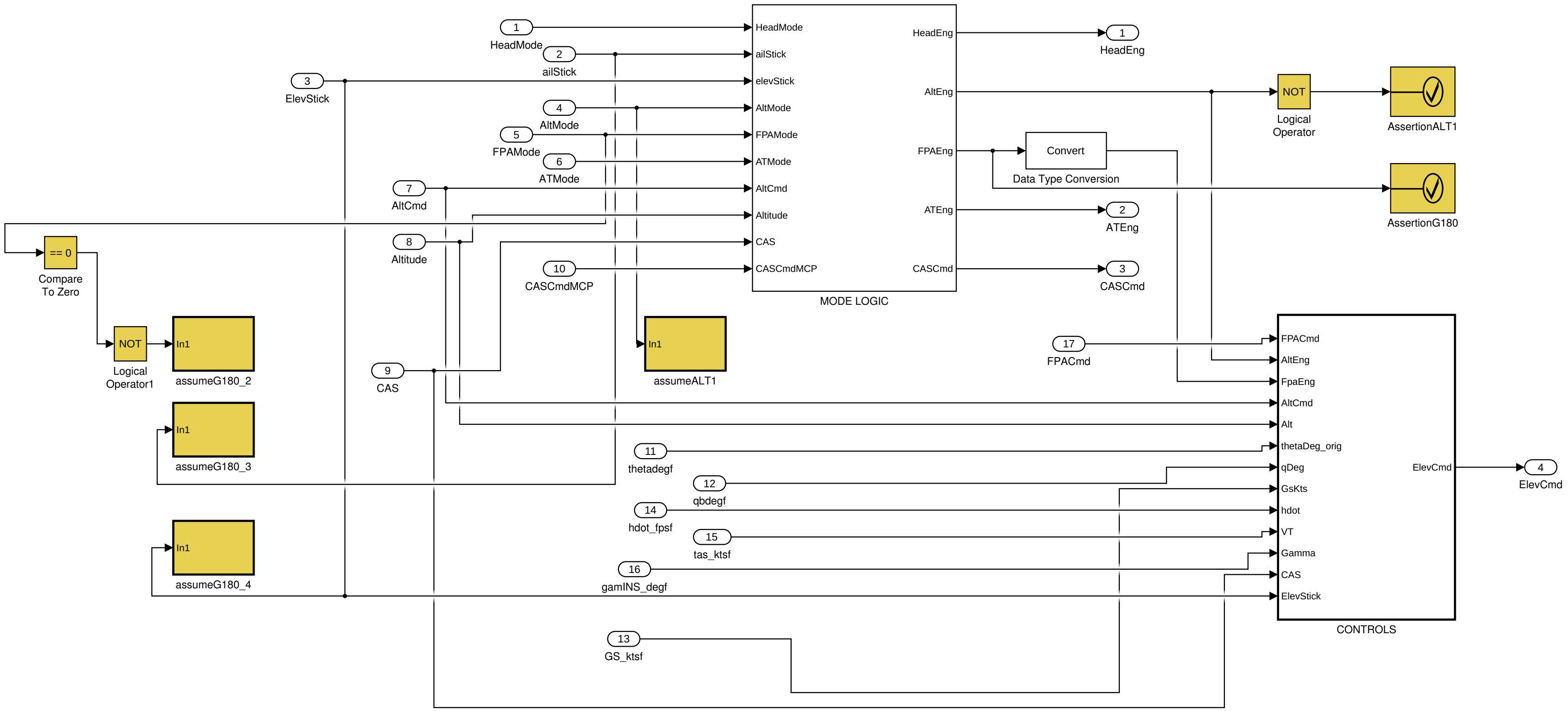


Figure 12.2: TCM with assumes and assertions – top level

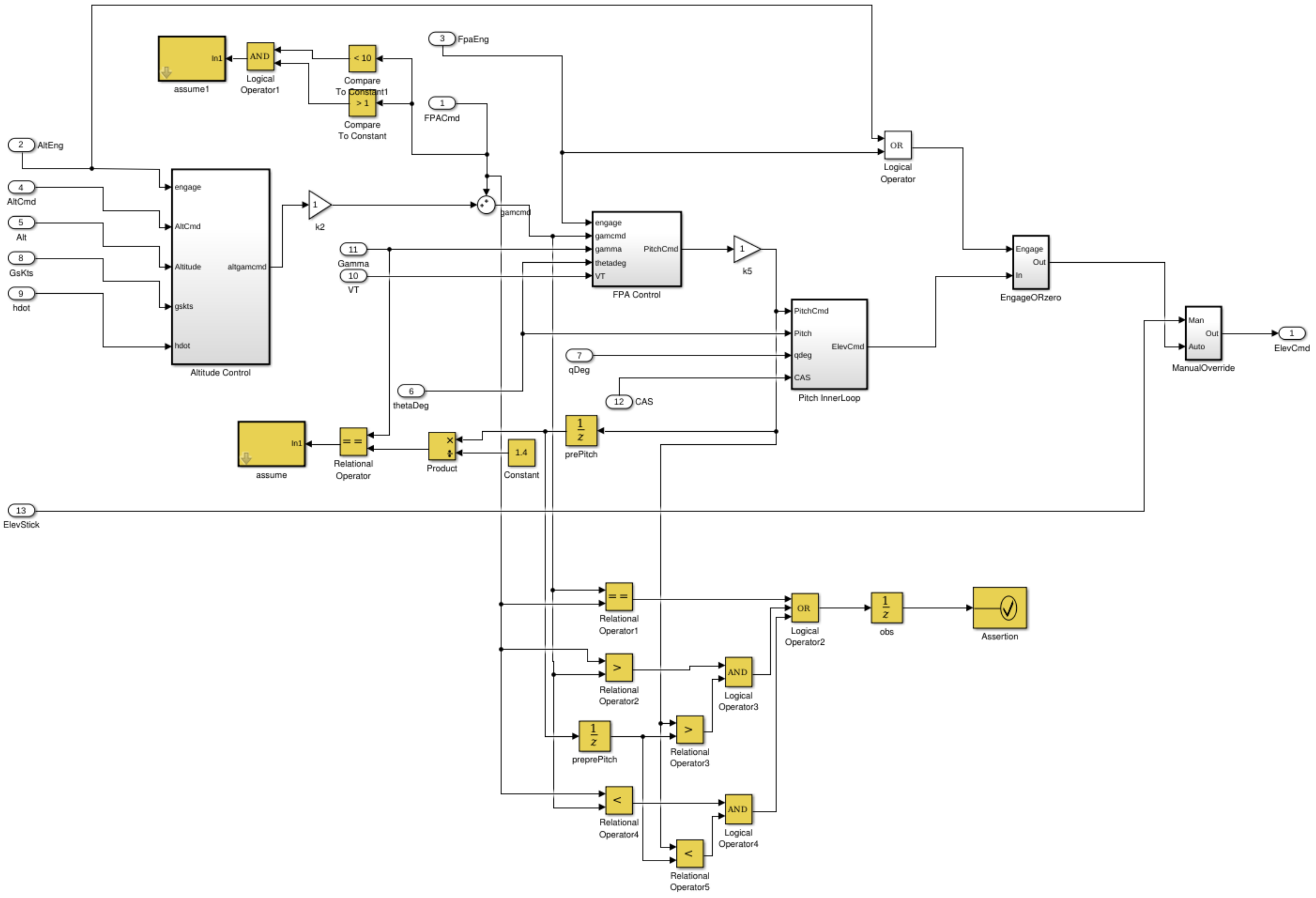


Figure 12.3: TCM with assumes and assertions – controls component

The properties were only available in the LUSTRE program. We have added them to the Simulink model using assume blocks (our custom Simulink block for specifying assumptions) and assert blocks (a built-in Simulink block), see Fig. 12.2 and Fig. 12.3 (added blocks are highlighted). We noticed that there was a type mismatch between the output port FPAEng of Mode Logic subsystem and the input port FpaEng of Controls subsystem. We resolved this issue by adding a Simulink data type converter block.

We ran our Simulink to nuXmv verification flow on the Simulink model with properties (see Fig. 12.2 and Fig. 12.3). We were able to verify them in less than 10 secs.

Remark 8. While adding the properties in the Simulink model, we found an assumption to be inconsistent. In fact, we reported this to the authors of the original paper, and they acknowledged our findings and provided us the corrected assumptions.

Remark 9. It is worth to remark that we did not approximate the model. Instead, we let nuXmv to handle the nonlinear dynamics of the model, via incremental linearization approach as presented in Chapter 8. Moreover, the overall process was done without human intervention.

12.2 Verilog to nuXmv

12.2.1 Implementation

Verilog2SMVis built on top of Yosys [Wol] and is written in C++. The call to nuXmv is done via a bash script.

nuXmv Language Extensions

We have extended the language of nuXmv for expressing bounded and unbounded arrays types, read and write operators over arrays, and constructing constant arrays. For dealing with memories, we need bounded arrays. Nevertheless, we explain the syntax for both bounded and unbounded arrays. For the complete syntax of the nuXmv language, we suggest to look at [BCC+16].

Array Types. Arrays types are multi-sorted: there is a sort for index and a sort for elements of arrays. The index type of bounded arrays is specified by a word (bit-vector) of fixed length in nuXmv, whereas the type of unbounded arrays is specified using the integer type in nuXmv. For example:

```
array word[5] of unsigned word[3];
array integer of unsigned word[3];
```

The first is a bounded array with 2^5 elements of type unsigned word[3]. The second is an unbounded array with elements of type unsigned word[3].

Read Operator. The read operator 'READ' extracts one element of an array at particular index. The first argument of the operator must be an expression of type either word or integer, and the type of second argument expression must be same as of the index type of the array expression in the first argument. The signature of the READ operator is:

`READ` : `array word[N] of` subtype * `word[N]` → subtype
: `array integer of` subtype * `integer` → subtype

Write Operator. The write operator 'WRITE' updates one element at a particular index of an array and returns the updated array as a new array.

The first argument of the operator must be an expression of type either word or integer. The type of the second and third argument expressions must be same as of the index type and element type of the array expression in the first argument. The signature of the `WRITE` operator is:

`WRITE` : `array word[N] of` subtype * `word[N]` * subtype
→ `array word[N] of` subtype
: `array integer of` subtype * `integer` * subtype
→ `array integer of` subtype

Constant Array. The constant array '`CONSTARRAY`' is a special constructor to create an array of given type having elements set to a uniform given value. For example, a constant array `CONSTARRAY(typeof(a), 0)` (suppose that `a` is of type `array integer of integer`), means an unbounded array of type `array integer of integer` with all elements value set to integer 0.

nuXmv VMT Extensions

We have extended the BMC, k-induction, and IC3ia procedures inside nuXmv for dealing with the theory of arrays.

12.2.2 Experimental Evaluation

In this section we describe an experimental evaluation we carried out to show the effectiveness of the Verilog2SMV + nuXmv tool-chain. We conducted an experimental analysis on real-world Verilog verification benchmarks with registers and memories. We compare our tool-chain using different back-end verification algorithms provided by nuXmv against other related tools.

Setup of the experimental evaluation

We have run our experiments on a cluster of 64-bit Linux machines with 2.7 GHz Intel Xeon X5650 CPUs, with a memory limit of 4GB and a time limit of 3600 seconds.

Benchmarks. We have considered a set of benchmark problems, VERILOG files and invariant properties files, from the VIS [Som] and VCEGAR [vce] benchmark suites. The collection includes 42 problems with memories and registers (40 from VIS and 2 from VCEGAR) and 44 problems with registers only (14 from VIS and 29 from VCEGAR), totalling 86 problems.

Other Tools. We have compared our tool-chain against v3, AVERROES, and EBMC on the collected benchmarks. We have also used the very-recently released version 4.2 of EBMC. Unfortunately, we can only show results against EBMC because v3 and AVERROES either were not able to process most of the VERILOG designs we collected, or they crashed without producing results.

Configurations. For benchmarks that contain both memories and registers, we have considered the following verification algorithms offered by NUXMV:

a) `nuxmv-k-ind`, SMT-based *k-induction/BMC*;

b) `nuxmv-bmc`, SMT-based *BMC*;

c) `nuxmv-ic3-ia`, SMT-based *IC3 with implicit abstraction* [CGMT16].

For registers-only benchmarks, besides a), b), and c), we have also considered:

d) `nuxmv-ic3` SAT-based IC3 [Bra11].

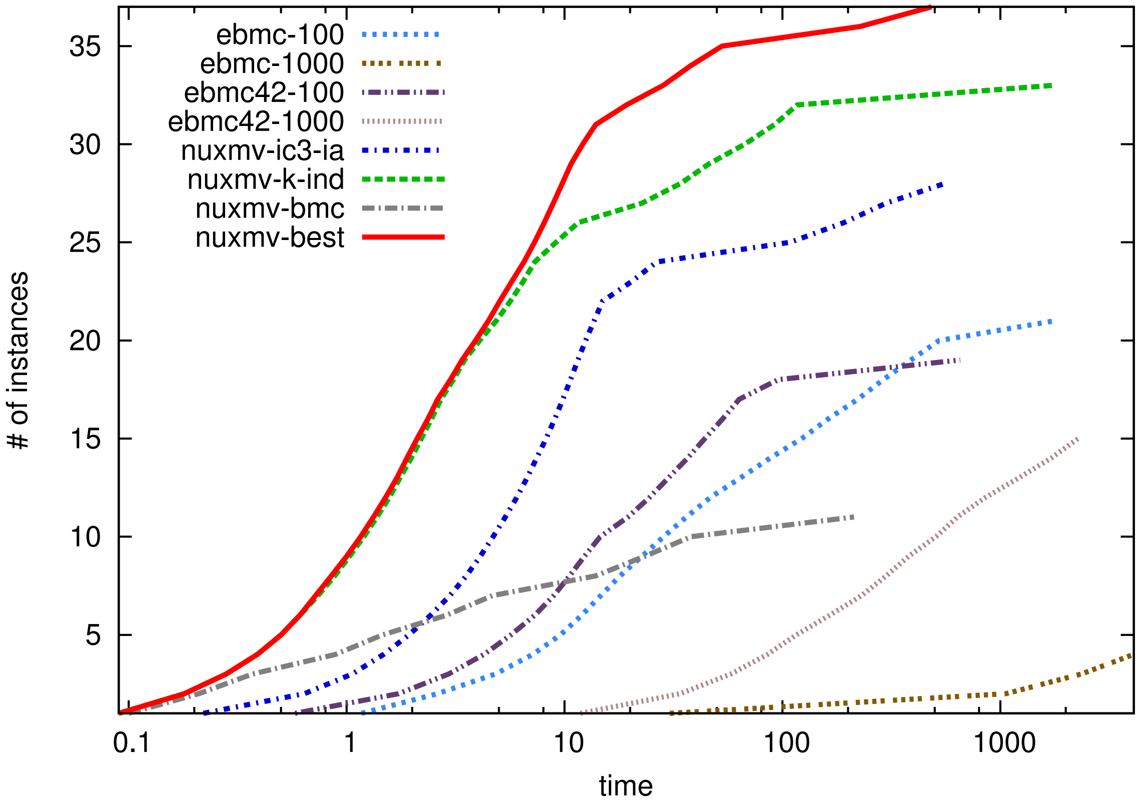


Figure 12.4: Accumulated plot for benchmarks with memories and registers

For each NUXMV configuration, the bound is 1000. For EBMC we use the the following configurations:

e) `ebmc-100`, *k-induction/BMC* with bound 100;

f) `ebmc-1000`, *k-induction/BMC* with bound 1000;

g) `ebmc42-100`, EMBC 4.2 using *k-induction/BMC* with bound 100;

h) `ebmc42-1000`, EBMC 4.2 using *k-induction/BMC* with bound 1000.

Results

The results of the experiments are shown in form of accumulated plots, where on the x-axis we have the accumulated solving time and on the y-axis we have the number of solved instances. Fig. 12.4 shows the results

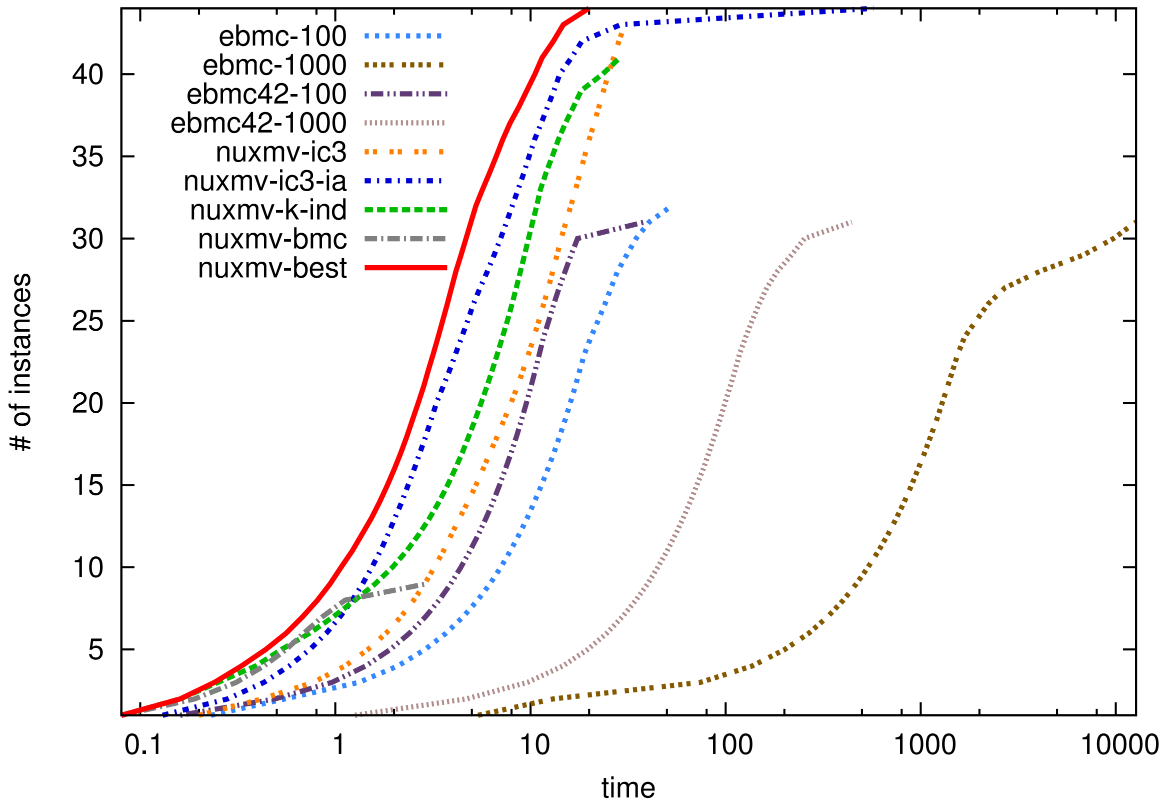


Figure 12.5: Accumulated plot for benchmarks with registers only

for the 42 benchmarks with memories and registers; Fig. 12.5 shows the results for the 44 benchmarks with registers only. In the plots, we also show `nuxmv-best` which is virtual best configuration for NUXMV.

The results clearly show that our tool-chain is performing better than EBMC, on the selected benchmarks. In particular, we see that on the benchmarks with memories and registers, `nuxmv-k-ind` has solved 33 benchmarks (22 safe, 11 unsafe), instead `nuxmv-ic3-ia` has been able to solve 28 benchmarks (25 safe, 3 unsafe). `ebmc42-100` has solved only 19 benchmarks (9 safe, 10 unsafe). We notice that `nuxmv-ic3-ia` has solved more safe instances than `nuxmv-k-ind`, probably since the former uses abstraction. However `nuxmv-ic3-ia` has solved less unsafe instances than `nuxmv-k-ind`, probably due to the limited support for array abstraction refinement in nuXmv. We skip the discussion for older version of EBMC, i.e., `ebmc-*` configurations.

For the benchmarks with registers only, `nuxmv-ic3-ia`, `nuxmv-ic3`, and `nuxmv-k-ind` are much closer. Indeed the first has solved all the 44 benchmarks (35 safe, 9 unsafe), the second has solved 43 benchmark (34 safe, 9 unsafe), and the last one has solved 41 benchmarks (32 safe, 9 unsafe). Interestingly `nuxmv-ic3-ia`, which is an SMT-based IC3, has shown better performance than SAT-based IC3 `nuxmv-ic3`. `ebmc42-100` has only solved 31 benchmarks (26 safe, 5 unsafe).

We conjecture that EBMC is slower than the nuXmv-based tools because it is not exploiting incrementality while solving.

Importantly, whenever terminating, on the latter benchmarks all tools always agree on the result, whilst on the former ones all `nuxmv-*` and `ebmc-*` always agree, whilst `ebmc42-*` disagrees with both `nuxmv-*` and `ebmc-*` on two instances.

We remark that, Verilog2SMV has the ability to generate benchmarks in different formats. In particular, It can generate Boolean-level benchmarks in AIGER [aig] format either using *synthebtor* tool, from the Boolector distribution, or using the converter in AIGER format built-in in nuXmv. This will enable for generating benchmarks and experimenting with any model checker from the hardware model checking competition.

Summary

In this part, we have worked towards bridging the gap between the design tools and the verification tools: we have integrated Simulink and Verilog with nuXmv via a compilation-based approach. The integration allowed us to develop workflows for verifying Simulink and Verilog designs with the state-of-the-art techniques provided by nuXmv.

We have presented the implementation details and evaluated our approaches on a Simulink case study and a set of Verilog benchmarks. In fact, the Simulink design contains nonlinear dynamics, and Simulink Design Verifier (the main verification tool for Simulink) does not handle it. As results of the proposed workflow and our contributions presented in part III for the verification of nonlinear transition systems, we can successfully verify invariant properties of the Simulink design. In the case of Verilog to nuXmv, the translation converts Verilog designs with assertions into RTL VMT problems, while fully treating memories. The effectiveness of the Verilog to nuXmv verification workflow has been demonstrated by an evaluation on a collection of Verilog benchmarks.

Our work can also provide a way to generate verification benchmarks from real-world Simulink and Verilog designs. In particular, for the case of Verilog, benchmarks at a higher level of abstraction – retaining registers and memories – can be generated, that can assist in the advancement of the hardware verification research which mostly focuses at the Boolean level due to the unavailability of word-level benchmarks.

Chapter 13

Thesis Conclusions

Formal verification has become extremely important for analyzing hardware and software systems. A prominent direction in formal verification is based on Satisfiability Modulo Theories (SMT), which is the problem of checking satisfiability of first-order formulae with respect to some background theories. Verification Modulo Theories (VMT) is the problem of analyzing systems described using SMT formulae. Based on the recent improvements in SMT technologies, effective VMT techniques have been developed for linear arithmetic. However, many real-world industrial designs (e.g., aerospace, automotive) also require modeling as systems over nonlinear arithmetic and transcendental functions, for which the VMT and SMT problems have received little attention.

In this thesis, we have addressed the SMT and VMT problems with respect to the theories of nonlinear arithmetic and transcendental functions. We have significantly improved the state of the art on the problems. Moreover, we have also contributed to the integration of the state-of-the-art nuXmv VMT model checker with two design tools – Simulink and Verilog. Our work provides the first automatic verification method for checking invariant properties of discrete-time Simulink designs.

We have proposed *incremental linearization* as a general framework for automated reasoning about nonlinear polynomials and transcendental functions such as exponentiation and trigonometric functions. We have implemented it inside the MathSAT SMT solver and the nuXmv VMT model checker. The experimental results show the merits of incremental linearization. The technique is surprisingly effective in SMT, even compared to other complete (when available) and more mature approaches. In VMT, incremental linearization significantly outperforms its competitors based on interval propagation.

The effectiveness of incremental linearization is possibly explained with the following insights. On the one hand, in contrast to linear arithmetic, nonlinear arithmetic (and also the extension with transcendental functions) is a hard-to-solve theory: in practice, most available complete solvers rely on expensive solvers; we try to avoid nonlinear reasoning, trading it for linear reasoning. On the other hand, proving properties of practical systems may not require the full power of nonlinear solving. In fact, some systems are mostly-linear (i.e., nonlinear constraints are associated to a tiny part of the system), an example being the Transport Class Model (TCM) for aircraft simulation from the Simulink model library [Hue11]. Furthermore, even a system with significant nonlinear dynamics may admit a piecewise-linear invariant that is strong enough to prove the property.

Overall, incremental linearization is a general and relatively simple idea that supports approaches to SMT and VMT, and can successfully tackle many practical problems. In fact, the approach has also been (independently) implemented in the CVC4 SMT solver for handling nonlinear polynomials, as presented in the paper [RTJB17]. Based on personal communication, also the extension to support transcendental functions via incremental linearization is underway.

13.1 Future Directions

This work opens a number of research directions.

In the SMT case, incremental linearization is clearly an incomplete technique in general for nonlinear problems, nevertheless, it would be interesting to identify subclasses of nonlinear problems for which incremental linearization is (theoretically) complete. The experimental results show a very good performance of incremental linearization on the real-world problems. Despite the success, we think there is a great potential for future work in the development of heuristics and preprocessing techniques (e.g., factorization techniques for polynomials), and the use of an additional set of the linearization rules (e.g., monomial- and polynomial-level axiomatization, and native handling of transcendental functions other than sin and exp). To further improve efficieny, we would like to investigate the integration of incremental linearization with complementary techniques, such as interval constraint propagation or Cylindrical Algebraic Decomposition (CAD) – a more powerful technique but also more expensive. Moreover, at the moment we do not use information from a failed attempt of the satisfiability detection heuristic. It would be interesting to study the impact of such information, which for instance can be extracted using unsatisfiable cores. For improving the performance on satisfiable instances, a possible direction is to integrate the work presented in [FOSV17], which provides a sufficient criteria for satisfiability without necessarily producing a model.

Since most state-of-the-art SMT solvers for $\mathcal{UF}$ and $\mathcal{LA}$ can compute interpolants [CGS10], extending incremental linearization to compute interpolants would be an exciting direction. Interpolants are extensively applied in verification of finite- and infinite-state transition systems. Currently, the interpolant computation for nonlinear arithmetic and transcendental functions is done using interval-based methods. It would be interesting to compare the strength of the interpolants produced by the current approaches against the ones using incremental linearization. Another application of incremental linearization would be in the case of Optimization Modulo Theories (OMT) [CFG$^+$10, ST15] problems w.r.t. $\mathcal{NRA}$, $\mathcal{NTA}$, and $\mathcal{NIA}$.

For the VMT case, an immediate thing to do is the evaluation of incremental linearization on the $\mathcal{NIA}$ benchmarks. Similar to the case of SMT, studying heuristics applied and exploring various options of the the refinement is an important direction to follow. Moreover, a significant step is an extension beyond invariant checking, to deal with full LTL specifications over nonlinear transition systems. We conjecture that the computation of limit values for series may help to deal with transition systems deriving from discrete-step controllers. Another interesting direction is an extension of incremental linearization to deal with hybrid automata featuring polynomial and transcendental dynamics. Exploring incremental linearization for proving termination of software program exhibiting nonlinear behavior is another possibility.

In the future, for the Simulink to nuXmv translation, we plan to investigate translation options that can support a wide range of Simulink designs as well as preserve the hierarchy of the design. This direction can open up the possibility to apply compositional reasoning using OCRA [CDT13].

Bibliography

[Ack54] W. Ackermann. *Solvable Cases of the Decision Problem.* Janua linguarum: Series maior. North-Holland Publishing Company, 1954.

[ACKS02] Gilles Audemard, Alessandro Cimatti, Artur Kornilowicz, and Roberto Sebastiani. Bounded model checking for timed systems. In *FORTE*, volume 2529 of *LNCS*, pages 243–259. Springer, 2002.

[AGMW13] Xavier Allamigeon, Stéphane Gaubert, Victor Magron, and Benjamin Werner. Certification of inequalities involving transcendental functions: combining sdp and max-plus approximation. In *Control Conference (ECC), 2013 European*, pages 2244–2250. IEEE, 2013.

[aig] `http://fmv.jku.at/aiger/`.

[ALS08] Zaher S Andraus, Mark H Liffiton, and Karem A Sakallah. Reveal: A formal verification tool for Verilog designs. In *LPAR*. Springer, 2008.

[alt] AltaRica. `https://altarica.labri.fr`.

[AP10] Behzad Akbarpour and Lawrence C. Paulson. MetiTarski: An automatic theorem prover for real-valued special functions. *J. Autom. Reasoning*, 44(3):175–205, 2010.

[APGR99] André Arnold, Gérald Point, Alain Griffault, and Antoine Rauzy. The AltaRica formalism for describing concurrent systems. *Fundam. Inform.*, 40(2-3):109–124, 1999.

[AR12] Alessandro Armando and Silvio Ranise. Scalable automated symbolic analysis of administrative role-based access control policies by SMT solving. *Journal of Computer Security*, 20(4):309–352, 2012.

[ARTW16] Alessandro Armando, Silvio Ranise, Riccardo Traverso, and Konrad S. Wrona. Smt-based enforcement and analysis of NATO content-based protection and release policies. In *ABAC@CODASPY*, pages 35–46. ACM, 2016.

[aut] AutoFocus3 (AF3). `https://af3.fortiss.org`.

[Bau06] Jason R Baumgartner. Semi-formal verification at IBM. In *HLDVT*. IEEE, 2006.

[BB09] Robert Brummayer and Armin Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In *TACAS*, TACAS '09. Springer, 2009.

[BBB13] J.L. Berggren, J. Borwein, and P. Borwein. *Pi: A Source Book*. Springer New York, 2013.

[BBC$^+$06] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi A. Junttila, Silvio Ranise, Peter van Rossum, and Roberto Sebastiani. Efficient theory combination via boolean search. *Inf. Comput.*, 204(10):1493–1525, 2006.

[BBC16] Benjamin Bittner, Marco Bozzano, and Alessandro Cimatti. Automated synthesis of timed failure propagation graphs. In *IJCAI*, pages 972–978. IJCAI/AAAI Press, 2016.

[BBD+15] Guillaume Brat, David H. Bushnell, Misty Davies, Dimitra Giannakopoulou, Falk Howar, and Temesghen Kahsai. Verifying the safety of a flight-critical system. In *FM*, volume 9109 of *LNCS*, pages 308–324. Springer, 2015.

[BBJ15] Stanley Bak, Sergiy Bogomolov, and Taylor T. Johnson. HYST: a source transformation and translation tool for hybrid automaton models. In *HSCC*, pages 128–133. ACM, 2015.

[BBL+17] Cristina Borralleras, Marc Brockschmidt, Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Proving termination through conditional termination. In *TACAS 2017, Proceedings, Part I*, volume 10205 of *LNCS*, pages 99–117, 2017.

[BBW14] Johannes Birgmeier, Aaron R. Bradley, and Georg Weissenbacher. Counterexample to induction-guided abstraction-refinement (CTIGAR). In *CAV*, volume 8559 of *LNCS*, pages 831–848. Springer, 2014.

[BCC+16] Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. nuXmv 1.1.1 user manual. `https://es.fbk.eu/tools/nuxmv/downloads/nuxmv-user-manual.pdf`, 2016.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic Model Checking Without BDDs. In *TACAS*, TACAS '99, pages 193–207, London, UK, 1999. Springer-Verlag.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and

Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.

[BCK$^+$11] Marco Bozzano, Alessandro Cimatti, Joost-Pieter Katoen, Viet Yen Nguyen, Thomas Noll, and Marco Roveri. Safety, dependability and performance analysis of extended AADL models. *Comput. J.*, 54(5):754–775, 2011.

[BCR$^+$09] Marco Bozzano, Alessandro Cimatti, Marco Roveri, Joost-Pieter Katoen, Viet Yen Nguyen, and Thomas Noll. Codesign of dependable systems: A component-based modeling language. In *MEMOCODE*, pages 121–130. IEEE, 2009.

[BD07] Christopher W. Brown and James H. Davenport. The complexity of quantifier elimination and cylindrical algebraic decomposition. In *ISSAC*, pages 54–60. ACM, 2007.

[BDG$^+$13] Martin Brain, Vijay D'Silva, Alberto Griggio, Leopold Haller, and Daniel Kroening. Interpolation-based verification of floating-point programs with abstract CDCL. In *SAS*, volume 7935 of *LNCS*, pages 412–432. Springer, 2013.

[BFT16] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). `www.SMT-LIB.org`, 2016.

[BG06] Frédéric Benhamou and Laurent Granvilliers. Continuous and interval constraints. In *Handbook of Constraint Programming*, volume 2 of *Foundations of Artificial Intelligence*, pages 571–603. Elsevier, 2006.

[BGG$^+$17] Hamza Bourbouh, Pierre-Loïc Garoche, Christophe Garion, Arie Gurfinkel, Temesghen Kahsai, and Xavier Thirioux. Automated analysis of stateflow models. In *LPAR*, volume 46 of *EPiC Series in Computing*, pages 144–161. EasyChair, 2017.

[BLO$^+$12] Cristina Borralleras, Salvador Lucas, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Sat modulo linear arithmetic for solving polynomial constraints. *J. Autom. Reason.*, 48(1):107–131, January 2012.

[BM07] Aaron R. Bradley and Zohar Manna. *The Calculus of Computation: Decision Procedures with Applications to Verification.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2007.

[BM10] Robert K. Brayton and Alan Mishchenko. ABC: an academic industrial-strength verification tool. In *CAV*, volume 6174 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2010.

[BMS06] Aaron R. Bradley, Zohar Manna, and Henny B. Sipma. What's decidable about arrays? In *Proceedings of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'06, pages 427–442, Berlin, Heidelberg, 2006. Springer-Verlag.

[BODF09] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. veriT: An open, trustable and efficient SMT-solver. In *CADE-22*, volume 5663 of *LNCS*, pages 151–156. Springer, 2009.

[Bra11] Aaron R. Bradley. SAT-based model checking without unrolling. In *VMCAI*, volume 6538 of *LNCS*, pages 70–87. Springer, 2011.

[BSST09] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, New York, NY, USA, 2004.

[CÁ11] Florian Corzilius and Erika Ábrahám. Virtual substitution for smt-solving. In *FCT*, volume 6914 of *Lecture Notes in Computer Science*, pages 360–371. Springer, 2011.

[CCD+14] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuXmv symbolic model checker. In *CAV*, volume 8559 of *LNCS*, pages 334–342. Springer, 2014.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In *CAV 2002, Proceedings*, pages 359–364, 2002.

[CDT13] Alessandro Cimatti, Michele Dorigatti, and Stefano Tonetta. OCRA: A tool for checking the refinement of temporal contracts. In *IEEE/ACM ASE 2013*, pages 702–705, 2013.

[CFG+10] Alessandro Cimatti, Anders Franzén, Alberto Griggio, Roberto Sebastiani, and Cristian Stenico. Satisfiability modulo the theory of costs: Foundations and applications. In *TACAS 2010, March 20-28, 2010. Proceedings*, volume 6015 of *LNCS*, pages 99–113. Springer, 2010.

[CFR14] Paula Chocron, Pascal Fontaine, and Christophe Ringeissen. A gentle non-disjoint combination of satisfiability procedures. In *IJCAR*, volume 8562 of *Lecture Notes in Computer Science*, pages 122–136. Springer, 2014.

[CG12] Alessandro Cimatti and Alberto Griggio. Software model checking via IC3. In *CAV*, volume 7358 of *Lecture Notes in Computer Science*, pages 277–293. Springer, 2012.

[CGI$^+$17a] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Invariant checking of NRA transition systems via incremental reduction to LRA with EUF. In *TACAS*, volume 10205 of *LNCS*, pages 58–75, 2017.

[CGI$^+$17b] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Satisfiability modulo transcendental functions via incremental linearization. In *CADE 26*, volume 10395 of *LNCS*, pages 95–113. Springer, 2017.

[CGI$^+$18] Alessandro Cimatti, Alberto Griggio, Ahmed Irfan, Marco Roveri, and Roberto Sebastiani. Experimenting on solving nonlinear integer arithmetic with incremental linearization. In *SAT*, LNCS. Springer, 2018. To appear. Available at `https://es.fbk.eu/people/irfan/papers/sat18.pdf`.

[CGJ$^+$00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, pages 154–169, 2000.

[CGKT16] Adrien Champion, Arie Gurfinkel, Temesghen Kahsai, and Cesare Tinelli. Cocospec: A mode-aware contract language for reactive systems. In *SEFM*, volume 9763 of *LNCS*, pages 347–366. Springer, 2016.

[CGM$^+$11] Alessandro Cimatti, Alberto Griggio, Andrea Micheli, Iman Narasamdya, and Marco Roveri. Kratos - A software model checker for SystemC. In *CAV, July 14-20, 2011. Proceedings*, pages 310–316, 2011.

[CGMT15] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. HyComp: An SMT-based model checker for hybrid systems. In *TACAS*, volume 9035 of *LNCS*, pages 52–67. Springer, 2015.

[CGMT16] Alessandro Cimatti, Alberto Griggio, Sergio Mover, and Stefano Tonetta. Infinite-state invariant checking with IC3 and predicate abstraction. *Formal Methods in System Design*, 49(3):190–218, 2016.

[CGS10] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Efficient generation of Craig interpolants in satisfiability modulo theories. *ACM Trans. Comput. Log.*, 12(1):7:1–7:54, 2010.

[CGS11] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. Computing small unsatisfiable cores in SAT modulo theories. *Journal of Artificial Intelligence Research, JAIR*, 40:701–728, April 2011.

[CGSS13] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT Solver. In *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.

[che] CHESS Project. `http://www.chess-project.org`.

[CKJ+15] 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 *SAT*, volume 9340 of *LNCS*, pages 360–368. Springer, 2015.

[CLJÁ12] Florian Corzilius, Ulrich Loup, Sebastian Junges, and Erika Ábrahám. SMT-RAT: an smt-compliant nonlinear real arithmetic toolbox - (tool presentation). In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 442–448. Springer, 2012.

[CLP+16] Gianpiero Cabodi, Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Danilo Vendraminetto, Armin Biere, and Keijo Heljanko. Hardware model checking competition 2014: an analysis and comparison of solvers and benchmarks. *Journal on Satisfiability, Boolean Modeling and Computation*, 9:135–172, 2016.

[CMR15] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Strong temporal planning with uncontrollable durations: A state-space approach. In *AAAI*, pages 3254–3260. AAAI Press, 2015.

[CMR16] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Dynamic controllability of disjunctive temporal networks: Validation and synthesis of executable strategies. In *AAAI*, pages 3116–3122. AAAI Press, 2016.

[CMR17] Alessandro Cimatti, Andrea Micheli, and Marco Roveri. Validating domains and plans for temporal planning via encoding into infinite-state linear temporal logic. In *AAAI*, pages 3547–3554. AAAI Press, 2017.

[CMS16] Alessandro Cimatti, Sergio Mover, and Mirko Sessa. From electrical switched networks to hybrid automata. In *FM*, volume 9995 of *LNCS*, pages 164–181. Springer, 2016.

[CMST16] Adrien Champion, Alain Mebsout, Christoph Sticksel, and Cesare Tinelli. The kind 2 model checker. In *CAV 2016, Proceedings, Part II*, pages 510–517, 2016.

[CMT12] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. A quantifier-free SMT encoding of non-linear hybrid automata. In *FMCAD*, pages 187–195. IEEE, 2012.

[CMT13] Alessandro Cimatti, Sergio Mover, and Stefano Tonetta. Smt-based scenario verification for hybrid systems. *Formal Methods in System Design*, 42(1):46–66, 2013.

[CNR12] Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. Software model checking with explicit scheduler and symbolic threads. *Logical Methods in Computer Science*, 8(2), 2012.

[coc] CoCoSim. `https://github.com/coco-team/cocoSim`.

[Col74] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition-preliminary report. *ACM SIGSAM Bulletin*, 8(3):80–90, 1974.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *STOC*, pages 151–158. ACM, 1971.

[CPHP87] Paul Caspi, Daniel Pilaud, Nicolas Halbwachs, and John Plaice. Lustre: A declarative language for programming synchronous systems. In *ACM POPL, 1987*, pages 178–188, 1987.

[CSW15] David R. Cok, Aaron Stump, and Tjark Weber. The 2013 evaluation of SMT-COMP and SMT-LIB. *J. Autom. Reasoning*, 55(1):61–90, 2015.

[dDLM11] Florent de Dinechin, Christoph Quirin Lauter, and Guillaume Melquiond. Certifying the floating-point implementation of an elementary function using Gappa. *IEEE Trans. Computers*, 60(2):242–253, 2011.

[DdM06] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for DPLL(T). In *CAV 2006, Proceedings*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.

[DE17] James H. Davenport and Matthew England. The potential and challenges of CAD with equational constraints for SC-Square. In *MACIS 2017, Proceedings*, volume 10693 of *LNCS*, pages 280–285. Springer, 2017.

[DH88] James H. Davenport and Joos Heintz. Real quantifier elimination is doubly exponential. *J. Symb. Comput.*, 5(1/2):29–35, 1988.

[DLL62] Martin Davis, George Logemann, and Donald W. Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, 1962.

[dMB08a] Leonardo Mendonça de Moura and Nikolaj Bjørner. Model-based theory combination. *Electr. Notes Theor. Comput. Sci.*, 198(2):37–49, 2008.

[dMB08b] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[dMJ13] Leonardo Mendonça de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *VMCAI*, volume 7737 of *LNCS*, pages 1–12. Springer, 2013.

[dMRS02] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Lazy theorem proving for bounded model checking over infinite domains. In *CADE*, volume 2392 of *Lecture Notes in Computer Science*, pages 438–455. Springer, 2002.

[dMRS03] Leonardo Mendonça de Moura, Harald Rueß, and Maria Sorea. Bounded model checking and induction: From refutation to verification (extended abstract, category A). In *CAV*, volume 2725 of *LNCS*, pages 14–26. Springer, 2003.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[Dut14] Bruno Dutertre. Yices 2.2. In *CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.

[EKK$^+$11] Andreas Eggers, Evgeny Kruglov, Stefan Kupferschmid, Karsten Scheibler, Tino Teige, and Christoph Weidenbach. Superposition modulo non-linear arithmetic. In *FroCoS*, volume 6989 of *LNCS*, pages 119–134. Springer, 2011.

[EMB11] Niklas Eén, Alan Mishchenko, and Robert K. Brayton. Efficient implementation of property directed reachability. In *FMCAD*, pages 125–134. FMCAD Inc., 2011.

[ES03a] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In *SAT*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003.

[ES03b] Niklas Eén and Niklas Sörensson. Temporal induction by incremental SAT solving. *Electr. Notes Theor. Comput. Sci.*, 89(4):543–560, 2003.

[FCN+10] Anders Franzén, Alessandro Cimatti, Alexander Nadel, Roberto Sebastiani, and Jonathan Shalev. Applying SMT in symbolic execution of microcode. In *FMCAD*, pages 121–128. IEEE, 2010.

[FGM+07] Carsten Fuhs, Jürgen Giesl, Aart Middeldorp, Peter Schneider-Kamp, René Thiemann, and Harald Zankl. SAT solving for termination analysis with polynomial interpretations. In *SAT 2007, Proceedings*, volume 4501 of *LNCS*, pages 340–354. Springer, 2007.

[FHT+07] Martin Fränzle, Christian Herde, Tino Teige, Stefan Ratschan, and Tobias Schubert. Efficient solving of large non-linear arithmetic constraint systems with complex boolean structure. *JSAT*, 1(3-4):209–236, 2007.

[FLVC04] Peter H. Feiler, Bruce A. Lewis, Steve Vestal, and Edward Colbert. An overview of the SAE architecture analysis & design language (AADL) standard: A basis for model-based architecture-driven embedded systems engineering. In *IFIP-WADL*, volume 176 of *IFIP*, pages 3–15. Springer, 2004.

[Fon09] Pascal Fontaine. Combinations of theories for decidable fragments of first-order logic. In *FroCoS*, volume 5749 of *Lecture Notes in Computer Science*, pages 263–278. Springer, 2009.

[FOSV17] Pascal Fontaine, Mizuhito Ogawa, Thomas Sturm, and Xuan-Tung Vu. Subtropical satisfiability. In *FroCoS*, volume 10483 of *LNCS*, pages 189–206. Springer, 2017.

[GAC12] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. δ-complete decision procedures for satisfiability over the reals. In *IJCAR*, volume 7364 of *LNCS*, pages 286–300. Springer, 2012.

[Gac16] A Gacek. JKind–an infinite-state model checker for safety properties in lustre, 2016.

[GB06] Laurent Granvilliers and Frédéric Benhamou. Algorithm 852: Realpaver: an interval solver using constraint satisfaction techniques. *ACM Trans. Math. Softw.*, 32(1):138–156, 2006.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dReal: An SMT solver for nonlinear theories over the reals. In *CADE-24*, volume 7898 of *LNCS*, pages 208–214. Springer, 2013.

[GLS10] Alberto Griggio, Thi Thieu Hoa Le, and Roberto Sebastiani. Efficient interpolant generation in satisfiability modulo linear integer arithmetic. *LNCS*, 8(3), 2010.

[GM15] Marco Gario and Andrea Micheli. PySMT: a solver-agnostic library for fast prototyping of SMT-based algorithms. In *SMT*, pages 373–384, 2015.

[Gri12] Alberto Griggio. A practical approach to satisability modulo linear integer arithmetic. *JSAT*, 8(1/2):1–27, 2012.

[GZ16] Sicun Gao and Damien Zufferey. Interpolants in nonlinear theories over the reals. In *TACAS 2016, Proceedings*, volume 9636 of *Lecture Notes in Computer Science*, pages 625–641. Springer, 2016.

[Haz93] M. Hazewinkel. *Encyclopaedia of Mathematics: Stochastic Approximation – Zygmund Class of Functions*. Encyclopaedia of Mathematics. Springer Netherlands, 1993.

[HB12a] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 157–171. Springer, 2012.

[HB12b] Krystof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *SAT*, volume 7317 of *LNCS*, pages 157–171. Springer, 2012.

[HKPV98] Thomas A. Henzinger, Peter W. Kopke, Anuj Puri, and Pravin Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57(1):94 – 124, 1998.

[Hue11] Richard M Hueschen. Development of the Transport Class Model (TCM) aircraft simulation from a sub-scale Generic Transport Model (GTM) simulation. Technical report, NASA Langley Research Center, 2011.

[ICG$^+$16] Ahmed Irfan, Alessandro Cimatti, Alberto Griggio, Marco Roveri, and Roberto Sebastiani. Verilog2smv: A tool for word-level verification. In *DATE*, pages 1156–1159. IEEE, 2016.

[IW15] Ahmed Irfan and Clifford Wolf. Yosys AppNote 012: Converting Verilog to BTOR. `http://www.clifford.at/yosys/files/yosys_appnote_012_verilog_to_btor.pdf`, 2015.

[JdM12] Dejan Jovanovic and Leonardo de Moura. Solving non-linear arithmetic. *ACM Comm. Computer Algebra*, 46(3/4):104–105, 2012.

[JKSC07] Himanshu Jain, Daniel Kroening, Natasha Sharygina, and Edmund M. Clarke. VCEGAR: verilog counterexample guided abstraction refinement. In *TACAS*, 2007.

[JLCÁ13] Sebastian Junges, Ulrich Loup, Florian Corzilius, and Erika Ábrahám. On gröbner bases in the context of satisfiability-modulo-theories solving over the real numbers. In *CAI*, volume 8080 of *Lecture Notes in Computer Science*, pages 186–198. Springer, 2013.

[Jov17] Dejan Jovanovic. Solving nonlinear integer arithmetic with MCSAT. In *VMCAI 2017, Proceedings*, volume 10145 of *LNCS*, pages 330–346. Springer, 2017.

[Kar84] N. Karmarkar. A new polynomial-time algorithm for linear programming. In *STOC 1984*, pages 302–311, New York, NY, USA, 1984. ACM.

[KB11] Stefan Kupferschmid and Bernd Becker. Craig interpolation in the presence of non-linear constraints. In *FORMATS*, volume 6919 of *LNCS*, pages 240–255. Springer, 2011.

[KCÁ16] Gereon Kremer, Florian Corzilius, and Erika Ábrahám. A generalised branch-and-bound approach and its application in SAT modulo nonlinear integer arithmetic. In *CASC 2016, Proceedings*, volume 9890 of *LNCS*, pages 315–335. Springer, 2016.

[KGC16a] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[KGC16b] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. SMT-based model checking for recursive programs. *Formal Methods in System Design*, 48(3):175–205, 2016.

[KGCC15] Soonho Kong, Sicun Gao, Wei Chen, and Edmund M. Clarke. dReach: δ-reachability analysis for hybrid systems. In *TACAS*, volume 9035 of *LNCS*, pages 200–205. Springer, 2015.

[KP] Daniel Kroening and Mitra Purandare. EBMC. `http://www.cprover.org/ebmc/`.

[KSJ09] Hyondeuk Kim, Fabio Somenzi, and HoonSang Jin. Efficient term-ite conversion for satisfiability modulo theories. In *SAT*, volume 5584 of *Lecture Notes in Computer Science*, pages 195–208. Springer, 2009.

[KT11] Temesghen Kahsai and Cesare Tinelli. PKind: A parallel k-induction based model checker. In *PDMC*, volume 72 of *EPTCS*, pages 55–62, 2011.

[LORR14] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. Minimal-model-guided approaches to solving polynomial constraints and extensions. In *SAT 2014, Proceedings*, volume 8561 of *LNCS*, pages 333–350. Springer, 2014.

[LS14] Suho Lee and Karem A Sakallah. Unbounded scalable verification based on approximate property-directed reachability and datapath abstraction. In *CAV*. Springer, 2014.

[Mag14] Victor Magron. NLCertify: A tool for formal nonlinear optimization. In *ICMS*, volume 8592 of *LNCS*, pages 315–320. Springer, 2014.

[Mat93] Yuri Vladimirovich Matiyasevich. *Hilbert's Tenth Problem.* Foundations of computing. MIT Press, 1993.

[McM00] Ken McMillan. Cadence smv. *Cadence Berkeley Labs, CA*, 2000.

[McM03] Kenneth L. McMillan. Interpolation and SAT-based model checking. In *CAV*, volume 2725 of *LNCS*, pages 1–13. Springer, 2003.

[McM05] Kenneth L. McMillan. Applications of Craig interpolants in model checking. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2005.

[Mel11] Guillaume Melquiond. Coq-interval, 2011.

[MF16] Stefano Minopoli and Goran Frehse. SL2SX translator: From simulink to SpaceEx models. In *HSCC*, pages 93–98. ACM, 2016.

[MFK$^+$16] Alexandre Maréchal, Alexis Fouilhé, Tim King, David Monniaux, and Michaël Périn. Polyhedral approximation of multivariate polynomials using Handelman's Theorem. In *VMCAI*, volume 9583 of *LNCS*, pages 166–184. Springer, 2016.

[MM16] Érik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *J. Autom. Reasoning*, 57(3):187–217, 2016.

[MSN$^+$16] Ahmed Mahdi, Karsten Scheibler, Felix Neubauer, Martin Fränzle, and Bernd Becker. Advancing software model checking beyond linear arithmetic theories. In *HVC*, volume 10028 of *Lecture Notes in Computer Science*, pages 186–201, 2016.

[Niv61] Ivan Niven. *Numbers: Rational and Irrational.* Mathematical Association of America, 1961.

[NO79] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.

[NO06] Robert Nieuwenhuis and Albert Oliveras. On SAT modulo theories and optimization problems. In *SAT 2006, Proceedings*, volume 4121 of *LNCS*, pages 156–169. Springer, 2006.

[NO07] Robert Nieuwenhuis and Albert Oliveras. Fast congruence closure and extensions. *Inf. Comput.*, 205(4):557–580, 2007.

[NOT06] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006.

[NPSS10] Pierluigi Nuzzo, Alberto Puggelli, Sanjit A. Seshia, and Alberto L. Sangiovanni-Vincentelli. CalCS: SMT solving for nonlinear convex constraints. In *FMCAD*, pages 71–79. IEEE, 2010.

[NW88] George L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley-Interscience, New York, NY, USA, 1988.

[Opp80] Derek C. Oppen. Complexity, convexity and combinations of theories. *Theor. Comput. Sci.*, 12:291–302, 1980.

[Pap81] Christos H. Papadimitriou. On the complexity of integer programming. *J. ACM*, 28(4):765–768, October 1981.

[Rat06] Stefan Ratschan. Efficient solving of quantified inequality constraints over the real numbers. *ACM Trans. Comput. Log.*, 7(4):723–748, 2006.

[RDK$^+$15] Andrew Reynolds, Morgan Deters, Viktor Kuncak, Cesare Tinelli, and Clark W. Barrett. Counterexample-guided quantifier instantiation for synthesis in SMT. In *CAV*, volume 9207 of *LNCS*, pages 198–216. Springer, 2015.

[RG14] Robert Reicherdt and Sabine Glesner. Formal verification of discrete-time MATLAB/Simulink models using boogie. In *SEFM*, volume 8702 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2014.

[Ric68] Daniel Richardson. Some undecidable problems involving elementary functions of a real variable. *J. Symb. Log.*, 33(4):514–520, 1968.

[RKFB17] Heinz Riener, Robert Könighofer, Görschwin Fey, and Roderick Bloem. SMT-based CPS parameter synthesis. In *ARCH@CPSWeek*, volume 43 of *EPiC Series in Computing*, pages 126–133. EasyChair, 2017.

[RPV17] Nima Roohi, Pavithra Prabhakar, and Mahesh Viswanathan. HARE: A hybrid abstraction refinement engine for verifying non-linear hybrid automata. In *TACAS*, volume 10205 of *LNCS*, pages 573–588, 2017.

[RTJB17] Andrew Reynolds, Cesare Tinelli, Dejan Jovanovic, and Clark Barrett. Designing theory solvers with extensions. In *FroCoS*, volume 10483 of *LNCS*. Springer, 2017.

[sca] SCADE Suite. `http://www.esterel-technologies.com/products/scade-suite`.

[Seb07] Roberto Sebastiani. Lazy satisability modulo theories. *JSAT*, 3(3-4):141–224, 2007.

[SH13] Alexey Solovyev and Thomas C. Hales. Formal verification of nonlinear inequalities with taylor interval approximations. In *NFM*, volume 7871 of *LNCS*, pages 383–397. Springer, 2013.

[sima] `https://www.mathworks.com/help/simulink/ug/managing-sample-times-in-systems.html`. Accessed: 2017-05-31.

[simb] Simulink. `https://www.mathworks.com/products/simulink.html`.

[Som] Fabio Somenzi. VIS Verification Benchmarks. `ftp://vlsi.colorado.edu/pub/vis/vis-verilog-models-1.3.tar.gz`.

[SS99] João P. Marques Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[SSS00] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, volume 1954 of *LNCS*, pages 108–125. Springer, 2000.

[ST15] Roberto Sebastiani and Silvia Tomasi. Optimization modulo theories with linear rational costs. *ACM Trans. Comput. Log.*, 16(2):12:1–12:43, 2015.

[STS$^+$10] Vassiliki Sfyrla, Georgios Tsiligiannis, Iris Safaka, Marius Bozga, and Joseph Sifakis. Compositional translation of simulink models into synchronous BIP. In *SIES*, pages 217–220. IEEE, 2010.

[Stu94] Bernd Sturmfels. Grobner bases - a computational approach to commutative algebra (thomas becker and volker weispfenning). *SIAM Review*, 36(2):323, 1994.

[Stu17] Thomas Sturm. A survey of some methods for real quantifier elimination, decision, and satisfiability and their applications. *Mathematics in Computer Science*, 11(3-4):483–502, 2017.

[Sug] Yoshihide Sugiura. AiPG RTL property checker. `http://www.revsonic.com/e/business/lsisolution/eda/aipg/`.

[TdHRZ17] Fatih Turkmen, Jerry den Hartog, Silvio Ranise, and Nicola Zannone. Formal analysis of XACML policies using SMT. *Computers & Security*, 66:185–203, 2017.

[Tiw15] Ashish Tiwari. Time-aware abstractions in HybridSal. In *CAV*, volume 9206 of *LNCS*, pages 504–510. Springer, 2015.

[TKO16] Vu Xuan Tung, To Van Khanh, and Mizuhito Ogawa. raSAT: An SMT solver for polynomial constraints. In *IJCAR*, volume 9706 of *LNCS*, pages 228–237. Springer, 2016.

[Ton09] Stefano Tonetta. Abstract model checking without computing the abstraction. In *FM*, volume 5850 of *LNCS*, pages 89–105. Springer, 2009.

[Tow07] E.J. Townsend. *Functions of a Complex Variable*. Read Books, 2007.

[TSCC05] Stavros Tripakis, Christos Sofronis, Paul Caspi, and Adrian Curic. Translating discrete-time Simulink to Lustre. *ACM Trans. Embedded Comput. Syst.*, 4(4):779–818, 2005.

[vce] VCEGAR Verification Benchmarks. `http://www.cprover.org/hardware/benchmarks/vcegar-benchmarks.tgz`.

[ver05] Verilog Register Transfer Level Synthesis. *IEC 62142-2005 First edition 2005-06 IEEE Std 1364.1*, 2005.

[ver06] IEEE Standard for Verilog Hardware Description Language. *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, 2006.

[VSS+96] Tiziano Villa, Gitanjali Swamy, Thomas Shiple, Adnan Aziz, Robert Brayton, Stephen Edwards, Gary Hachtel, Sunil Khatri, and Yuji Kukimoto. VIS users manual. *Electronics Research Laboratory, University of Colorado at Boulder*, 1996.

[WDF+09] Christoph Weidenbach, Dilyana Dimova, Arnaud Fietzke, Rohit Kumar, Martin Suda, and Patrick Wischnewski. SPASS version 3.5. In *CADE-22*, volume 5663 of *LNCS*, pages 140–145. Springer, 2009.

[Wei88] Volker Weispfenning. The complexity of linear problems in fields. *J. Symb. Comput.*, 5(1/2):3–27, 1988.

[Wei97] Volker Weispfenning. Quantifier elimination for real algebra - the quadratic case and beyond. *Appl. Algebra Eng. Commun. Comput.*, 8(2):85–101, 1997.

[Wol] Clifford Wolf. Yosys open synthesis suite. `http://www.clifford.at/yosys/`.

[WWH] Cheng-Yin Wu, Chi-An Wu, and Chung-Yang (Ric) Huang. V3. `http://dvlab.ee.ntu.edu.tw/~publication/V3/index.html`.

[YWH12] Hu-Hsi Yeh, Cheng-Yin Wu, and Chung-Yang Ric Huang. Qutertl: towards an open source framework for RTL design synthesis and verification. In *TACAS*. Springer, 2012.

[ZM10] Harald Zankl and Aart Middeldorp. Satisfiability of non-linear (ir)rational arithmetic. In *LPAR-16, 2010, Revised Selected Papers*, pages 481–500, 2010.