Comparing and Integrating CVC4 and Alt-Ergo

Hanwen Wu and Wenxin Feng

Department of Computer Science
Boston University
{hwwu,wenxinf}@bu.edu

May 6, 2013

Abstract

This technical report summarizes the abilities of CVC4 and Alt-Ergo by testing them using SMT-LIB 2.0 benchmarks within the theories of boolean, free functions, integers, bit-vectors, and quantifiers. An extra k-induction benchmarks are also tested. The results show that CVC4 is more powerful and efficient than Alt-Ergo. They both can support quantifiers and non-linear integer arithmetic in a limited way. For induction over integers, they both can handle, and Alt-Ergo outperform CVC4 for some inputs in this case. We also implement a lightweight frontend calling both solvers to get the first valid response as the result.

1 Overview

Our main goal is thoroughly characterizing and comparing the abilities of two different SMT solvers, CVC4[BCD$^+$11] version 1.1 and Alt-Ergo[BCC$^+$08] version 0.95.1. Since they both can take SMT-LIB 2.0[BST10a] as their input language, we will also summarize it.

In this report, we (1) give a short introduction on SMT-LIB logic, (2) formally classify input formulas using SMT-LIB logic, (3) introduce and summarize the two solvers, (4) carefully characterize and compare the abilities of the two solvers by testing them within different classes of formulas. And (5) integrate them using a lightweight frontend written in C programming language.

2 SMT-LIB 2.0 Logic

In this section, we briefly introduce SMT-LIB 2.0 logic so that we can more easily describe the classification of input formulas using SMT-LIB format later. It is developed as a standard logic to describe theories, input languages, and output languages, which is supported by a variety of SMT solvers. The SMT community also hold competitions for SMT solvers every year, and provides a collection of benchmarks that we can use for our tests.

2.1 Introduction

Since we use SMT-LIB 2.0 logic to describe the classification of formulas, it is necessary to understand the SMT-LIB 2.0 logic itself, which can be used as an input language for both solvers (Alt-Ergo, CVC4).

SMT-LIB 2.0 is basically a version of many-sorted first-order logic with equality[BST10a]. It enables us the ability to write formulas, define theories and logics, and interact with provers using scripts. Provers that support SMT-LIB 2.0 should implement required functionalities and use correct semantics.

2.1.1 Sets of Symbols

These are part of the sets defined by SMT-LIB 2.0[BST10b]. They are alphabets of the logic, namely, the sources of symbols. These symbols are used in the following subsections.

- $\mathcal{S}$: Infinite set of sort symbols, containing `bool`.
- $\mathcal{U}$: Infinite set of sort parameters.
- $\mathcal{X}$: Infinite set of variables.
- $\mathcal{F}$: Infinite set of function symbols.
- $\mathcal{B}$: Boolean values {**true, false**}.
- $\vdots$

2.1.2 Sorts

SMT-LIB 2.0 is a sorted logic. Sorts over a set of sort symbols $\mathcal{S}$ are defined as Sort($\mathcal{S}$). Sorts are defined inductively as follows.

- $\sigma \in \mathcal{S}$ of arity 0 is a sort.
- $\sigma, \sigma_1, \sigma_2, \sigma_3, \ldots, \sigma_n$ is a sort if $\sigma \in \mathcal{S}$ is of arity n, and σ_1 to σ_n are sorts.

The second item uses sorts to form new sorts. A list of integers can be a good example.

2.1.3 Signature

Basically, a signature Σ defines sort symbols and arities, function symbols and ranks, some variables and their sorts.

- $\Sigma^{\mathcal{S}} \subset \mathcal{S}$: sort symbols, containing `bool`.
- $\Sigma^{\mathcal{F}} \subset \mathcal{F}$: function symbols, containing equality, conjunction, and negation.
- $\Sigma^{\mathcal{S}}$ to $\mathbb{N}$: a total mapping from sort symbol to its arity, including $\texttt{bool} \Rightarrow \texttt{0}$.
- $\Sigma^{\mathcal{F}}$ to Sort($\Sigma^{\mathcal{S}}$)+: a left total mapping from a function symbol to its rank, containing $= (\sigma, \sigma, \texttt{bool})$, $\neg(\texttt{bool}, \texttt{bool})$, $\wedge(\texttt{bool}, \texttt{bool}, \texttt{bool})$.
- $\mathcal{X}$ to Sort($\Sigma^{\mathcal{S}}$): a partial mapping from a variable to its sort.

2.1.4 Formulas

In SMT-LIB 2.0, formulas are well sorted terms of sort `bool` over Σ. In actual scripts, all the formulas are being treated as closed formulas. This is possible since non-closed formulas can be quantified using existential quantifier, as far as its satisfiability is concerned.

2.1.5 Structure

A structure $\mathbf{A}$ in SMT-LIB 2.0 can be regarded as a model. It is defined as a tuple.

$$\mathbf{A} = \{A, \sigma^{\mathbf{A}1}, (f:\sigma)^{\mathbf{A}}, (f:\sigma_1,\sigma_2,\ldots,\sigma_n,\sigma)^{\mathbf{A}}\}$$

And the meaning of these four elements are as followings.

- A: the universe (of values) of $\mathbf{A}$, including $\texttt{bool}^{\mathbf{A}} = \{\textbf{true, false}\}$.
- $\sigma^{\mathbf{A}} \subset A$: gives the sort $\sigma \in \mathrm{Sort}(\Sigma^{\mathcal{S}})$ a universe $\sigma^{\mathbf{A}} \subset A$. For example, $\texttt{bool}^{\mathbf{A}}$ is $\{\textbf{true, false}\} \subset A$. $\texttt{int}^{\mathbf{A}}$ could be all the integers $\mathbb{Z} \subset A$.
- $(f:\sigma)^{\mathbf{A}} \in \sigma^{\mathbf{A}}$: gives the constant symbol $f:\sigma$ a value in the universe of σ
- $(f:\sigma_1,\sigma_2,\ldots,\sigma_n,\sigma)^{\mathbf{A}}$: defines the function symbol as a relation from $(\sigma_1,\sigma_2,...,\sigma_n)^{\mathbf{A}}$ to $\sigma^{\mathbf{A}}$. This must include the equality relations (or identity predicate over $\sigma^{\mathbf{A}}$, that is $= (\sigma,\sigma,\texttt{bool})$ as standard equality relations from $(\sigma^{\mathbf{A}},\sigma^{\mathbf{A}})$ to $\{\textbf{true, false}\}$).

2.1.6 Valuation and Interpretation

Valuation v is a partial mapping from $\mathcal{X} \times \mathrm{Sort}(\Sigma^{\mathcal{S}})$ to $\sigma^{\mathbf{A}}$. That is to give variable x of sort σ a value in $\sigma^{\mathbf{A}}$.

Interpretation $\mathcal{I}$ is defined as $\mathcal{I} = (\mathbf{A}, v)$, that is the structure together with the valuation make the Σ-interpretation.

$\mathcal{I}$ assigns meanings to well-sorted terms by uniquely mapping them into the $\mathbf{A}$. And that is the semantic.

As long as we have semantics, we can discuss satisfiability. If φ is mapped to **true** by some $\mathcal{I}$, then it is satisfiable. If φ is not closed, we say $\mathcal{I} = (\mathbf{A}, v)$ makes true φ. If φ is closed, we say the structure $\mathbf{A}$ makes true φ (since it doesn't matter what valuation it is), and that means $\mathbf{A}$ is a model of φ.

2.1.7 Theories

Theory is a very important concept. SMT stands for Satisfiability Modulo Theory, that is to check the satisfiability of a given logical formula within some background theories. Traditionally, a theory is a set of enough axioms, with which we can induct the formula. But here a theory $\mathcal{T}$ consists of three parts.

- Signature: Σ
- Models: A set of Σ-structures, all of which are models of the theory.
- Axioms: This is actually part of the models, and is left for the people who implement solvers. Take integer theory as an example. Since we have the plus sign in our signature (we just denote it as `ADD`, so that we know it is only a symbol, not the actual operation), we will have an axiom like $\forall x:\texttt{int}.\forall y:\texttt{int}.\exists z:\texttt{int}.\texttt{ADD}(x,y,z) \leftrightarrow x+y=z$. Therefore, our model (or structure) must contain the correct relations to map `ADD` to the actual addition operation to satisfy this axiom. Also, some theories, like real numbers, include those axioms as plain text, like associativity, commutativity, etc.

The SMT-LIB 2.0 standard has defined six theories. They are Core (for propositional logic), Integer, Real, Real and Integer, Fixed Size Bit-Vector, and Arrays. Each of them defines corresponding signature, and model. The actual implementations are left for the provers.

[1] $\sigma^{\mathbf{A}}$ is called the extension of σ in $\mathbf{A}$.

2.1.8 Logics

Logic in SMT-LIB is also very important. It is a sublogic of SMT-LIB logic with restrictions, and is based on some theories. Common restrictions are

- fixing a signature Σ and its theory $\mathcal{T}$
- restricting structures to the models of $\mathcal{T}$
- restricting input sentences as subset of Σ-sentences

The SMT-LIB standard classifies formulas into many well-defined logics, including QF-UF, QF-LIA, QF-NIA, QF-IDL, QF-LRA, QF-NRA, QF-RDL, QF-BV, QF-AX, etc. In the project, we focus on integers and fixed-size bit-vectors.

2.2 Theory

In the following, we present some abstract definition of different theories in SMT-LIB 2.0. Note that the Core theory is included in all other theories by default.

In all the figures, function symbols will only be applied to well-sorted terms according to their own function ranks/signatures/definitions.

2.2.1 Core Theory

Core Theory is about boolean sort and boolean functions/constants. It is the very base for all other theories.

Beyond propositional logic, there are two more features in the Core theory. The first is equality/distinction. These two function symbols are defined not only for `bool`, but also for all potential sorts in an expanded signature. The second is **ite**, which is the **if − then − else** operator. It is also defined for other sorts. See Table 1.

sort	α	::=	`bool`
function	f	::=	**true** : `bool` | **false** : `bool`
		|	(**not** `bool`) : `bool` | (**and** `bool bool`) : `bool`
		|	(**or** `bool bool`) : `bool`
		|	(**xor** `bool bool`) : `bool`
		|	($\Rightarrow$ `bool bool`) : `bool` | (= α α) : `bool`
		|	(**distinct** α α) : `bool` | (**ite** `bool` α α) : α
term	t	::=	**true** | **false**
		|	(**not** t) | (**and** t t) | (**or** t t) | (**xor** t t)
		|	($\Rightarrow$ t t) | (= t t) | (**distinct** t t) | (**ite** t t t)

Table 1: Core Theory

2.2.2 Integer Theory

Integer Theory defines the integer domain, and operations over integers. It is a superset of Core theory, thus includes all the sorts and function symbols defined in Core theory.

Note that the Integer theory itself does not have any restriction on linear or nonlinear operations, which should instead be defined in logics based on Integer theory. Also, the division, modulo operations here are defined for integers which actually involve flooring and ceiling. See Table 2.

sort	α	::=	`bool` | `int`
function	f	::=	...
		|	$\mathbb{Z}$: `int`
		|	($-$ `int`) : `int` | ($-$ `int int`) : `int`
		|	($+$ `int int`) : `int` | ($\times$ `int int`) : `int`
		|	(**div** `int int`) : `int` | (**mod** `int int`) : `int`
		|	(**abs** `int`) : `int`
		|	($\leqslant$ `int int`) : `bool` | ($<$ `int int`) : `bool`
		|	($\geqslant$ `int int`) : `bool` | ($>$ `int int`) : `bool`
		|	((_ **divisible** n) `int`) : `bool` (n is a positive integer)
term	t	::=	...
		|	... $-1, 0, 1$ ...
		|	($-$ t) | ($-$ t t) | ($+$ t t) | ($\times$ t t)
		|	(**div** t t) | (**mod** t t) | (**abs** t)
		|	($\leqslant$ t t) | ($<$ t t) | ($\geqslant$ t t) | ($>$ t t)
		|	((_ **divisible** n) t)

Table 2: Integer Theory

2.2.3 Fixed-Size Bit-Vectors Theory

This theory defines a series of sorts for different size of bit-vectors. Concatenation and extraction of bit-vectors, and the usual logical and arithmetic operations are also defined. The universe of bit-vectors theory is those numeral constants in bit-vector format. They are defined using a SMT-LIB syntax of the form `#b`X and `#x`X for binary and hexadecimal constants.

In Table 3, we use `bv` for (_ `BitVec` m), and omitting the size of the bit-vectors, only for layout reasons.

2.3 Logic

Logic is the main tool we use for classifying formulas and testing solver abilities. In the followings, we will formalize the definition of various logics of SMT-LIB.

2.3.1 Quantifier-Free Uninterpreted Functions

Closed quantifier-free formulas built over an arbitrary expansion of the Core signature with free sort and function symbols [BST10a]. Users can define their own sorts and function symbols, but all of them are abstract. Functions can contain variables, but they must be bounded by **let** binder, so that the formulas are closed. See Table 4.

2.3.2 Quantifier-Free Linear Integer Arithmetic

Closed quantifier-free formulas built over an arbitrary expansion of the Integer Theory with free *constant* symbols, but whose terms of sort `int` are all linear [BST10a]. Note that user can only define constants, not arbitrary functions who take one or more arguments. User can't define sort either. Also, non-linear functions like **div**, **mod**, **abs** and non-linear $\times$ are not allowed. See Table 5.

sort	α	::=	`bool`
		|	(`_ BitVec` m) (m is a positive integer, we use `bv` for short)
function	f	::=	...
		|	`#b`X : `bv` (all binary constants)
		|	`#x`X : `bv` (all hexadecimal constants)
		|	(**concat** `bv bv`) : `bv`
		|	((_ **extract** i j) `bv`) : `bv` (i, j specify the range)
		|	(**bvnot** `bv`) : `bv` | (**bvneg** `bv`) : `bv`
		|	(**bvand** `bv bv`) : `bv` | (**bvor** `bv bv`) : `bv`
		|	(**bvadd** `bv bv`) : `bv` | (**bvmul** `bv bv`) : `bv`
		|	(**bvudiv** `bv bv`) : `bv` | (**bvurem** `bv bv`) : `bv`
		|	(**bvshl** `bv bv`) : `bv` | (**bvlshr** `bv bv`) : `bv`
		|	(**bvult** `bv bv`) : `bool`
term	t	::=	...
		|	`#b`X (all binary constants)
		|	`#x`X (all hexadecimal constants)
		|	(**concat** t t) | ((_ **extract** i j) t)
		|	(**bvnot** t) | (**bvneg** t) | (**bvand** t t) | (**bvor** t t)
		|	(**bvadd** t t) | (**bvmul** t t) | (**bvudiv** t t) | (**bvurem** t t)
		|	(**bvshl** t t) | (**bvlshr** t t) | (**bvult** t t)

Table 3: Fixed-Size Bit-Vectors Theory

sort	α	::=	... | α' (α^*) (user defined, abstract)
function	f	::=	... | (f' α^*) : α (user defined, abstract)
term	t	::=	...
		|	(**let** (bindings$^+$) t)
		|	(f t^*)

Table 4: QF-UF Logic

2.3.3 Quantifier-Free Fixed-size Bit-Vectors

It is based on Fixed-sized Bit-Vectors theory, with an expension of user defined bit-vector constants. See Table 6.

2.3.4 Quantifier-Free Integers Difference Logic

It is a subset of integer theory with the restrictions that all the formulas should be in a differencial format as presented in Table 7.

sort	α	::=	`bool` | `int`
function	f	::=	… | b : `bool` | c : `int` (user defined constant)
term	t	::=	…
		|	$b \mid c$
		|	$(-\ t) \mid (-\ t\ t) \mid (+\ t\ t)$
		|	$(\times\ n\ t) \mid (\times\ t\ n)$ (n is an integer literal)
		|	$(\leqslant t\ t) \mid (< t\ t) \mid (\geqslant t\ t) \mid (> t\ t)$
		|	((_ **divisible** n) t)
		|	(**let** (bindings$^+$) t)

Table 5: QF-LIA Logic

sort	α	::=	… | (_ `BitVec` m) (m is a positive integer, we use `bv` for short)
function	f	::=	…
		|	… (same as bit-vectors theory)
		|	(**bvnand** `bv bv`) : `bv` | (**bvnor** `bv bv`) : `bv`
		|	(**bvxor** `bv bv`) : `bv` | (**bvxnor** `bv bv`) : `bv`
		|	(**bvcomp** `bv bv`) : (_ `BitVec` 1) | (**bvsub** `bv bv`) : `bv`
		|	(**bvdiv** `bv bv`) : `bv` | (**bvsrem** `bv bv`) : `bv`
		|	(**smod** `bv bv`) : `bv` | (**bvashr** `bv bv`) : `bv`
		|	((_ **repeat** i) `bv`) : `bv` ($i \geqslant 1$)
		|	((_ **zero_extend** i) `bv`) : `bv` ($i \geqslant 0$)
		|	((_ **sign_extend** i) `bv`) : `bv`
		|	((_ **rotate_left** i) `bv`) : `bv`
		|	((_ **rotate_right** i) `bv`) : `bv`
		|	(**bvule** `bv bv`) : `bool` | (**bvugt** `bv bv`) : `bool`
		|	(**bvuge** `bv bv`) : `bool` | (**bvslt** `bv bv`) : `bool`
		|	(**bvsle** `bv bv`) : `bool` | (**bvsgt** `bv bv`) : `bool`
		|	(**bvsge** `bv bv`) : `bool`
term	t	::=	…
		|	(**let** (bindings$^+$) t)
		|	$(f\ t^*)$

Table 6: QF-BV Logic

2.3.5 Quantifier-Free Non-Linear Integer Arithmetic

It is a superset of linear integer arithmetic, since there is no limitation for non-linear operations. See Table **??**.

2.3.6 Quantifier-Free Uninterpreted Functions over Fixed-size Bit-Vectors

It is a disjunction of empty theory and bit-vectors theory. See Table 9.

sort	α	$::=$	$\texttt{bool} \mid \texttt{int}$	
function	f	$::=$	$\ldots \mid b : \texttt{bool} \mid c : \texttt{int}$	(user defined constant)
term	t	$::=$	$\ldots$	
		$\mid$	$b \mid c$	
		$\mid$	$(\mathbf{op}\ (-\ c\ c)\ n)$	(n is a numeral; **op** is $<, \leqslant, >, \geqslant, =$ or **distinct**)
		$\mid$	$(\mathbf{op}\ (-\ c\ c)\ (-\ n)) \mid (\mathbf{op}\ c\ c)$	

Table 7: QF-IDL Logic

sort	α	$::=$	$\texttt{bool} \mid \texttt{int}$	
function	f	$::=$	$\ldots \mid b : \texttt{bool} \mid c : \texttt{int}$	(user defined constant)
term	t	$::=$	$\ldots$	
		$\mid$	$b \mid c$	
		$\mid$	$\ldots$	

Table 8: QF-NIA Logic

sort	α	$::=$	$\ldots \mid \alpha'\ (\alpha^*)$	(user defined, abstract)
function	f	$::=$	$\ldots \mid (f'\ \alpha^*) : \alpha$	(user defined, abstract)
term	t	$::=$	$\ldots$	
		$\mid$	$(f\ t^*)$	
		$\mid$	$\ldots$	(the same as QF-BV)

Table 9: QF-UFBV Logic

2.3.7 Quantifier-Free Uninterpreted Functions over Integer Difference Logic

For historical reasons, it is not a disjunction of empty theory and difference logics. There are more restrictions on the form of formulas which are presented in the form. See Table 10.

2.3.8 Quantifier-Free Uninterpreted Functions over Linear Integer Arithmetic

This is a disjunction of empty theory with linear integer arithmetic. See Table 11.

2.3.9 Uninterpreted Functions over Non-linear Integer Arithmetic

This is an expansion of non-linear integer arithmetic with empty theory and quantifiers. See Table 12.

$$
\begin{array}{rcll}
\text{sort} & \alpha & ::= & \texttt{bool} \mid \texttt{int} \mid \alpha'\ (\alpha^*) \qquad \text{(user defined, abstract)} \\[1ex]
\text{function} & f & ::= & \ldots \mid (f'\ \alpha^*) : \alpha \qquad \text{(user defined, abstract)} \\
& & \mid & \mathbb{Z} : \texttt{int} \\
& & \mid & (-\ \ \texttt{int int}) : \texttt{int} \mid (+\ \ \texttt{int int}) : \texttt{int} \\
& & \mid & (\leqslant\ \ \texttt{int int}) : \texttt{bool} \mid (<\ \ \texttt{int int}) : \texttt{bool} \\
& & \mid & (\geqslant\ \ \texttt{int int}) : \texttt{bool} \mid (>\ \ \texttt{int int}) : \texttt{bool} \\[1ex]
\text{term} & t & ::= & \ldots \\
& & \mid & (f\ t^*) \\
& & \mid & \ldots \quad -1, 0, 1 \quad \ldots \\
& & \mid & (-\ t\ c) \mid (-\ c\ t) \mid (+\ t\ c) \mid (+\ c\ t) \qquad (c \text{ is a numeral}) \\
& & \mid & (\leqslant t\ t) \mid (< t\ t) \\
& & \mid & (\geqslant t\ t) \mid (> t\ t)
\end{array}
$$

Table 10: QF-UFIDL Logic

$$
\begin{array}{rcll}
\text{sort} & \alpha & ::= & \ldots \mid \alpha'\ (\alpha^*) \qquad \text{(user defined, abstract)} \\[1ex]
\text{function} & f & ::= & \ldots \mid (f'\ \alpha^*) : \alpha \qquad \text{(user defined, abstract)} \\[1ex]
\text{term} & t & ::= & \ldots \\
& & \mid & (f\ t^*) \\
& & \mid & \ldots \qquad \text{(the same as QF-LIA)}
\end{array}
$$

Table 11: QF-UFLIA Logic

$$
\begin{array}{rcll}
\text{sort} & \alpha & ::= & \texttt{bool} \mid \texttt{int} \mid \alpha'\ (\alpha^*) \qquad \text{(user defined, abstract)} \\[1ex]
\text{function} & f & ::= & \ldots \mid b : \texttt{bool} \mid c : \texttt{int} \qquad \text{(user defined constant)} \\
& & \mid & (f'\ \alpha^*) : \alpha \qquad \text{(user defined, abstract)} \\[1ex]
\text{term} & t & ::= & \ldots \\
& & \mid & b \mid c \\
& & \mid & (\ \textbf{let}\ (\ \text{bindings}^+\ )\ t\ ) \\
& & \mid & (\ \textbf{forall}\ (\ \text{bindings}^+\ )\ t\ ) \\
& & \mid & (\ \textbf{exists}\ (\ \text{bindings}^+\ )\ t\ ) \\
& & \mid & (f\ t^*) \\
& & \mid & \ldots
\end{array}
$$

Table 12: UFNIA Logic

3 Comparing CVC4 and Alt-Ergo

In this section, we give a short summary of both solvers first. Their overall architectures, built-in theories, combination methods, and unique features are briefly discussed. Then we summarize our test methods and results to show their capabilities within different sub logics using SMT-LIB 2.0 benchmarks.

3.1 CVC4

CVC4, the fifth generation of Cooperating Validity Checker from NYU and U Iowa, is a DPLL(T) solver with a SAT solver core and a delegation path to different decision procedure implementations, each in charge of solving formulas in some background theory[BCD$^+$11]. It works for first-order logics. It has implemented decision procedures for the theory of uninterpreted/free functions, arithmetic(integer, real, linear, non-linear), arrays, bit-vectors and datatypes. It uses a combination method based on Nelson-Oppen to cooperate different theories. Also, CVC4 supports quantifiers through heuristic instantiaion[2] and has the ability to generate model. By our test results of k-induction over linear integer arithmetic, it supports induction very well.

For both satisfiable(`sat`)/unsatisfiable(`unsat`) formulas, CVC4 can come up with the correct answer.

3.2 Alt-Ergo

Alt-Ergo is dedicated to program verification. It works in first-order logic. It uses a CC(X)[3], a variant of Shostak algorithm, to combine free theory with equality and an arbitrary solvable built-in theory X[Con]. Alt-Ergo has implemented decision procedures for the theory of uninterpreted/free functions, arithmetic(integer, real, linear, non-linear), arrays, bit-vectors, datatypes, etc. It also has direct support for polymorphism in its native input language. Associative and commutative symbols are being handled specially using its AC(X) theory to boost the performance. It has limited support for universal and existential quantifiers through instantiation. It has the ability to generate proof. Also, by our test results of k-induction, Alt-Ergo can prove them quickly.

Alt-Ergo can handle `unsat` formulas correctly, but only returns `unknown` for `sat` formulas.

Since integer theory is intensively used in program verification, Alt-Ergo puts its efforts in the combination of empty/free theory with integer arithmetic theory. Alt-Ergo uses a Simplex-based extension of Fourier-Motzkin for solving linear integer arithmetic[BCC$^+$12].

3.3 Testing Both Solvers

3.3.1 Benchmarks

SMT-LIB community has been contributing benchmarks and holding competitions for several years[BST10a]. They can be considered as good standard benchmarks for SMT solvers. We have select integers, bit-vectors, and quantifiers related benchmarks for our tests. These tests are within QF-IDL, QF-NIA, QF-UFBV, QF-UFLIA, UFNIA, QF-BV, QF-LIA, QF-UF, QF-UFIDL.

We also select the Bounded Model Checking and k-induction problems within QF-LIA benchmarks from 2012 SMT Competition[4][CGB12] to test their abilities of applying inductive axioms.

All of the benchmarks are in the SMT-LIB 2.0 format, most of which should be handled by both solvers. But in practice, Alt-Ergo reports typing error and parsing error on some test inputs because its current version 0.95.1 does not fully support it yet. And also, some of the benchmarks do not indicate expected results. They are either `unknown` or not available at all. Therefore, we only considered those can be handled by both solvers, and had an expected answer of `sat/unsat` as valid benchmarks when we were analysing the result.

[2]See `http://cvc4.cs.nyu.edu/wiki/About_CVC4`

[3]CC(X): Congruence closure modulo X

[4]`http://smtcomp.sourceforge.net/2012/application.shtml`

3.3.2 Testing Methods

We wrote a testing script to run the tests. The script first randomly selected test inputs from all the benchmarks. Secondly, both solvers were invoked for each input, individually, not simultaneously. Thirdly, their execution time was captured by Unix `time` utility, and the real wall time was considered as their execution time. Finally, there was a timeout of 30 seconds. If any of them reached 30 seconds, it was killed.

Since Alt-Ergo cannot handle inputs with an expected answer of `sat`, we just skipped that for Alt-Ergo, and only use that input as a measure of efficiency/capability for CVC4 only.

For BMC and k-induction problems, we had a different strategy. Each of those benchmarks consists of multiple smaller problems, which require the solver to solve them incrementally. Therefore we set up a timeout of 15 seconds, to compare how much problems they can solve in a limited time.

We ran the tests on a Core i5-3320M 2.6GHz dual core CPU with 8GB memory, on Ubuntu 13.04 32bit operating system. During testing, no other job is allowed.

3.4 Testing Results

3.4.1 General

The general test results are shown in Table 13, Figure 6 and Figure 2 3 4 5 for `unsat` tests for both solvers, and Figure 7 for `sat` tests for CVC4 only. The timeout rate is relatively high in some divisions, due to the small 30 seconds timeout. And this is partly because we don't have enough computation resources to perform long enough tests. The error rate is also relatively high. For Alt-Ergo, this is mainly because it does not fully support SMT-LIB 2.0 standard, and particularly, its bit-vectors and typing parts. Alt-Ergo supports its native input language very well, but it seems not to be a good translation from SMT-LIB format into its native format.

For those parts which they both can handle, they generally respond quickly. For solver ability, CVC4 always wins with very high solved percentage. For solver efficiency, CVC4 is also better than Alt-Ergo.

In the following sections, we first compare them for those `unsat` inputs. And then, we talk about `sat` formulas for CVC4 only, and BMC and k-induction problems.

Logic	Total	Solver	Correct	Timeout	Exception	Unknown
QF-BV	16	Alt-Ergo	0	0	100%	0
		CVC4	81.25%	12.50%	6.25%	0
QF-IDL	106	Alt-Ergo	10.38%	67.92%	13.21%	8.49%
		CVC4	63.21%	36.79%	0	0
QF-LIA	143	Alt-Ergo	0	100%	0	0
		CVC4	95.80%	4.20%	0	0
QF-NIA	58	Alt-Ergo	0	0	96.55%	3.45%
		CVC4	60.34%	0	29.31%	10.34%
QF-UF	195	Alt-Ergo	36.41%	63.08%	0.51%	0
		CVC4	97.95%	2.05%	0	0
QF-UFBV	31	Alt-Ergo	0	0	100%	0
		CVC4	19.35%	80.65%	0	0
QF-UFIDL	91	Alt-Ergo	0	1.10%	98.90%	0
		CVC4	57.14%	42.86%	0	0
QF-UFLIA	163	Alt-Ergo	6.75%	6.75%	86.50%	0
		CVC4	95.71%	4.29%	0	0
UFNIA	1562	Alt-Ergo	9.80%	29.64%	60.37%	0.19%
		CVC4	40.59%	31.82%	27.59%	0

Table 13: Capabilities of CVC4 and Alt-Ergo for `unsat` Tests

Logic	Total	Solver	Correct	Avg.Time (sec)
QF-BV	16	Alt-Ergo	0	
		CVC4	13	16.31
QF-IDL	106	Alt-Ergo	11	2.56
		CVC4	67	2.80
QF-LIA	143	Alt-Ergo	0	
		CVC4	137	4.05
QF-NIA	58	Alt-Ergo	0	
		CVC4	35	0.10
QF-UF	195	Alt-Ergo	71	9.51
		CVC4	191	1.04
QF-UFBV	31	Alt-Ergo	0	
		CVC4	6	4.02
QF-UFIDL	91	Alt-Ergo	0	
		CVC4	52	3.43
QF-UFLIA	163	Alt-Ergo	11	5.07
		CVC4	156	0.70
UFNIA	1562	Alt-Ergo	153	3.08
		CVC4	634	2.57

Table 14: Efficiencies of CVC4 and Alt-Ergo for `unsat` Tests

3.4.2 QF-BV Division

In this division, there are only 16 test inputs which expect `unsat` answers. CVC4 solved 13 of them in an average of 16.31 seconds. CVC4 has 2 timeout, and 1 exception. The only exception is for input file `src_wget_vc18197.smt2`[5], which expects an `unsat` answer but CVC4 answers `sat`. This is the only wrong answer among all the tests. We double checked the answer using Z3[DMB08], it returns `unsat`. We have contacted the CVC4 team, and the reason is due to the semantics of bit-vectors divide-by-zero. CVC4 is now fully support the revised SMT-LIB standard which defines the semantics of it clearly enough. While the benchmark result is computed by not-up-to-date solvers which have their own non-standard semantics, like Z3. In a word, the benchmark itself might be revised to be `sat` in the near future, and therefore this is not considered as a wrong answer for CVC4 anymore.

Alt-Ergo does not fully support the SMT-LIB 2.0 language for bit-vectors, and reports `Smtlib2_to_why.Not_Implemented` for all the inputs in this division, which means it cannot translate SMT-LIB 2.0 input language into its native Why3 language[BFMP11] for now.

Because of this situation, we cannot compare their abilities in this division. But for CVC4, it can solve most of the problems, but in a very long time compared to other divisions, most of which takes less than five seconds.

3.4.3 QF-IDL Division

In this division, there are 106 valid benchmarks in total. CVC4 shows very strong ability compared with Alt-Ergo, with 52.83% more correctly solved tests, and all those 10.38% tests solved by Alt-Ergo are also correctly solved by CVC4. They both have relatively high timeout rate compared to other divisions, and Alt-Ergo has more time out than CVC4.

Alt-Ergo has 13.21% exceptions, five of them report syntax error during parsing, others are out of memory exceptions. The syntax errors are caused also by the limited support for SMT-LIB 2.0 format, the out of memory exceptions are probably caused by both the size of input formulas and the limitation of testing environment.

[5]`http://smtexec.org/exec/smtlib-portal-benchmarks.php?download&inline&b=QF_BV%2Fspear%2Fwget_v1.10.2%2Fsrc_wget_vc18197.smt2`

The timeout rate and out of memory exceptions show that some QF-IDL problems in the benchmark are more difficult for solvers. But for those they can solve, they are quick. CVC4 solve 67 problems with an average of 2.80s, and Alt-Ergo solve 11 using 2.56s in average. Although Alt-Ergo is more quickly, but its ability is very limited.

3.4.4 QF-LIA Division

Linear integer arithmetic is a well developed area. Decision procedures for it have long been investigated, and are decidable (at least for quantifier free fragments). CVC4 is an absolute winner. With its DPLL(T) plus Simplex based decision procedure[BCD$^+$11], it not only solves 95.80% problems, but also uses only 4.05s per problem on average.

Unfortunately, Alt-Ergo timeouts all the tests here, which is out of our expectation. We have inspected the reason, and it turns out that the input size is very big. Most of them contain more than 500 user defined constants, and the formula is very big as well. We plan to add some more tests for Alt-Ergo in QF-LIA in the near future. But for now, The newly designed expression subsystem of CVC4[BCD$^+$11] plays an important role on dealing with large input formulas.

3.4.5 QF-NIA Division

Non-linear integer arithmetic problems are much harder than linear problems. But CVC4 can still handle 60.34% of our tests, which is good enough. CVC4 reports errors on some inputs, which is an assertion error in the `DioSolver` component under `theory/arith` directory of CVC4 source code. But we don't have an explanation for the error yet. CVC4 also reports 6 `unknown`, but neither of them takes more than 1 second.

Alt-Ergo throws exceptions on all tests. It reports typing errors for the condition part of normal `ite` expressions which surprises us. It might be the limited support of SMT-LIB.

For those 35 solved problems by CVC4, it is extremely fast. This is partly due to the small input size. But the overall capability and efficiency is quite good.

3.4.6 QF-UF Division

Quantifier free empty theory with equality is also decidable. Both solvers perform very good in this division. Alt-Ergo solves 36.41% using 9.51s on average, while CVC4 solves 97.95% using 1.04s on average. CVC4's solved problems are a superset of Alt-Ergo's, and CVC's timeout problems are a subset of Alt-Ergo's timeout problems.

Unfortunately, Alt-Ergo timeouts a lot, but compared to its capability in other divisions, Alt-Ergo actually performs much better here. Alt-Ergo has one out of memory exception. That is a result of input size. On the same problem, CVC4 also takes 20.92 seconds to return the answer.

3.4.7 QF-UFBV Division

Compared to QF-BV and QF-UF divisions, CVC4 downgrades a lot. It only solves 19.35% of the problems, and takes 4.02s on average. All other problems are timeout, and the timeout rate is the second highest among all divisions for CVC4. That reflects the fact that theory combination and the bit-vectors are more challenging for solvers.

As the same as QF-BV, Alt-Ergo reports parsing errors due to the limited support of SMT-LIB standard.

3.4.8 QF-UFIDL Division

CVC4 again performs very well in this division, with 57.14% solved problems in an average of 3.43s. Its 42.86% timeout ratio is a little bit high. But overall, it is very good.

Alt-Ergo faces out of memory exceptions after about an average of about 20 seconds of computation for all of the problems, except for one that is reported as timeout by both of the solvers. Although these test inputs are quite big with more than 500 variable bindings, this is still not a good result for Alt-Ergo, since CVC4 can handle them very quickly. We do not know the internal problem with Alt-Ergo, but it might be the reason that it is implemented using functional programming language OCaml, which might be problematic in memory management. We will cover this issue in the conclusion section.

3.4.9 QF-UFLIA Division

In this division, CVC4 performs very good. It solves 95.71% problems, which is a superset of those 6.75% solved by Alt-Ergo. CVC4 takes 0.70s on average, while Alt-Ergo takes 5.07s.

CVC4 has several timeout, all of which come from the Wisconsin Safety Analyzer benchmarks in the SMT-LIB benchmarks package for QF-UFLIA. They are not big, with no more than 300 variables or bindings.

Alt-Ergo again fails on the condition part of `ite` expressions. The condition is in the form of linear inequality.

3.4.10 UFNIA Division

In this division, problems get harder by allowing quantifiers in addition to non-linear arithmetic. CVC4 solves 40.59%, while Alt-Ergo solves only 9.80%. But they are actually relatively quick, for an average of 2.57s and 3.08s respectively. CVC4 again solves a superset of problems solved by Alt-Ergo.

They both timeout a lot, and throws many exceptions. For CVC4, its exceptions come from a unexpected token `repeat` as reported by CVC4, which is actually a syntactically correct user defined free function symbol. This token might be conflict with some internal keyword, which causes CVC4 to fail. For the same input, Alt-Ergo doesn't report the issue reported by CVC4, which means that is a potential bug of CVC4. Alt-Ergo, on the other hand, fails on the `ite` expression parsing and typing.

But compared to other divisions, Alt-Ergo performs good, since it does not downgraded a lot like CVC4 does. It can solve some problems, and with high efficiency.

3.4.11 Tests with `sat` Expected Answers

For those `sat` tests, we only test CVC4 since Alt-Ergo cannot handle them directly. Among all logic divisions, CVC4 performs very good in QF-UF, QF-LIA, QF-UFIDL and QF-UFLIA as shown in Figure 7. In QF-NIA division, it throws assertion failure in its `DioSolver` component as mentioned previously in QF-NIA `unsat` tests. It reports `unknown` a lot within 1 second in QF-NIA, most of which are small formulas with around 30 variables. We have double checked them using Z3, and it reports `sat` quickly.

3.4.12 Tests with Bounded Model Checking and k-Induction Problems

In this devision, we use a 15-second timeout limit to see how many problems they can correctly solve. These benchmarks and the answers come from SMT-COMP 2012 benchmarks. They are within the logic of QF-LIA. We select only `unsat` problems from it to test both solvers.

k-Induction problem is a generalization of induction problems with k base cases. These benchmarks test the ability whether solvers can apply inductive axioms to some specific terms.

We test 40 `unsat` benchmarks, and each benchmark contain multiple smaller problems. CVC4 can solve 9.0 problems per second on average, and Alt-Ergo can do 4.2 per second. Among these benchmarks, CVC4 wins 18 of them, and Alt-Ergo wins 19 of them, shown in Figure 1.

Generally, Alt-Ergo performs very well in this division.

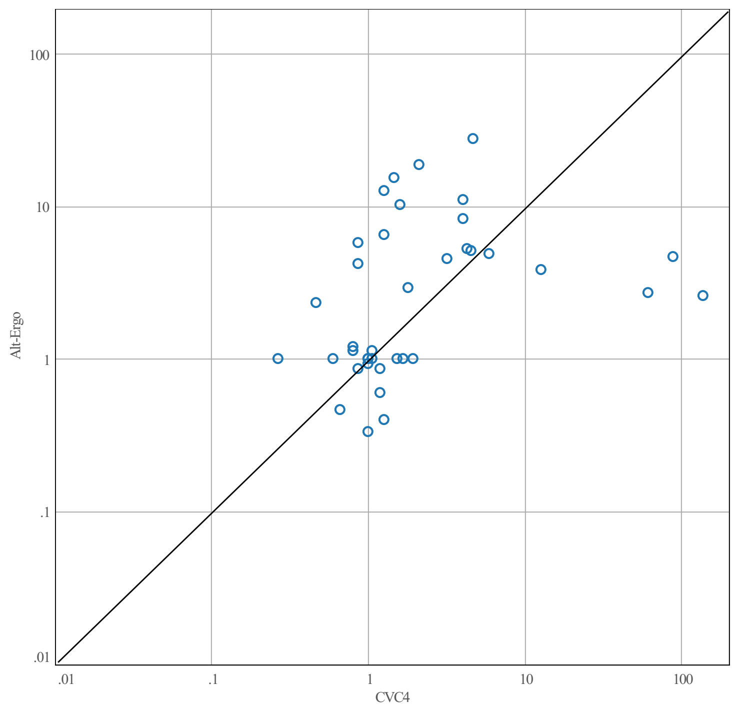

Figure 1: BMC and k-Induction Problems (correct answers per second)

4 Conclusion

CVC4 as a DPLL(T) solver, implemented in C++, is very powerful and effective, especially in QF-UF, QF-LIA, QF-UFIDL, and QF-UFLIA. It works in QF-BV, QF-UFBV and QF-IDL, but not well enough. It has limited support for QF-NIA, and even more limited for UFNIA. It is very efficient in all divisions except QF-BV.

In other words, its theory implementation for free functions and linear integer arithmetic are particularly good. Bit-vectors theory is not as good as previous ones, and quantifiers and non-linear arithmetic are not supported well.

It has good enough support for BMC and k-induction problems, which means it has the capability to apply inductive axioms very well.

CVC4 handles large inputs very well, even for input formulas with more than 500 variables and bindings.

Alt-Ergo as a CC(X) solver, implemented using functional language OCaml, performs good in QF-UF. It supports its own Why3 input language, but only supports SMT-LIB 2.0 in a very limited way. According to its performance, it should be good in QF-IDL, QF-UFLIA as well, if it can translate SMT-LIB language better. It runs relatively good enough in UFNIA division, which may due to its AC(X) theory that can quickly solve some non-linear integer arithmetic that only involves associative and commutative properties.

In BMC and k-induction problems, Alt-Ergo performs very well, which reflects the fact that it is designed for program verification. It has a very good ability to apply inductive axioms.

Alt-Ergo has problems dealing with large input, which is partly due to the case that it is implemented using a functional language OCaml. Compared to C++, OCaml is not good at memory management, which causes Alt-Ergo to run out of memory a lot.

Comparing the two solvers, CVC4 wins in both capabilities and efficiency. Its solvable problems are a superset of Alt-Ergo's. And Alt-Ergo can't handle `sat` input. But Alt-Ergo performs good enough considering that CVC4 is implemented in C++ while Alt-Ergo is in OCaml.

5 Integrating CVC4 and Alt-Ergo

Based on our tests, we found that it is not worth enough to integrate them into one solver. Therefore, we implemented a C frontend that calls both solvers using SMT-LIB scripts, and return the first `sat/unsat` response and kills the other solver. The solver is neither more powerful nor more efficient as expected, which is nearly identical to CVC4 alone.

6 Future Work

Currently, we do not have benchmarks in Why3 format, which is the native language for Alt-Ergo. But we plan to contact the authors of Alt-Ergo to get more detailed information of it. Alt-Ergo demonstrats good ability in solving verification related problems in free theory and linear arithmetic. We believe it has good enough potential ability in its native language.

For bit-vectors, CVC4 does not perform very well, and we plan to investigate the reason more carefully in the future.

For those exceptions happened in the tests, we still have some unresolved parts, which needs more time to get a response from the authors of provers. We plan to follow up with those authors to clearify them.

7 Acknowledgements

We would like to thank Professor Assaf Kfoury, and Andrei Lapets, who gave us the big picture of this area, and helped answer important questions on k-induction. Also thank William Blair and Yilei Chen, who constantly exchanged ideas with us. Without all of your help, we can't get it done smoothly.

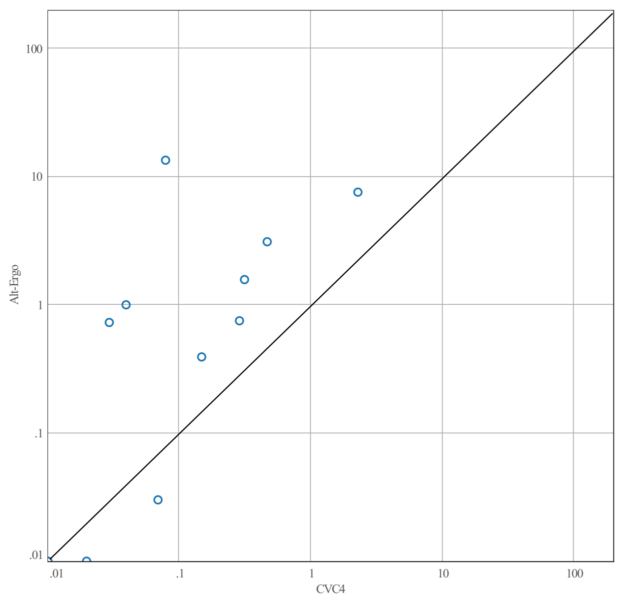

Figure 2: QF-IDL (avg. time, log scale)

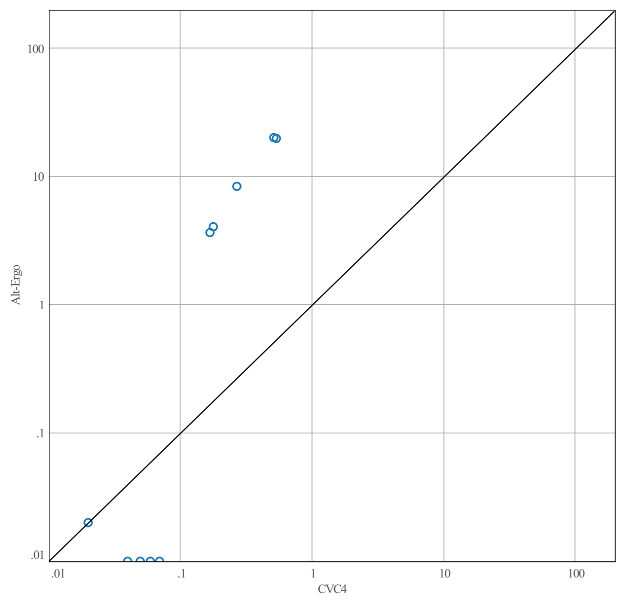

Figure 4: QF-UFLIA (avg. time, log scale)

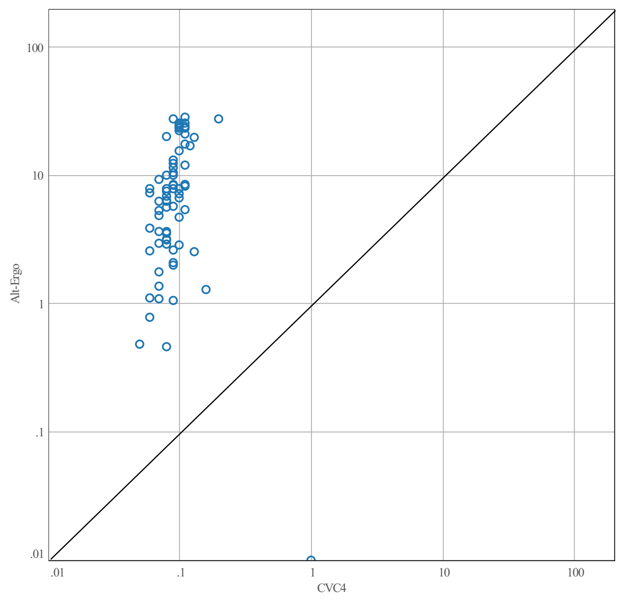

Figure 3: QF-UF (avg. time, log scale)

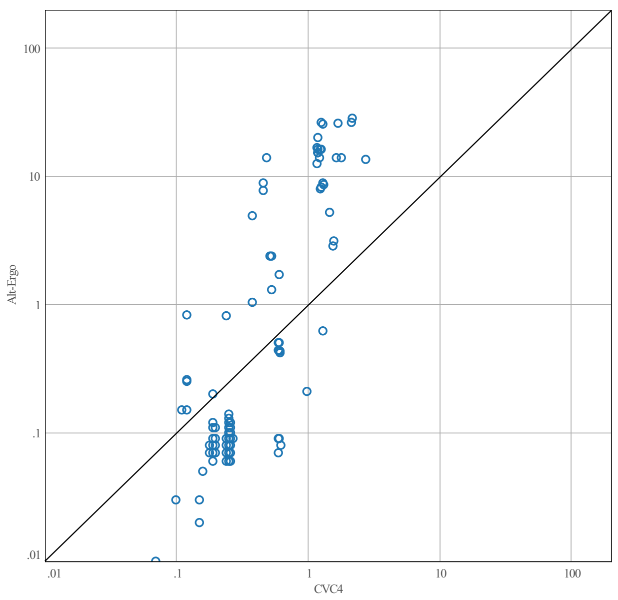

Figure 5: UFNIA (avg. time, log scale)

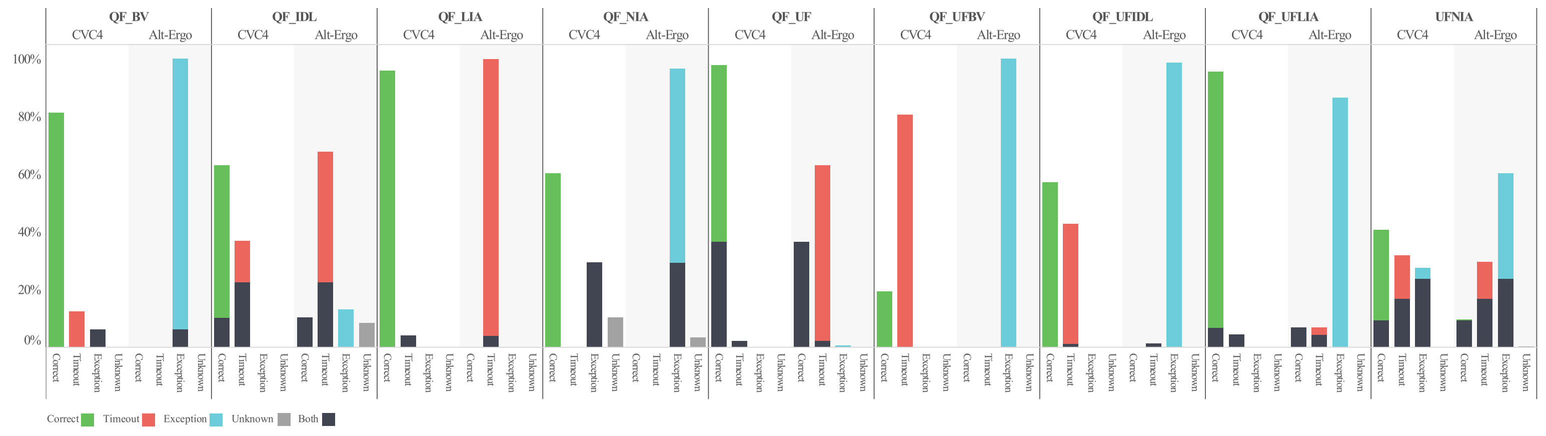

Figure 6: Capabilities of CVC4 and Alt-Ergo on `unsat` Tests

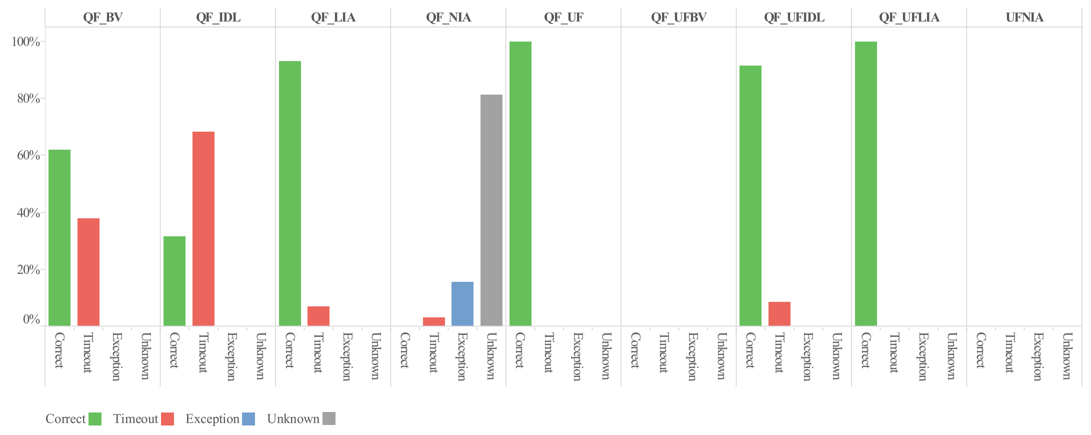

Figure 7: Capabilities of CVC4 on `sat` Tests

References

[BCC+08] François Bobot, Sylvain Conchon, Évelyne Contejean, Mohamed Iguernelala, Stéphane Lescuyer, and Alain Mebsout. The Alt-Ergo automated theorem prover. 2008. `http://alt-ergo.lri.fr/`.

[BCC+12] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A Simplex-Based Extension of Fourier-Motzkin for Solving Linear Integer Arithmetic. In Bernhard Gramlich, Dale Miller, and Uli Sattler, editors, *6th International Joint Conference on Automated Reasoning*, volume 7364 of *Lecture Notes in Computer Science*, pages 67–81, Manchester, Royaume-Uni, 2012. Springer.

[BCD+11] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovi, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *Proceedings of the 23rd international conference on Computer aided verification*, CAV'11, pages 171–177, Berlin, Heidelberg, 2011. Springer-Verlag.

[BFMP11] François Bobot, Jean-Christophe Fillitre, Claude March, and Andrei Paskevich. *The Why3 platform.* LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, 2011.

[BST10a] Clark Barrett, Aaron Stump, and Cesare Tinelli. The satisfiability modulo theories library (SMT-LIB). *www.smtlib.org*, 2010.

[BST10b] Clark Barrett, Aaron Stump, and Cesare Tinelli. The smt-lib standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, volume 13, 2010.

[CGB12] David R Cok, Alberto Griggio, and Roberto Bruttomesso. The 2012 smt competition. 2012.

[Con] Sylvain Conchon. *SMT Techniques and their Applications: from Alt-Ergo to Cubicle.* PhD thesis.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.