SMTFrontEnd Documentation

Release 0.1

Hanwen Wu, Wenxin Feng

May 10, 2013

CONTENTS

This is a course project for Formal Methods. It aims at designing and implementing a lightweight front end for various SMT solvers. Currently, we are going to support Alt-Ergo and CVC4.

CHAPTER

ONE

CONTENTS

1.1 CVC4

1.1.1 Introduction

CVC4, the fifth generation of Cooperating Validity Checker from NYU and U Iowa, is a DPLL 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. 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 instantiaionfootnote and has the ability to generate model. For both satisfiable(sat)/unsatisfiable(unsat) formulas, CVC4 can come up with the correct answer.

1.1.2 Building

Since ANTLR has been largely changed, the building process listed here should be changed a little bit.

1. Download source code from official links.
2. Following the building instruction to build the CVC4.
3. Enter `contrib` directory, use `get-antlr-3.4` to get ANTLR.

   > **Note:** The `get-antlr-3.4` file should be changed. All the hyperlinks including "**antlr.org**" should be changed to "**antlr3.org**".

4. Use `./configure --with-antlr-dir=`pwd`/antlr-3.4 ANTLR=`pwd`/antlr-3.4/bin/antlr3` to do configuration.
5. Following the rest steps in the building instruction. If the `configure` reports missing something, just install them all.
6. Make

1.1.3 Using

To invoke command line interface, just type

```
./cvc4 scriptfile.smt2
```

It will use the correct parser based on file extensions. If you want to test all the scripts in a folder, try this

```
ls | xargs -n 1 cvc4
```

1.1.4 Kinds of Expressions

From Built-in Theory

Kind	Meaning
SORT_TAG	sort tag
SORT_TYPE	sort type
UNINTER-PRETED_CONSTANT	The kind of expressions representing uninterpreted constants
ABSTRACT_VALUE	The kind of expressions representing abstract values (other than uninterpreted sort constant
BUILTIN	The kind of expressions representing built-in operators
FUNCTION	function
APPLY	defined function application
EQUAL	equality
DISTINCT	disequality
VARIABLE	variable
BOUND_VARIABLE	bound variable
SKOLEM	skolem var
SEXPR	a symbolic expression
LAMBDA	lambda
CHAIN	chain operator
TYPE_CONSTANT	basic types
FUNCTION_TYPE	function type
SEXPR_TYPE	symbolic expression type
CONST_STRING	a string of characters
SUBTYPE_TYPE	predicate subtype

From Boolean Theory

Kind	Meaning
CONST_BOOLEAN	truth and falsity
NOT	logical not
AND	logical and
IFF	logical equivalence
IMPLIES	logical implication
OR	logical or
XOR	exclusive or
ITE	if-then-else

From UF Theory

Kind	Meaning
APPLY_UF	uninterpreted function application
CARDINALITY_CONSTRAINT	cardinality constraint

From Arithmatic Theory

Kind	Meaning
PLUS	arithmetic addition
MULT	arithmetic multiplication
MINUS	arithmetic binary subtraction operator
UMINUS	arithmetic unary negation
DIVISION	real division (user symbol)
DIVISION_TOTAL	real division with interpreted division by 0 (internal symbol)
INTS_DIVISION	ints division (user symbol)
INTS_DIVISION_TOTAL	ints division with interpreted division by 0 (internal symbol)
INTS_MODULUS	ints modulus (user symbol)
INTS_MODULUS_TOTAL	ints modulus with interpreted division by 0 (internal symbol)
POW	arithmetic power
SUBRANGE_TYPE	the type of an integer subrange
CONST_RATIONAL	a multiple-precision rational constant
LT	less than, x < y
LEQ	less than or equal, x <= y
GT	greater than, x > y
GEQ	greater than or equal, x >= y

From Array Theory

Kind	Meaning
ARRAY_TYPE	array type
SELECT	array select
STORE	array store
STORE_ALL	array store-all
ARR_TABLE_FUN	array table function (internal symbol)

From BitVector Theory

Kind	Meaning
BITVECTOR_TYPE	bit-vector type
CONST_BITVECTOR	a fixed-width bit-vector constant
BITVECTOR_CONCAT	bit-vector concatenation
BITVECTOR_AND	bitwise and
BITVECTOR_OR	bitwise or
BITVECTOR_XOR	bitwise xor
BITVECTOR_NOT	bitwise not
BITVECTOR_NAND	bitwise nand
BITVECTOR_NOR	bitwise nor
BITVECTOR_XNOR	bitwise xnor
BITVECTOR_COMP	equality comparison (returns one bit)
BITVECTOR_MULT	bit-vector multiplication
BITVECTOR_PLUS	bit-vector addition
BITVECTOR_SUB	bit-vector subtraction
BITVECTOR_NEG	bit-vector unary negation
BITVECTOR_UDIV	bit-vector unsigned division, truncating towards 0 (undefined if divisor is 0)

Continued on next page

Table 1.1 – continued from previous page

Kind	Meaning
BITVECTOR_UREM	bit-vector unsigned remainder from truncating division (undefined if divisor is 0)
BITVECTOR_SDIV	bit-vector 2's complement signed division
BITVECTOR_SREM	bit-vector 2's complement signed remainder (sign follows dividend)
BITVECTOR_SMOD	bit-vector 2's complement signed remainder (sign follows divisor)
BITVECTOR_UDIV_TOTAL	bit-vector total unsigned division, truncating towards 0 (undefined if divisor is 0)
BITVECTOR_UREM_TOTAL	bit-vector total unsigned remainder from truncating division (undefined if divisor is 0)
BITVECTOR_SHL	bit-vector left shift
BITVECTOR_LSHR	bit-vector logical shift right
BITVECTOR_ASHR	bit-vector arithmetic shift right
BITVECTOR_ULT	bit-vector unsigned less than
BITVECTOR_ULE	bit-vector unsigned less than or equal
BITVECTOR_UGT	bit-vector unsigned greater than
BITVECTOR_UGE	bit-vector unsigned greater than or equal
BITVECTOR_SLT	bit-vector signed less than
BITVECTOR_SLE	bit-vector signed less than or equal
BITVECTOR_SGT	bit-vector signed greater than
BITVECTOR_SGE	bit-vector signed greater than or equal
BITVECTOR_BITOF_OP	operator for the bit-vector boolean bit extract
BITVECTOR_EXTRACT_OP	operator for the bit-vector extract
BITVECTOR_REPEAT_OP	operator for the bit-vector repeat
BITVECTOR_ZERO_EXTEND_OP	operator for the bit-vector zero-extend
BITVECTOR_SIGN_EXTEND_OP	operator for the bit-vector sign-extend
BITVECTOR_ROTATE_LEFT_OP	operator for the bit-vector rotate left
BITVECTOR_ROTATE_RIGHT_OP	operator for the bit-vector rotate right
BITVECTOR_BITOF	bit-vector boolean bit extract
BITVECTOR_EXTRACT	bit-vector extract
BITVECTOR_REPEAT	bit-vector repeat
BITVECTOR_ZERO_EXTEND	bit-vector zero-extend
BITVECTOR_SIGN_EXTEND	bit-vector sign-extend
BITVECTOR_ROTATE_LEFT	bit-vector rotate left
BITVECTOR_ROTATE_RIGHT	bit-vector rotate right

From Datatype Theory

Kind	Meaning
CONSTRUCTOR_TYPE	constructor
SELECTOR_TYPE	selector
TESTER_TYPE	tester
APPLY_CONSTRUCTOR	constructor application
APPLY_SELECTOR	selector application
APPLY_TESTER	tester application
DATATYPE_TYPE	datatype type
PARAMETRIC_DATATYPE	parametric datatype
APPLY_TYPE_ASCRIPTION	type ascription, for datatype constructor applications
ASCRIPTION_TYPE	a type parameter for type ascription
TUPLE_TYPE	tuple type
TUPLE	a tuple
TUPLE_SELECT_OP	operator for a tuple select
TUPLE_SELECT	tuple select
TUPLE_UPDATE_OP	operator for a tuple update
TUPLE_UPDATE	tuple update
RECORD_TYPE	record type
RECORD	a record
RECORD_SELECT_OP	operator for a record select
RECORD_SELECT	record select
RECORD_UPDATE_OP	operator for a record update
RECORD_UPDATE	record update

From Quantifier Theory

Kind	Meaning
FORALL	universally quantified formula
EXISTS	existentially quantified formula
INST_CONSTANT	instantiation constant
BOUND_VAR_LIST	bound variables
INST_PATTERN	instantiation pattern
INST_PATTERN_LIST	instantiation pattern list

From RewriteRule Theory

Kind	Meaning
REWRITE_RULE	generale rewrite rule
RR_REWRITE	actual rewrite rule
RR_REDUCTION	actual reduction rule
RR_DEDUCTION	actual deduction rule

1.1.5 Built-in Atomic Types

Type	Meaning
BUILTIN_OPERATOR_TYPE	Built in type for built in operators
STRING_TYPE	String type
BOOLEAN_TYPE	Boolean type
REAL_TYPE	Real type
INTEGER_TYPE	Integer type
BOUND_VAR_LIST_TYPE	Bound Var type
INST_PATTERN_TYPE	Instantiation pattern type
INST_PATTERN_LIST_TYPE	Instantiation pattern list type
RRHB_TYPE	head and body of the rule type

1.1.6 Theories

ID	Meaning
THEORY_BUILTIN	
THEORY_BOOL	
THEORY_UF	
THEORY_ARITH	
THEORY_ARRAY	
THEORY_BV	
THEORY_DATATYPES	
THEORY_QUANTIFIERS	
THEORY_REWRITERULES	

1.2 AltErgo

1.2.1 Introduction

Alt-Ergo is dedicated to program verification. It works in first-order logic. It uses a CC(X), a variant of Shostak algorithm, to combine free theory with equality and an arbitrary solvable built-in theory X. 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. 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.

1.2.2 Building

It depends largely on OCaml, so during configuration and making, if they report missing something, google that and install related packages. Most of them are OCaml related, and try to google `ocamlfind`, `ocaml-core`, `typeconv` for more information.

1.2.3 Using

To invoke command line interface, just type

```
./alt-ergo.opt scriptfile.smt2
```

to execute a SMT-LIB v2.0 script file. AltErgo will convert it into its native language, and then execute it. The result will be printed on the standard output.

To invoke GUI, just type

```
./altgr-ergo.opt scriptfile.smt2
```

to open it. If the file is successfully parsed and translated, then the GUI will open. Otherwise, it exits.

Note: It may take a very long time for either way to process the whole script file.

1.3 The SMT-LIB Language

The SMT-LIB Language uses *S-expression*, which is actually legal Common Lisp syntax. It has three main components: *theory declarations*, *logic declarations*, and *scripts*.

They contain only ASCII characters, not Unicode.

The followings are a preview of the ANTLR v4 grammar of SMTLIB v2.0 by Hanwen Wu, according to [BaST10]. (And, subject to change.)

1.3.1 A Summary of SMT-LIB Logic 2.0

We are working on integrating two SMT solvers, therefore it is necessary to understand the SMT-LIB 2.0 logic, which can be used as an input language for both solvers (AltErgo, CVC4). Part of the project is to classify formulas which can be solved by one solver, while not by the others. Therefore knowing how theories and logics are defined in SMT-LIB is also important.

SMT-LIB Logics

- $\mathcal{S}$: Infinite set of sort symbols, containing BOOL.
- $\mathcal{V}$: Infinite set of sort parameters
- $\mathcal{X}$: Infinite set of variables
- $\mathcal{F}$: Infinite set of function symbols
- $\mathcal{B}$: Boolean values {true, false}
- ...

Sorts

Sorts over a set of sort symbols $\mathcal{S}$ are defined as *Sort* ($\mathcal{S}$).

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

Signature

Baiscly, 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 : a total mapping from sort symbol to its arity, including $\models$
- $\Sigma^{\mathcal{F}}$ to *Sort* ($\Sigma^{\mathcal{S}}$)+: a left total mapping from a function symbol to its rank, containing = (σ, σ ,BOOL), $\neg$ (BOOL,BOOL), $\wedge$ (BOOL,BOOL,BOOL).
- $\mathcal{X}$ to *Sort* ($\Sigma^{\mathcal{S}}$): a partial mapping from a variable to its sort.

Formulas

Well sorted terms of sort BOOL over Σ.

Structure

A can be regarded as a model.

- A: the universe (of values) of A, including *BOOL* ^{A} ={true,false}.
- $\sigma^A \subset A$: give the sort $\sigma \in$ *Sort* ($\Sigma^{\mathcal{S}}$) a universe $\sigma^A \subset A$. For example, *BOOL* ^{A} is {true,flase} $\in A$. *INT* ^{A} could be all the integers $Z \in A$.
- $(f : \sigma)^A \in \sigma^A$: give the constant symbol f : a value in the universe of σ
- $(f : \sigma_1, \sigma_2, ..., \sigma_n, \sigma)^A$: define the function symbol as a relation from $(\sigma_1, \sigma_2, ..., \sigma_n)^A$ to σ^A. This must include the equality relations (or identity predicate over σ^A, that is =(σ, σ,BOOL) as standard equality relations from (σ^A, σ^A) to {true,false}).

σ^A is called the extension of σ in A.

Valuation and Interpretation

- Valuations v: a partial mapping v from $\mathcal{X}\times$ *Sort* ($\Sigma^{\mathcal{S}}$) to σ^A. That is to give variable x of sort σ a value in σ^A.
- Interpretation $\mathcal{I}$: $\mathcal{I} = (A, v)$, that is the structure together with the valuations make the Σ-interpretation.
- Semantics: $\mathcal{I}$ will assign a meaning to well-sorted terms by uniquely mapping them into the A.
- **Satisfiability:**
  - If φ is mapped to true by some $\mathcal{I}$, then it is satisfiable.
  - If φ is not closed, we say $\mathcal{I} = (A, v)$ makes true φ.
  - If φ is closed, we say the structure A makes true φ.(Since it does not matter what valuation it is.)
  - If φ is closed, we say the structure A a model of φ.

Theories

- Traditionally, its a set of axioms
- Here it consists of three parts
  - Signature: Σ

  - **Axioms:** We think this part is left for the people who implement solvers. Take INT theory as an example, since we have the plus sign in our signature (we just denote it as ADD, so that you know it is only a symbol, not the actual operation), we will have an axiom like x:.y:.z:.(x,y,z)x+y=z. Therefore, our model (or structure) must contain the correct relations to map to the actual addition operation to satisfy this axiom.

    Also, some theories like REAL include those axioms as plain text, like associativity, commutativity, etc.

  - Models: A set of Σ-structures, all of which are models of the theory.

Logics

- Sublogic: it is a sublogic of SMT-LIB logic
- **Restrictions:**
  - 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

1.3.2 Lexer Rules

```
// Predefined Symbols
SYM_BOOL                : 'Bool';
SYM_CONTINUED_EXECUTION : 'continued-execution';
SYM_ERROR               : 'error';
SYM_FALSE               : 'false';
SYM_IMMEDIATE_EXIT      : 'immediate-exit';
SYM_INCOMPLETE          : 'incomplete';
SYM_LOGIC               : 'logic';
SYM_MEMOUT              : 'memout';
SYM_SAT                 : 'sat';
SYM_SUCCESS             : 'success';
SYM_THEORY              : 'theory';
SYM_TRUE                : 'true';
SYM_UNKNOWN             : 'unknown';
SYM_UNSAT               : 'unsat';
SYM_UNSUPPORTED         : 'unsupported';

// Predefined Keywords
KEYWORD_ALL_STATISTICS              : ':all-statistics';
KEYWORD_AUTHORS                     : ':authors';
KEYWORD_AXIOMS                      : ':axioms';
KEYWORD_CHAINABLE                   : ':chainable';
KEYWORD_DEFINITION                  : ':definition';
KEYWORD_DIAGNOSTIC_OUTPUT_CHANNEL   : ':diagnostic-output-channel';
KEYWORD_ERROR_BEHAVIOR              : ':error-behavior';
KEYWORD_EXPAND_DEFINITIONS          : ':expand-definitions';
KEYWORD_EXTENSIONS                  : ':extensions';
KEYWORD_FUNS                        : ':funs';
KEYWORD_FUNS_DESCRIPTION            : ':funs-description';
KEYWORD_INTERACTIVE_MODE            : ':interactive-mode';
KEYWORD_LANGUAGE                    : ':language';
KEYWORD_LEFT_ASSOC                  : ':left-assoc';
KEYWORD_NAME                        : ':name';
```

```
KEYWORD_NAMED                       : ':named';
KEYWORD_NOTES                       : ':notes';
KEYWORD_PRINT_SUCCESS               : ':print-success';
KEYWORD_PRODUCE_ASSIGNMENTS         : ':produce-assignments';
KEYWORD_PRODUCE_MODELS              : ':produce-models';
KEYWORD_PRODUCE_PROOFS              : ':produce-proofs';
KEYWORD_PRODUCE_UNSAT_CORES         : ':produce-unsat-cores';
KEYWORD_RANDOM_SEED                 : ':random-seed';
KEYWORD_REASON_UNKNOWN              : ':reason-unknown';
KEYWORD_REGULAR_OUTPUT_CHANNEL      : ':regular-output-channel';
KEYWORD_RIGHT_ASSOC                 : ':right-assoc';
KEYWORD_SORTS                       : ':sorts';
KEYWORD_SORTS_DESCRIPTION           : ':sorts-description';
KEYWORD_STATUS                      : ':status';
KEYWORD_THEORIES                    : ':theories';
KEYWORD_VALUES                      : ':values';
KEYWORD_VERBOSITY                   : ':verbosity';
KEYWORD_VERSION                     : ':version';

// Predifined General Token
TOKEN_BANG       : '!';
TOKEN_UNDERSCORE : '_';
TOKEN_AS         : 'as';
TOKEN_DECIMAL    : 'DECIMAL';
TOKEN_EXISTS     : 'exists';
TOKEN_FORALL     : 'forall';
TOKEN_LET        : 'let';
TOKEN_NUMERAL    : 'NUMERAL';
TOKEN_PAR        : 'par';
TOKEN_STRING     : 'STRING';

// Predefined Command Token
TOKEN_CMD_ASSERT          : 'assert';
TOKEN_CMD_CHECK_SAT       : 'check-sat';
TOKEN_CMD_DECLARE_SORT    : 'declare-sort';
TOKEN_CMD_DECLARE_FUN     : 'declare-fun';
TOKEN_CMD_DEFINE_SORT     : 'define-sort';
TOKEN_CMD_DEFINE_FUN      : 'define-fun';
TOKEN_CMD_EXIT            : 'exit';
TOKEN_CMD_GET_ASSERTIONS  : 'get-assertions';
TOKEN_CMD_GET_ASSIGNMENT  : 'get-assignment';
TOKEN_CMD_GET_INFO        : 'get-info';
TOKEN_CMD_GET_OPTION      : 'get-option';
TOKEN_CMD_GET_PROOF       : 'get-proof';
TOKEN_CMD_GET_UNSAT_CORE  : 'get-unsat-core';
TOKEN_CMD_GET_VALUE       : 'get-value';
TOKEN_CMD_POP             : 'pop';
TOKEN_CMD_PUSH            : 'push';
TOKEN_CMD_SET_LOGIC       : 'set-logic';
TOKEN_CMD_SET_INFO        : 'set-info';
TOKEN_CMD_SET_OPTION      : 'set-option';

fragment DIGIT      : [0-9];
fragment HEXDIGIT   : DIGIT | [a-fA-F];
fragment ALPHA      : [a-zA-Z];
fragment ESCAPE     : '\\' ('\\' | '"');
fragment SYM_CHAR   : [+-/*=%?!.$_~&^<>@];
```

```
NUMERAL       : '0' | [1-9] DIGIT*;
DECIMAL       : NUMERAL '.' [0]* NUMERAL;
HEXADECIMAL   : '#x' HEXDIGIT+;
BINARY        : '#b' [01]+;
STRING        : '"' (ESCAPE | ~('\\' | '"')*) '"';
WS            : [\t\r\n\f ]+ {skip();};
SIMPLE_SYM    : (ALPHA | SYM_CHAR) (DIGIT | ALPHA | SYM_CHAR)*;
QUOTED_SYM    : '|' ~('|' | '\\')* '|';
COMMENT       : ';' ~('\n' | '\r')* {skip();};
KEYWORD_TOKEN : ':' (ALPHA | DIGIT | SYM_CHAR)+;
```

1.3.3 Parser Rules

```
symbol      : SIMPLE_SYM
            | QUOTED_SYM
            | SYM_BOOL
            | SYM_CONTINUED_EXECUTION
            | SYM_ERROR
            | SYM_FALSE
            | SYM_IMMEDIATE_EXIT
            | SYM_INCOMPLETE
            | SYM_LOGIC
            | SYM_MEMOUT
            | SYM_SAT
            | SYM_SUCCESS
            | SYM_THEORY
            | SYM_TRUE
            | SYM_UNKNOWN
            | SYM_UNSAT
            | SYM_UNSUPPORTED
            ;

keyword     : KEYWORD_TOKEN
            | KEYWORD_ALL_STATISTICS
            | KEYWORD_AUTHORS
            | KEYWORD_AXIOMS
            | KEYWORD_CHAINABLE
            | KEYWORD_DEFINITION
            | KEYWORD_DIAGNOSTIC_OUTPUT_CHANNEL
            | KEYWORD_ERROR_BEHAVIOR
            | KEYWORD_EXPAND_DEFINITIONS
            | KEYWORD_EXTENSIONS
            | KEYWORD_FUNS
            | KEYWORD_FUNS_DESCRIPTION
            | KEYWORD_INTERACTIVE_MODE
            | KEYWORD_LANGUAGE
            | KEYWORD_LEFT_ASSOC
            | KEYWORD_NAME
            | KEYWORD_NAMED
            | KEYWORD_NOTES
            | KEYWORD_PRINT_SUCCESS
            | KEYWORD_PRODUCE_ASSIGNMENTS
            | KEYWORD_PRODUCE_MODELS
            | KEYWORD_PRODUCE_PROOFS
            | KEYWORD_PRODUCE_UNSAT_CORES
            | KEYWORD_RANDOM_SEED
```

```
            | KEYWORD_REASON_UNKNOWN
            | KEYWORD_REGULAR_OUTPUT_CHANNEL
            | KEYWORD_RIGHT_ASSOC
            | KEYWORD_SORTS
            | KEYWORD_SORTS_DESCRIPTION
            | KEYWORD_STATUS
            | KEYWORD_THEORIES
            | KEYWORD_VALUES
            | KEYWORD_VERBOSITY
            | KEYWORD_VERSION
            ;

spec_constant   : NUMERAL | DECIMAL | HEXADECIMAL | BINARY | STRING;
s_expr          : spec_constant | symbol | keyword | '(' s_expr* ')';

identifier      : symbol | '(' TOKEN_UNDERSCORE symbol NUMERAL+ ')';
sort            : identifier | '(' identifier sort+ ')';
attribute_value : symbol | spec_constant | '(' s_expr* ')';
attribute       : keyword | keyword attribute_value;

qual_identifier : identifier | '(' TOKEN_AS identifier sort ')';
var_binding     : '(' symbol term ')';
sorted_var      : '(' symbol sort ')';
term
    : spec_constant
    | qual_identifier
    | '(' qual_identifier term+ ')'
    | '(' TOKEN_LET '(' var_binding+ ')' term ')'
    | '(' TOKEN_FORALL '(' sorted_var+ ')' term ')'
    | '(' TOKEN_EXISTS '(' sorted_var+ ')' term ')'
    | '(' TOKEN_BANG term attribute+ ')'
    ;

sort_symbol_decl    : '(' identifier NUMERAL attribute* ')';
meta_spec_constant  : TOKEN_NUMERAL | TOKEN_DECIMAL | TOKEN_STRING;
fun_symbol_decl
    : '(' spec_constant sort attribute* ')'
    | '(' meta_spec_constant sort attribute* ')'
    | '(' identifier sort+ attribute* ')'
    ;
par_fun_symbol_decl
    : fun_symbol_decl
    | '(' TOKEN_PAR '(' symbol+ ')' '(' identifier sort+ attribute* ')' ')'
    ;

theory_decl : '(' SYM_THEORY symbol? theory_attribute+ ')';

theory_attribute
    : KEYWORD_SORTS '(' sort_symbol_decl+ ')'
    | KEYWORD_FUNS '(' par_fun_symbol_decl+ ')'
    | KEYWORD_SORTS_DESCRIPTION STRING
    | KEYWORD_FUNS_DESCRIPTION STRING
    | KEYWORD_DEFINITION STRING
    | KEYWORD_VALUES STRING
    | KEYWORD_NOTES STRING
    | attribute
    ;
```

```
logic_attribute
    : KEYWORD_THEORIES '(' symbol+ ')'
    | KEYWORD_LANGUAGE STRING
    | KEYWORD_EXTENSIONS STRING
    | KEYWORD_VALUES STRING
    | KEYWORD_NOTES STRING
    | attribute
    ;

logic   : '(' SYM_LOGIC symbol logic_attribute+ ')';

b_value : SYM_TRUE | SYM_FALSE;
option
    : KEYWORD_PRINT_SUCCESS b_value
    | KEYWORD_EXPAND_DEFINITIONS b_value
    | KEYWORD_INTERACTIVE_MODE b_value
    | KEYWORD_PRODUCE_PROOFS b_value
    | KEYWORD_PRODUCE_UNSAT_CORES b_value
    | KEYWORD_PRODUCE_MODELS b_value
    | KEYWORD_PRODUCE_ASSIGNMENTS b_value
    | KEYWORD_REGULAR_OUTPUT_CHANNEL STRING
    | KEYWORD_DIAGNOSTIC_OUTPUT_CHANNEL STRING
    | KEYWORD_RANDOM_SEED NUMERAL
    | KEYWORD_VERBOSITY NUMERAL
    | attribute
    ;

info_flag
    : KEYWORD_ERROR_BEHAVIOR
    | KEYWORD_NAME
    | KEYWORD_AUTHORS
    | KEYWORD_VERSION
    | KEYWORD_STATUS
    | KEYWORD_REASON_UNKNOWN
    | keyword
    | KEYWORD_ALL_STATISTICS
    ;

command
    : '(' TOKEN_CMD_SET_LOGIC symbol ')'
    | '(' TOKEN_CMD_SET_OPTION option ')'
    | '(' TOKEN_CMD_SET_INFO attribute ')'
    | '(' TOKEN_CMD_DECLARE_SORT symbol NUMERAL ')'
    | '(' TOKEN_CMD_DEFINE_SORT symbol '(' symbol* ')' sort ')'
    | '(' TOKEN_CMD_DECLARE_FUN symbol '(' sort* ')' sort ')'
    | '(' TOKEN_CMD_DEFINE_FUN symbol '(' sorted_var* ')' sort term ')'
    | '(' TOKEN_CMD_PUSH NUMERAL ')'
    | '(' TOKEN_CMD_POP NUMERAL ')'
    | '(' TOKEN_CMD_ASSERT term ')'
    | '(' TOKEN_CMD_CHECK_SAT ')'
    | '(' TOKEN_CMD_GET_ASSERTIONS ')'
    | '(' TOKEN_CMD_GET_PROOF ')'
    | '(' TOKEN_CMD_GET_UNSAT_CORE ')'
    | '(' TOKEN_CMD_GET_VALUE '(' term+ ')' ')'
    | '(' TOKEN_CMD_GET_ASSIGNMENT ')'
    | '(' TOKEN_CMD_GET_OPTION keyword ')'
    | '(' TOKEN_CMD_GET_INFO info_flag ')'
    | '(' TOKEN_CMD_EXIT ')'
```

```
    ;

script : command+;

gen_response     : SYM_UNSUPPORTED | SYM_SUCCESS | '(' SYM_ERROR STRING ')';
error_behavior   : SYM_IMMEDIATE_EXIT | SYM_CONTINUED_EXECUTION;
reason_unknown   : SYM_MEMOUT | SYM_INCOMPLETE;
status           : SYM_SAT | SYM_UNSAT | SYM_UNKNOWN;
info_response
    : KEYWORD_ERROR_BEHAVIOR error_behavior
    | KEYWORD_NAME STRING
    | KEYWORD_AUTHORS STRING
    | KEYWORD_VERSION STRING
    | KEYWORD_REASON_UNKNOWN reason_unknown
    | attribute
    ;

get_info_response       : '(' info_response+ ')';
check_sat_response      : status;
get_assertions_response : '(' term+ ')';
proof                   : s_expr;
get_proof_response      : proof;
get_unsat_core_response : '(' symbol+ ')';
valuation_pair          : '(' term term ')';
get_value_response      : '(' valuation_pair+ ')';
t_valuation_pair        : '(' symbol b_value ')';
get_assignment_response : '(' t_valuation_pair* ')';
get_option_response     : attribute_value;
```

1.3.4 Examples

1.3.5 Script File

1.4 Test

1.4.1 Benchmarks

SMT-LIB community has been contributing benchmarks and holding competitions for several years. 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 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.

1.4.2 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.

1.5 Results

You can find detailed testing data analysis and figures in our full paper `here`.

1.5.1 CVC4

CVC4 as a DPLL 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.

1.5.2 Alt-Ergo

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.

1.5.3 Comparison

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

implemented in C++ while Alt-Ergo is in OCaml.

1.5.4 Integrating CVC4 & 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.

1.6 References

We have used a lot of materials from the Internet. And these are an incomplete list of them. We want to say thank you to all those geniuses behind these documentations/papers/reports.

1.6.1 Website Links

1.6.2 Books

1.6.3 Papers/Reports

CHAPTER

TWO

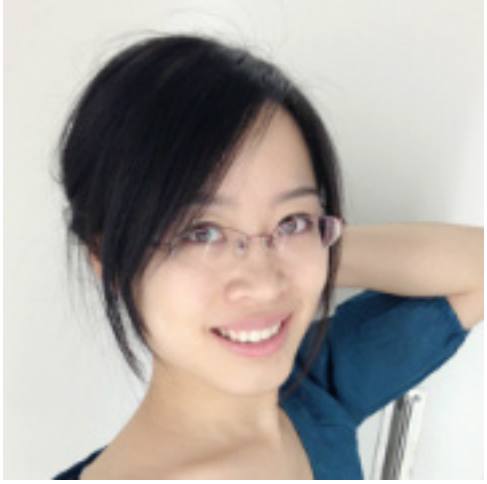
MEMBERS

Hanwen Wu

Homepage http://steinwaywu.info/

Email steinwaywhw AT gmail DOT com

Wenxin Feng



Homepage http://cs-people.bu.edu/wenxinf/

Email wenxinf AT bu DOT edu

CHAPTER

THREE

MILESTONES

This is a tracking table for the whole lifecycle. It is subject to change according to the actual situation.

Ticket	Task	Status	Assign
1	Collecting papers/reports/documentations for these projects	Finished	Hanwen/Wenxin
2	Investigating SMTLIB v2.0, CVC4, Alt-Ergo input languages	Finished	Wenxin/Hanwen
3	Investigating abstract syntax for those input language within the bit-vectors and integer theories	Finished	Hanwen/Wenxin
4	Testing Alt-Ergo using SMTLIB v2.0	Finished	Hanwen
5	Testing CVC4 using SMTLIB v2.0	Finished	Wenxin
6	Testing Data Analysis	Finished	Wenxin/Hanwen
7	Implementing CVC4 & Alt-Ergo front end	Finished	Hanwen/Wenxin

BIBLIOGRAPHY

[Alt-Ergo] http://alt-ergo.lri.fr/

[CVC4] http://cvc4.cs.nyu.edu/web/

[SMTLIB] http://smtlib.org/

[BaBE11] Barker-Plummer, David; Barwise, Jon; Etchemendy, John: Language, Proof, and Logic. 2. Aufl. : Center for the Study of Language and Inf, 2011 — ISBN 1575866323

[HuRy04] Huth, M.; Ryan, M.: Logic in Computer Science: Modelling and Reasoning About Systems. 2. Aufl. : Cambridge University Press Cambridge, UK, 2004

[AFGK11] Armand, Mickaël; Faure, Germain; Grégoire, Benjamin; Keller, Chantal; Théry, Laurent; Wener, Benjamin: Verifying SAT and SMT in Coq for a fully automated decision procedure. In: PSATTT'11: International Workshop on Proof-Search in Axiomatic Theories and Type Theories, 2011

[AvBM01] Avellone, A.; Benini, M.; Moscato, U.: How to avoid the formal verification of a theorem prover. In: Logic journal of IGPL 9 (2001), Nr. 1, S. 1–25

[BaST10] Barrett, Clark; Stump, Aaron; Tinelli, Cesare: The smt-lib standard: Version 2.0. In: Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England). Bd. 13, 2010

[BDOS08] Barrett, Clark; Deters, Morgan; Oliveras, Albert; Stump, Aaron: Design and results of the 4th annual satisfiability modulo theories competition (SMT-COMP 2008). In: To appear 6 (2008)

[BeKL12] Bestavros, Azer; Kfoury, Assaf; Lapets, Andrei: Seamless Composition and Integration: A Perspective on Formal Methods Research (2012)

[BFMP11] Bobot, Franccois; Filliâtre, Jean-Christophe; Marché, Claude; Paskevich, Andrei: The Why3 platform : LRI, CNRS & Univ. Paris-Sud & INRIA Saclay, version 0.64 edition, 2011

[BMBC08] Barrett, Clark; de Moura, Leonardo; Bjørner, Nikolaj; Cimatti, Alessandro; ITC-IRST, Trento; Dutertre, Bruno; Krstic, Sava; Nieuwenhuis, Robert; u. a.: SMT 2008: 6th International Workshop on Satisfiability Modulo Theories. In: Workshop: July. Bd. 7, 2008, S. 8

[BoFi00] Bobot, Franccois; Filliâtre, Jean-Christophe: Separation Predicates: a Taste of Separation Logic in First-Order Logic

[CoCK06] Conchon, Sylvain; Contejean, Evelyne; Kanig, Johannes: Ergo: a theorem prover for polymorphic first-order logic modulo theories. In: Artigo sobre o Ergo. Disponível em:< http://ergo. lri. fr/papers/ergo. ps>. Acesso em 17 (2006)

[EcPe09] Echenim, M.; Peltier, N.: A New Instantiation Scheme for Satisfiability Modulo Theories (Research Report) (2009)

[GuKM11] Guitton, Jérôme; Kanig, Johannes; Moy, Yannick: Why Hi-Lite Ada? In: Rustan, et al.[32] (2011), S. 27–39

[LaMi00] Lapets, Andrei; Mirzaei, Saber: Towards Lightweight Integration of SMT Solvers

[MoBj09] De Moura, L.; Bjørner, N.: Satisfiability modulo theories: An appetizer. In: Formal Methods: Foundations and Applications (2009), S. 23–36

[Mour00] de Moura, L.: SMT Solvers: Theory and Implementation

[PrBG05] Prasad, M. R.; Biere, A.; Gupta, A.: A survey of recent advances in SAT-based formal verification. In: International Journal on Software Tools for Technology Transfer (STTT) 7 (2005), Nr. 2, S. 156–173

[RuSh07] Rushby, John; Shankar, Natarajan: AFM'07: Second Workshop on Automated Formal Methods: November 6, 2007, Atlanta, Georgia : Association for Computing Machinery, 2007