A

SMT-LIB: a Brief Tutorial

A.1 The Satisfiability-Modulo-Theory Library and Standard (SMT-LIB)

A bit of history: The growing interest and need for decision procedures such as those described in this book led to the **SMT-LIB initiative** (short for Satisfiability-Modulo-Theory Library). The main purpose of this initiative was to streamline the research and tool development in the field to which this book is dedicated. For this purpose, the organizers developed the **SMT-LIB standard** [239], which formally specifies the theories that attract enough interest in the research community, and that have a sufficiently large set of publicly available benchmarks. As a second step, the organizers started collecting benchmarks in this format, and today (2016) the SMT-LIB repository includes more than 100 000 benchmarks in the SMT-LIB 2.5 format, classified into dozens of logics. A third step was to initiate **SMT-COMP**, an annual competition for SMT solvers, with a separate track for each division.

These three steps have promoted the field dramatically: only a few years back, it was very hard to get benchmarks, every tool had its own language standard and hence the benchmarks could not be migrated without translation, and there was no good way to compare tools and methods.[1] These problems have mostly been solved because of the above initiative, and, consequently, the number of tools and research papers dedicated to this field is now steadily growing.

The SMT-LIB initiative was born at FroCoS 2002, the fourth Workshop on Frontiers of Combining Systems, after a proposal by Alessandro Armando. At the time of writing this appendix, it is co-led by Clark Barrett, Pascal Fontaine, and Cesare Tinelli. Clark Barrett, Leonardo de Moura, and Cesare Tinelli currently manage the SMT-LIB benchmark repository.

[1] In fact, it was reported in [94] that each tool tended to be the best on its own set of benchmarks.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

The current state: The current SMT-LIB standard is at version 2.5 (as of 2015). It supports the theories that are presented in Fig. A.1. The symbols should be interpreted as follows:

- QF for the restriction to quantifier-free formulas
- A or AX for arrays without or with extensionality
- BV for fixed-size bit-vectors
- FP for Floating-Point
- IA for integer arithmetic
- RA for real arithmetic
- IRA for mixed integer arithmetic
- IDL for integer difference logic
- RDL for rational difference logic
- L before IA, RA, or IRA for the linear fragment of those arithmetics
- N before IA, RA, or IRA for the nonlinear fragment of those arithmetics
- UF for the extension allowing free sort and function symbols

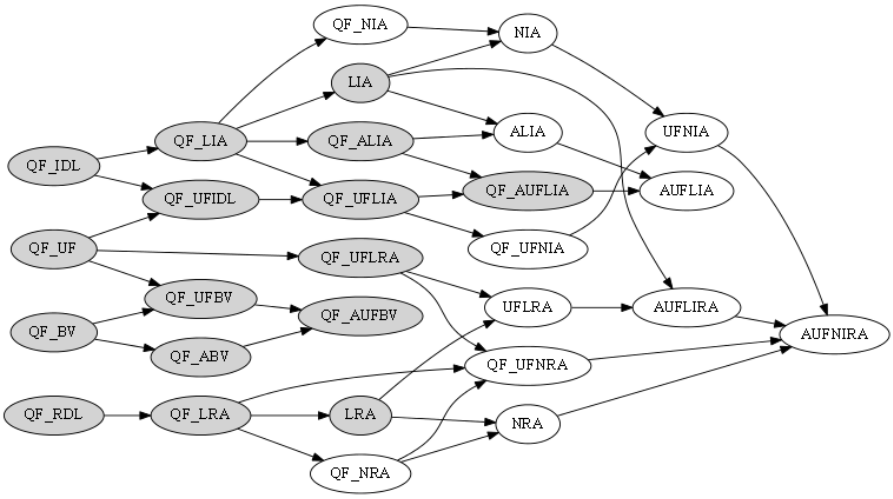

Fig. A.1. The theories supported by the SMT-LIB standard have associated benchmarks and at least one tool that (attempts to) solve them. An arrow (T_1, T_2) means that T_1 is a special case of T_2. The greyed nodes are theories that are covered in this book. The figure is copied (with permission) from the SMT-LIB web-site

A.2 The SMT-LIB File Interface

The SMT-LIB standard defines a file format for describing decision problems. The benefit of a standardized file format is that it is easy to experiment with

a range of solvers, and to replace the solver used in case better solvers are developed. The description below refers to ver. 2.0 of the standard, but ver. 2.5 is backward-compatible.

SMT-LIB files are ASCII text files, and as a consequence can be written with any text editor that can save plain text files. The syntax is derived from that of Common Lisp's S-expressions. All popular solvers are able to read formulas from files or the standard input of the program, which permits the use of POSIX pipes to communicate with the solver. We will refrain from giving a formal syntax and semantics for SMT-LIB files, and will instead give examples for the most important theories.

A.2.1 Propositional Logic

We will begin with an example in propositional logic. Suppose we wanted to check the satisfiability of

$$(a \vee b) \wedge \neg a \ .$$

We first need to *declare* the Boolean variables a and b. The SMT-LIB syntax offers the command `declare-fun` for declaring functions, i.e., mappings from some sequence of function arguments to the domain of the function. Variables are obtained by creating a function without arguments. Thus, we will write

```
1 (declare-fun a () Bool)
2 (declare-fun b () Bool)
```

to obtain two Boolean variables named a and b. Note the empty sequence of arguments after the name of the variable.

We can now write constraints over these variables. The syntax for the usual Boolean constants and connectives is as follows:

TRUE	`true`
FALSE	`false`
$\neg a$	`(not a)`
$a \implies b$	`(=> a b)`
$a \wedge b$	`(and a b)`
$a \vee b$	`(or a b)`
$a \oplus b$	`(xor a b)`

Using the operators in the table, we can write the formula above as follows:

```
1 (and (or a b) (not a))
```

Constraints are given to the SMT solver using the command `assert`. We can add the formula above as a constraint by writing

```
1 (assert (and (or a b) (not a)))
```

As our formula is a conjunction of two constraints, we could have equivalently written

```
(assert (or a b))
(assert (not a))
```

After we have passed all constraints to the solver, we can check satisfiability of the constraint system by issuing the following command:

```
(check-sat)
```

The solver will reply to this command with `unsat` or `sat`, respectively. In the case of the formula above, we will get the answer `sat`. To inspect the satisfying assignment, we issue the `get-value` command.

```
(get-value (a b))
```

This command takes a list of variables as argument. This makes it possible to query the satisfying assignment for any subset of the variables that have been declared.

A.2.2 Arithmetic

The SMT-LIB format standardizes syntax for arithmetic over integers and over reals. The type of the variable is also called the *sort*. The SMT-LIB syntax has a few constructs that can be used for all sorts. For instance, we can write `(= x y)` to denote equality of x and y, provided that x and y have the same sort. Similarly, we can write `(disequal x y)` to say that x and y are different. The operator `disequal` can be applied to more than two operands, e.g., as in `(disequal a b c)`. This is equivalent to saying that all the arguments are different.

The SMT-LIB syntax furthermore offers a trinary if-then-else operator, which is denoted as `(ite c x y)`. The first operand must be a Boolean expression, whereas the second and third operands may have any sort as long as the sort of x matches that of y. The expression evaluates to x if c evaluates to TRUE, and to y otherwise.

To write arithmetic expressions, SMT-LIB offers predefined sorts called `Real` and `Int`. The obvious function symbols are defined, as given in the table below.

addition	+
subtraction	−
unary minus	−
multiplication	*
division	/ (reals) div (integers)
remainder	mod (integers only)
relations	< > <= >=

Many of the operators can be *chained*, with the obvious meaning, as in, for example, `(+ x y z)`. The solver will check that the variables in an expression have the same sort. The nonnegative integer and decimal constant symbols 1,

2, 3.14, and so on are written in the obvious way. Thus, the expression $2x + y$ is written as `(+ (* 2 x) y)`. To obtain a negative number, one uses the unary minus operator, as in `(- 1)`. By contrast, `-1` is not accepted.

A.2.3 Bit-Vector Arithmetic

The SMT-LIB syntax offers a *parametric* sort `BitVec`, where the parameter indicates the number of bits in the bit vector. The underscore symbol is used to indicate that `BitVec` is parametric. As an example, we can define an 8-bit bit vector a and a 16-bit bit vector b as follows:

```
(declare-fun a () (_ BitVec 8))
(declare-fun b () (_ BitVec 16))
```

Constants can be given in hexadecimal or binary notation, e.g., as follows:

```
(assert (= a #b11110000))
(assert (= b #xff00))
```

The operators are given in the table below. Recall from Chap. 6 that the semantics of some of the arithmetic operators depend on whether the bit vector is interpreted as an unsigned integer or as two's complement. In particular, the semantics differs for the division and remainder operators, and the relational operators.

	Unsigned	Two's complement
addition	`bvadd`	
subtraction	`bvsub`	
multiplication	`bvmul`	
division	`bvudiv`	`bvsdiv`
remainder	`bvurem`	`bvsrem`
relations	`bvult`, `bvugt`, `bvule`, `bvuge`	`bvslt`, `bvsgt`, `bvsle`, `bvsge`
left shift	`bvshl`	
right shift	`bvlshr`	`bvashr`

Bit vector concatenation is done with `concat`. A subrange of the bits of a bit vector can be extracted with `(_ extract` i j`)`, which extracts the bits from index j to index i (inclusive).

A.2.4 Arrays

Recall from Chap. 7 that arrays map an index type to an element type. As an example, we would write

```
(declare-fun a () (Array Int Real))
```

in SMT-LIB syntax to obtain an array a that maps integers to reals. The SMT-LIB syntax for $a[i]$ is `(select a i)`, and the syntax for the array update operator $a\{i \leftarrow e\}$ is `(store a i e)`.

A.2.5 Equalities and Uninterpreted Functions

Equality logic can express equalities and disequalities over variables taken from some unspecified set. The only assumption is that this set has an infinite number of elements. To define the variables, we first need to declare the set itself, i.e., in SMT-LIB terminology, we declare a new sort. The command is `declare-sort`. We obtain a new sort `my_sort` and variables a, b, and c of that sort as follows:

```
(declare-sort my_sort 0)
(declare-fun a () my_sort)
(declare-fun b () my_sort)
(declare-fun c () my_sort)
(assert (= a b))
(assert (disequal a b c))
```

The number zero in the `declare-sort` command is the arity of the sort. The arity can be used for subtyping, e.g., the arrays from above have arity two.

B

A C++ Library for Developing Decision Procedures

B.1 Introduction

A decision procedure is always more than one algorithm. A lot of infrastructure is required to implement even simple decision procedures. We provide a large part of this infrastructure in the form of the DPLIB library in order to simplify the development of new procedures. DPLIB is available for download,[1] and consists of the following parts:

- A template class for a basic data structure for graphs, described in Sect. B.2.
- A parser for a simple fragment of first-order logic given in Sect. B.3.
- Code for generating propositional SAT instances in CNF format, shown in Sect. B.4.
- A template for a decision procedure that performs a lazy encoding, described in Sect. B.5.

To begin with, the decision problem (the formula) has to be read as input by the procedure. The way this is done depends on how the decision procedure interfaces with the program that generates the decision problem.

In industrial practice, many decision procedures are embedded into larger programs in the form of a subprocedure. Programs that use a decision procedure are called *applications*. If the run time of the decision procedure dominates the total run time of the application, solvers for decision problems are often interfaced to by means of a *file interface*. This chapter provides the basic ingredients for building a decision procedure that uses a file interface. We focus on the C/C++ programming language, as all of the best-performing decision procedures are written in this language.

The components of a decision procedure with a file interface are shown in Fig. B.1. The first step is to *parse* the input file. This means that a sequence of characters is transformed into a *parse tree*. The parse tree is subsequently

[1] `http://www.decision-procedures.org/`

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

checked for type errors (e.g., adding a Boolean to a real number can be considered a type error). This step is called *type checking*. The module of the program that performs the parsing and type-checking phases is usually called the *front end*.

Most of the decision procedures described in this book permit an arbitrary Boolean structure in the formula, and thus have to reason about propositional logic. The best method to do so is to use a modern SAT solver. We explain how to interface to SAT solvers in Sect. B.4. A simple template for a decision procedure that implements an incremental translation to propositional logic, as described in Chap. 3, is given in Sect. B.5.

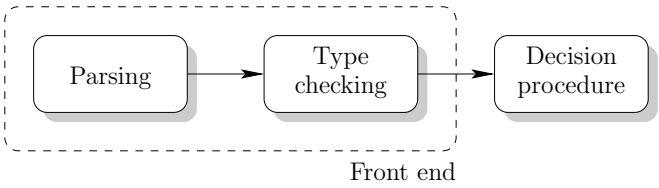

Fig. B.1. Components of a decision procedure that implements a file interface

B.2 Graphs and Trees

Graphs are a basic data structure used by many decision procedures, and can serve as a generalization of many more data structures. As an example, trees and directed acyclic graphs are obvious special cases of graphs. We have provided a template class that implements a generic graph container.

This class has the following design goals:

- It provides a *numbering* of the nodes. Accessing a node by its number is an $O(1)$ operation. The node numbers are *stable*, i.e., stay the same even if the graph is changed or copied.
- The data structure is optimized for *sparse* graphs, i.e., with few edges. Inserting or removing edges is an $O(\log k)$ operation, where k is the number of edges. Similarly, determining if a particular edge exists is also $O(\log k)$.
- The nodes are stored densely in a `vector`, i.e., with very little overhead per node. This permits a large number (millions) of nodes. However, adding or removing nodes may invalidate references to already existing nodes.

An instance of a graph named `G` is created as follows:

```
#include "graph.h"
...
  graph<graph_nodet<> > G;
```

Initially, the graph is empty. Nodes can be added in two ways: a single node is added using the method `add_node()`. This method adds one node, and returns the number of this node. If a larger number of nodes is to be added, the method `resize(`i`)` can be used. This changes the number of nodes to i by either adding or removing an appropriate number of nodes. Means to erase individual nodes are not provided.

The class `graph` can be used for both directed and undirected graphs. Undirected graphs are simply stored as directed graphs where edges always exist in both directions. We write $a \longrightarrow b$ for a directed edge from a to b, and $a \longleftrightarrow b$ for an undirected edge between a and b.

Class:	`graph<T>`	
Methods:	`add_edge(`a`, `b`)`	adds $a \longrightarrow b$
	`remove_edge(`a`, `b`)`	removes $a \longrightarrow b$, if it exists
	`add_undirected_edge(`a`, `b`)`	adds $a \longleftrightarrow b$
	`remove_undirected_edge(`a`, `b`)`	removes $a \longleftrightarrow b$
	`remove_in_edges(`a`)`	removes $x \longrightarrow a$, for any node x
	`remove_out_edges(`a`)`	removes $a \longrightarrow x$, for any node x
	`remove_edges(`a`)`	removes $a \longrightarrow x$ and $x \longrightarrow a$, for any node x

Table B.1. Interface of the template class `graph<T>`

The methods of this template class are shown in Table B.1. The method `has_edge(`a`, `b`)` returns `true` if and only if $a \longrightarrow b$ is in the graph. The set of nodes x such that $x \longrightarrow a$ is returned by `in(`a`)`, and the set of nodes x such that $a \longrightarrow x$ is returned by `out(`a`)`.

The class `graph` provides an implementation of the following two algorithms:

- The set of nodes that are reachable from a given node a can be computed using the method `visit_reachable(`a`)`. This method sets the member `.visited` of all nodes that are reachable from node a to `true`. This member can be set for all nodes to `false` by calling the method `clear_visited()`.
- The shortest path from a given node a to a node b can be computed with the method `shortest_path(`a`, `b`, `p`)`, which takes an object p of type `graph::patht` (a list of node numbers) as its third argument, and stores the shortest path between a and b in there. If b is not reachable from a, then p is empty.

B.2.1 Adding "Payload"

Many algorithms that operate on graphs may need to store additional information per node or per edge. The container class provides a convenient way to do so by defining a new class for this data, and using this new class as a template argument for the template `graph`. As an example, this can be used to define a graph that has an additional string member in each node:

```
#include "graph.h"

class my_nodet {
public:
  std::string name;
};
...

  graph<my_nodet> G;
```

Data members can be added to the edges by passing a class type as a second template argument to the template `graph_nodet`. As an example, the following fragment allows a `weight` to be associated with each edge:

```
#include "graph.h"

class my_edget {
  int weight;

  my_edget():weight(0) {
  }
};

class my_nodet {
};
...

  graph<my_nodet, my_edget> G;
```

Individual edges can be accessed using the method `edge()`. The following example sets the weight of edge $a \longrightarrow b$ to 10:

```
  G.edge(a, b).weight=10;
```

B.3 Parsing

B.3.1 A Grammar for First-Order Logic

Many decision problems are stored in a file. The decision procedure is then passed the name of the file. The first step of the program that implements

$$
\begin{array}{ll}
\textit{id} & : \texttt{[a-zA-Z\_\$][a-zA-Z0-9\_\$]}^{+} \\
\mathbb{N}\textit{-elem} & : \texttt{[0-9]}^{+} \\
\mathbb{Q}\textit{-elem} & : \texttt{[0-9]}^{*}\texttt{.[0-9]}^{+} \\
\textit{infix-function-id} & : + \mid - \mid * \mid / \mid \textit{mod} \\
\textit{boolop-id} & : \wedge \mid \vee \mid \Leftrightarrow \mid \Longrightarrow \\
\textit{infix-relop-id} & : < \mid > \mid \leq \mid \geq \mid = \\
\textit{quantifier} & : \forall \mid \exists \\
\textit{term} & : \textit{id} \\
 & \mid \mathbb{N}\textit{-elem} \mid \mathbb{Q}\textit{-elem} \\
 & \mid \textit{id}\ \texttt{(}\ \textit{term-list}\ \texttt{)} \\
 & \mid \textit{term infix-function-id term} \\
 & \mid - \textit{term} \\
 & \mid \texttt{(}\ \textit{term}\ \texttt{)} \\
\textit{formula} & : \textit{id} \\
 & \mid \textit{id}\ \texttt{(}\ \textit{term-list}\ \texttt{)} \\
 & \mid \textit{term infix-relop-id term} \\
 & \mid \textit{quantifier variable-list}\ \texttt{:}\ \textit{formula} \\
 & \mid \texttt{(}\ \textit{formula}\ \texttt{)} \\
 & \mid \textit{formula boolop-id formula} \\
 & \mid \neg\ \textit{formula} \\
 & \mid \texttt{true} \mid \texttt{false}
\end{array}
$$

Fig. B.2. Simple BNF grammar for formulas

the decision procedure is therefore to parse the file. The file is assumed to follow a particular syntax. We have provided a parser for a simple fragment of first-order logic with quantifiers.

Figure B.2 shows a grammar of this fragment of first-order logic. The grammar in Fig. B.2 uses mathematical notation. The corresponding ASCII representations are listed in Table B.2.

All predicates, variables, and functions have *identifiers*. These identifiers must be declared before they are used. Declarations of variables come with a *type*. These types allow a problem that is in, for example, linear arithmetic over the integers to be distinguished from a problem in linear arithmetic over the reals. Figure B.3 lists the types that are predefined. The *domain* $\mathbb{U}$ is used for types that do not fit into the other categories.

$$
\begin{array}{ll}
\mathbb{B} & \texttt{boolean} \\
\mathbb{N}_0 & \texttt{natural} \\
\mathbb{Z} & \texttt{int} \\
\mathbb{R} & \texttt{real} \\
\mathbb{B}^N & \texttt{unsigned [N]} \\
\mathbb{B}^N & \texttt{signed [N]} \\
\mathbb{U} & \texttt{untyped}
\end{array}
$$

Fig. B.3. Supported types and their ASCII representations

Mathematical symbol	Operation	ASCII
$\neg$	Negation	`not`, `!`
$\wedge$	Conjunction	`and`, `&`
$\vee$	Disjunction	`or`, `|`
$\Leftrightarrow$	Biimplication	`<=>`
$\Longrightarrow$	Implication	`=>`
$<$	Less than	`<`
$>$	Greater than	`>`
$\leq$	Less than or equal to	`<=`
$\geq$	Greater than or equal to	`>=`
$=$	Equality	`=`
$\forall$	Universal quantification	`forall`
$\exists$	Existential quantification	`exists`
$-$	Unary minus	`-`
$\cdot$	Multiplication	`*`
$/$	Division	`/`
mod	Modulo (remainder)	`mod`
$+$	Addition	`+`
$-$	Subtraction	`-`

Table B.2. Built-in function symbols

Table B.2 also defines the precedence of the built-in operators: the operators with higher precedence are listed first, and the precedence levels are separated by horizontal lines. All operators are left-associative.

B.3.2 The Problem File Format

The input files for the parser consist of a sequence of *declarations* (Fig. B.4 shows an example). All variables, functions, and predicates are declared. The declarations are separated by semicolons, and the elements in each declaration are separated by commas. Each variable declaration is followed by a type (as listed in Fig. B.3), which specifies the type of all variables in that declaration.

A declaration may also define a formula. Formulas are *named* and *tagged*. Each entry starts with the name of the formula, followed by a colon and one of the keywords `theorem`, `axiom`, or `formula`. The keyword is followed by a formula. Note that the formulas are not necessarily *closed*: the formula `simplex` contains the unquantified variables `i` and `j`. Variables that are not quantified explicitly are implicitly quantified with a universal quantifier.

```
a, b, x, p, n: int;
el: natural;
pi: real;
i, j: real;
u: untyped;        -- an untyped variable
abs: function;
prime, divides: predicate;

absolute: axiom    forall a: ((a >=0 ==> abs(a) = a) and
                    (a < 0 ==> abs(a) = -a)) ==>
                    (exists el: el = abs(a));
divides: axiom     (forall a, b: divides (a, b) <=>
                    exists x: b = a * x);
simplex: formula   (i + 5*j <= 3) and
                   (3*i < 3.7) and
                   (i > -1) and (j > 0.12)
```

Fig. B.4. A realistic example

B.3.3 A Class for Storing Identifiers

Decision problems often contain a large set of variables, which are represented by identifier strings. The main operation on these identifiers is comparison. We therefore provide a specialized string class that features string comparison in time $O(1)$. This is implemented by storing all identifiers inside a hash table. Comparing strings then reduces to comparing indices for that table.

Identifiers are stored in objects of type `dstring`. This class offers most of the methods that the other string container classes feature, with the exception of any method that modifies the string. Instances of type `dstring` can be copied, compared, ordered, and destroyed in time $O(1)$, and use as much space as an integer variable.

B.3.4 The Parse Tree

The parse tree is stored in a graph class `ast::astt` and is generated from a file as follows (Fig. B.5):

1. Create an instance of the class `ast::astt`.
2. Call the method `parse(file)` with the name of the file as an argument. The method returns `true` if an error was encountered during parsing.

The class `ast::astt` is a specialized form of a graph, and stores nodes of type `ast::nodet`. The root node is returned by the method `root()` of the class `ast::astt`. Each node stores the following information:

```
#include "parsing/ast.h"

...

  ast::astt ast;

  if(ast.parse(argv[1])) {
    std::cerr << "parsing␣failed" << std::endl;
    exit(1);
  }
```

Fig. B.5. Generating a parse tree

1. Each node has a numeric label (an integer). This is used to distinguish the operators and the terminal symbols. Table B.3 contains a list of the symbolic constants that are used for the numeric labels.
2. Nodes that contain identifiers or a numeric constant also have a string label, which is of type `dstring` (see Sect. B.3.3). We use strings for the numeric constants instead of the numeric types offered by C++ in order to support unbounded numbers.
3. Each node may have up to two child nodes.

As described in Sect. B.2, the nodes of the graph are numbered. In fact, the `ast::nodet` class is only a wrapper around these numbers, and thus can be copied efficiently. The methods it offers are shown in Table B.4. The methods `c1()` and `c2()` return NIL if there is no first or second child node, respectively.

For convenience, the `ast::astt` class provides a *symbol table*, which is a mapping from the set of identifiers to their types. Given an identifier s, the method `get_type_node(`s`)` returns the node in the parse tree that corresponds to the type of s.

B.4 CNF and SAT

B.4.1 Generating CNF

The library provides algorithms for converting propositional logic into CNF using Tseitin's method (see Sect. 1.3). The resulting clauses can be passed directly to a propositional SAT solver. Alternatively, they can be written to disk in the DIMACS format. The interface to both back ends is defined in the `propt` base class. This class is used wherever the specific propositional back end is to be left unspecified. Literals (i.e., variables or their negations) are

Name	Used for
N_IDENTIFIER	Identifier
N_INTEGER	Integer constant
N_RATIONAL	Rational constant
N_INT	Integer type
N_REAL	Real type
N_BOOLEAN	Boolean type
N_UNSIGNED	Unsigned type
N_SIGNED	Signed type
N_AXIOM	Axiom
N_DECLARATION	Declaration
N_THEOREM	Theorem
N_CONJUNCTION	$\wedge$
N_DISJUNCTION	$\vee$
N_NEGATION	$\neg$
N_BIIMPLICATION	$\Longleftrightarrow$
N_IMPLICATION	$\Longrightarrow$
N_TRUE	True
N_FALSE	False
N_ADDITION	$+$
N_SUBTRACTION	$-$
N_MULTIPLICATION	$*$
N_DIVISION	$/$
N_MODULO	mod
N_UMINUS	Unary minus
N_LOWER	$<$
N_GREATER	$>$
N_LOWEREQUAL	$\leq$
N_GREATEREQUAL	$\geq$
N_EQUAL	$=$
N_FORALL	$\forall$
N_EXISTS	$\exists$
N_LIST	A list of nodes
N_PREDICATE	Predicate
N_FUNCTION	Function

Table B.3. Numeric labels of nodes and their meanings

Class:	`ast::nodet`	
Methods:	`id()`	Returns the numeric label
	`string()`	Returns the string label
	`c1()`	Returns the first child node
	`c2()`	Returns the second child node
	`number()`	Returns the number of the node
	`is_nil()`	Returns `true` if the node is NIL

Table B.4. Interface of the class `ast::nodet`

stored in objects of type `literalt`. The constants TRUE and FALSE are returned by `const_literal(true)` and `const_literal(false)`, respectively.

Class:	`propt`	
Methods:	`land(`a`, `b`)`	Returns a literal l with $l \iff a \wedge b$
	`land(`v`)`	Given a vector $v = \langle v_1, \ldots, v_n \rangle$, returns a literal l with $l \iff \bigwedge_i v_i$
	`lor(`a`, `b`)`	Returns a literal l with $l \iff a \vee b$
	`lor(`v`)`	Given a vector $v = \langle v_1, \ldots, v_n \rangle$, returns a literal l with $l \iff \bigvee_i v_i$
	`lxor(`a`, `b`)`	Returns a literal l with $l \iff a \oplus b$
	`lnot(`a`, `b`)`	Returns a literal l with $l \iff \neg a$
	`lnand(`a`, `b`)`	Returns a literal l with $l \iff \neg(a \wedge b)$
	`lnor(`a`, `b`)`	Returns a literal l with $l \iff \neg(a \vee b)$
	`lequal(`a`, `b`)`	Returns a literal l with $l \iff (a \iff b)$
	`limplies(`a`, `b`)`	Returns a literal l with $l \iff (a \implies b)$
	`lselect(`a`, `b`, `c`)`	Returns a literal l with $(a \implies (l \iff b)) \wedge (\neg a \implies (l \iff c))$
	`set_equal(`a`, `b`)`	Adds the constraint $a \iff b$
	`new_variable()`	Returns a new variable
	`const_literal(`c`)`	Returns a literal with a constant Boolean truth value given by c

Table B.5. Interface of the class `propt`

The interface of the class `propt` is specified in Table B.5. The classes `satcheckt` and `dimacs_cnft` are derived from this class. An implementation of a state-of-the-art propositional SAT solver is given by the class `satcheckt`. The additional methods it provides are shown in Table B.6. The class `dimacs_cnft` is used to store the clauses and dump them into a text file that uses the DIMACS CNF format. Its interface is given in Table B.7.

Class:	`satcheckt`, derived from `propt`	
Methods:	`prop_solve()`	Returns `P_SATISFIABLE` if the formula is SAT
	`l_get(`l`)`	Returns the value of l in the satisfying assignment
	`solver_text()`	Returns a string that identifies the solver

Table B.6. Interface of the class `satcheckt`

Class:	`dimacs_cnft`, derived from `propt`	
Methods:	`write_dimacs_cnf(s)`	Dumps the formula in DIMACS CNF format into the stream s

Table B.7. Interface of the class `dimacs_cnft`

B.4.2 Converting the Propositional Skeleton

The **propositional skeleton** (see Chap. 3) of a parse tree can be generated using the class `skeletont`. This offers an operator `()`, which can be applied as follows, where `root_node` is the root node of a formula, and `prop` is an instance of `propt`:

```
#include "sat/skeleton.h"

...

  skeletont skeleton;

  skeleton(root_node, prop);
```

Besides converting the propositional part, the method also generates a vector `skeleton.nodes`, where each element corresponds to a node in the parse tree. Each node has two attributes:

- The attribute `type` is one of `PROPOSITIONAL` or `THEORY`, and distinguishes the skeleton from the theory atoms.
- In the case of a skeleton node, the attribute `l` is the literal that encodes the node.

B.5 A Template for a Lazy Decision Procedure

The library provides two templates for decision procedures that compute a propositional encoding of a given formula φ in the lazy manner. These algo-

rithms are described in detail under the names LAZY-DPLL (Algorithm 3.3.2) and DPLL(T) (Algorithm 3.4.1) in Chap. 3.

We first define a common interface for any kind of decision procedure. This interface is defined by a class `decision_proceduret` (Table B.8). This class offers a method `is_satisfiable(`φ`)`, which returns TRUE if and only if the formula φ is satisfiable. If so, one may call the methods `print_assignment(`s`)` and `get(`n`)`. The method `print_assignment(`s`)` dumps the entire satisfying assignment into a stream, whereas `get(`n`)` permits querying the value of an individual node n of φ.

Class:	`decision_proceduret`	
Methods:	`is_satisfiable(`φ`)`	Returns TRUE if the formula φ is found to be SAT
	`print_assignment(`s`)`	Dumps the satisfying assignment into the stream s
	`get(`n`)`	Returns the value assigned to node n of φ

Table B.8. Interface of the class `decision_proceduret`

Class:	`lazy_dpllt`, derived from `decision_proceduret`	
Methods:	`assignment(`n`, `v`)`	This method is called by the SAT solver for every assignment to a Σ-literal in φ. The node it corresponds to is n; the value assigned is given by v.
	`deduce()`	This method is called once a satisfying assignment to the current propositional encoding is found.
	`add_clause(`c`)`	Called by `deduce()` to add a clause as a consequence of a T-inconsistent assignment
Members:	`f`	A copy of φ
	`skeleton`	An instance of `skeletont`

Table B.9. Interface of the classes `lazy_dpllt` and `dpll_tt`, which are implementations of LAZY-DPLL (Algorithm 3.3.2) and DPLL(T) (Algorithm 3.4.1). The theory T is assumed to be defined over a signature Σ

The templates that we have provided implement two of the algorithms given in Chap. 3: LAZY-DPLL and DPLL(T). These templates include the conversion of the propositional skeleton of φ into CNF, and the interface to

the SAT solver. We provide a common interface to both algorithms, which is given in Table B.9.

Class:	`dpll_tt`, derived from `decision_proceduret`	
Methods:	`deduce()`	This method is called by the SAT solver to check a partial assignment for T-consistency.
	`add_clause(`c`)`	Called to add a clause as consequence of `assignment`
	`theory_implication(`n`,` v`)`	Called to communicate a T-implication to the SAT solver: n is the node implied, and v is the value.
Members:	`f`	A copy of φ
	`skeleton`	An instance of `skeletont`

Table B.10. Interface of the class `dpll_tt`, an implementation of DPLL(T) (Algorithm 3.4.1)

The only part that is left open is the interface to the decision procedure for the conjunction of Σ-literals. In the case of both algorithms, this is the method `deduce()`. The assignment to the Σ-literals is passed from the SAT solver to the deductive engine by means of calls to the method `assignment(`n`,` v`)`, where n is the node and v is the value that is assigned.

The method `deduce()` inspects this assignment to the Σ-literals. If it is found to be consistent, `deduce()` is expected to return TRUE. Otherwise, it is expected to add appropriate constraints using the method `add_clause`, and to return FALSE.

In the case of LAZY-DPLL, `deduce()` is called only for full assignments, whereas DPLL(T) may call `deduce()` for partial assignments.

References

1. W. Ackermann. *Solvable cases of the Decision Problem.* Studies in Logic and the Foundations of Mathematics. North-Holland, Amsterdam, 1954.
2. V. S. Adve and J. M. Mellor-Crummey. Using integer sets for data-parallel program analysis and optimization. In J. W. Davidson, K. D. Cooper, and A. M. Berman, editors, *PLDI*, pages 186–198. ACM, 1998.
3. J. Alglave, D. Kroening, and M. Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In N. Sharygina and H. Veith, editors, *Computer Aided Verification (CAV)*, volume 8044 of *LNCS*, pages 141–157. Springer, 2013.
4. A. Armando, C. Castellini, and E. Giunchiglia. SAT-based procedures for temporal reasoning. In *5th European Conference on Planning (ECP)*, volume 1809 of *LNCS*, pages 97–108. Springer, 1999.
5. A. Armando and E. Giunchiglia. Embedding complex decision procedures inside an interactive theorem prover. *Annals of Mathematics and Artificial Intelligence*, 8(3–4):475–502, 1993.
6. A. Armando, S. Ranise, and M. Rusinowitch. A rewriting approach to satisfiability procedures. *Inf. Comput.*, 183(2):140–164, 2003.
7. G. Audemard, P. Bertoli, A. Cimatti, A. Kornilowicz, and R. Sebastiani. A SAT based approach for solving formulas over Boolean and linear mathematical propositions. In *18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 195–210. Springer, 2002.
8. G. Audemard and L. Simon. Predicting learnt clauses quality in modern SAT solvers. In C. Boutilier, editor, *21st International Joint Conference on Artificial Intelligence (IJCAI)*, pages 399–404, 2009.
9. A. Ayari and D. A. Basin. QUBOS: deciding quantified boolean logic using propositional satisfiability solvers. In M. Aagaard and J. W. O'Leary, editors, *Formal Methods in Computer-Aided Design, 4th International Conference (FMCAD)*, volume 2517 of *LNCS*, pages 187–201. Springer, 2002.
10. L. Bachmair and A. Tiwari. Abstract congruence closure and specializations. In D. A. McAllester, editor, *17th International Conference on Automated Deduction (CADE)*, volume 1831 of *LNCS*, pages 64–78. Springer, 2000.
11. E. Balas, S. Ceria, G. Cornuejols, and N. Natraj. Gomory cuts revisited. *Operations Research Letters*, 19:1–9, 1996.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

12. T. Ball, A. Podelski, and S. K. Rajamani. Boolean and Cartesian abstraction for model checking C programs. In T. Margaria and W. Yi, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2031 of *LNCS*, pages 268–283. Springer, 2001.
13. T. Ball and S. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN 2001 Workshop on Model Checking of Software*, volume 2057 of *LNCS*, 2001.
14. C. Barrett and S. Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *International Conference on Computer-Aided Verification (CAV)*, volume 3114 of *LNCS*. Springer, 2004.
15. C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanovic, T. King, A. Reynolds, and C. Tinelli. CVC4. In *23rd International Conference on Computer Aided Verification (CAV)*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
16. C. Barrett, D. Dill, and J. Levitt. Validity checking for combinations of theories with equality. In M. K. Srivas and A. J. Camilleri, editors, *Formal Methods in Computer-Aided Design, First International Conference (FMCAD)*, volume 1166 of *LNCS*, pages 187–201. Springer, 1996.
17. C. Barrett and C. Tinelli. CVC3. In *19th International Conference on Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.
18. C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for bit-vector arithmetic. In *Design Automation Conference (DAC)*, pages 522–527. ACM Press, 1998.
19. C. W. Barrett, D. L. Dill, and A. Stump. Checking satisfiability of first-order formulas by incremental translation to SAT. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 236–249. Springer, 2002.
20. C. W. Barrett, D. L. Dill, and A. Stump. A generalization of Shostak's method for combining decision procedures. In A. Armando, editor, *Frontiers of Combining Systems, 4th International Workshop (FroCos)*, volume 2309 of *LNCS*, pages 132–146. Springer, 2002.
21. R. Bayardo and R. Schrag. Using CSP look-back techniques to solve real-world SAT instances. In B. Kuipers and B. L. Webber, editors, *Fourteenth National Conference on Artificial Intelligence (AAAI)*, pages 203–208. AAAI Press, 1997.
22. P. Beame, H. Kautz, and A. Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, 2004.
23. E. Ben-Sasson and A. Wigderson. Short proofs are narrow – resolution made simple. *J. ACM*, 48(2):149–169, 2001.
24. M. Benedikt, T. W. Reps, and S. Sagiv. A decidable logic for describing linked data structures. In *Programming Languages and Systems, 8th European Symposium on Programming (ESOP)*, volume 1576 of *LNCS*, pages 2–19. Springer, 1999.
25. J. Berdine, C. Calcagno, and P. W. O'Hearn. A decidable fragment of separation logic. In *Foundations of Software Technology and Theoretical Computer Science, 24th International Conference (FSTTCS)*, volume 3328 of *LNCS*, pages 97–109. Springer, 2004.

26. J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *Programming Languages and Systems, 3rd Asian Symposium, (APLAS)*, volume 3780 of *LNCS*, pages 52–68. Springer, 2005.
27. J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects (FMCO)*, volume 4111 of *LNCS*, pages 115–137. Springer, 2006.
28. D. Beyer, T. A. Henzinger, and G. Théoduloz. Lazy shape analysis. In *Computer Aided Verification, 18th International Conference (CAV)*, volume 4144 of *LNCS*, pages 532–546. Springer, 2006.
29. A. Biere. Resolve and expand. In *7th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, 2004.
30. A. Biere. PicoSAT essentials. *JSAT*, 4(2-4):75–97, 2008.
31. A. Biere. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. SAT race: system description, 2010.
32. A. Biere, D. L. Berre, E. Lonca, and N. Manthey. Detecting cardinality constraints in CNF. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing (SAT), 17th International Conference*, volume 8561 of *LNCS*, pages 285–301. Springer, 2014.
33. A. Biere, A. Cimatti, E. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Workshop on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1579 of *LNCS*, pages 193–207. Springer, 1999.
34. A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Design Automation Conference (DAC)*, pages 317–320. ACM, 1999.
35. A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, 2009.
36. J. D. Bingham and Z. Rakamaric. A logic and decision procedure for predicate abstraction of heap-manipulating programs. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *LNCS*, pages 207–221. Springer, 2006.
37. C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *23rd International Conference on Data Engineering (ICDE)*, pages 506–515. IEEE, 2007.
38. C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. QAGen: generating query-aware test databases. In *ACM SIGMOD International Conference on Management of Data*, pages 341–352. ACM, 2007.
39. N. Bjørner. Linear quantifier elimination as an abstract decision procedure. In J. Giesl and R. Hähnle, editors, *Automated Reasoning, 5th International Joint Conference (IJCAR)*, volume 6173 of *LNCS*, pages 316–330. Springer, 2010.
40. M. P. Bonacina, S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decidability and undecidability results for Nelson–Oppen and rewrite-based decision procedures. In *Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 513–527. Springer, 2006.
41. M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, S. Ranise, P. van Rossum, and R. Sebastiani. Efficient satisfiability modulo theories via delayed theory combination. In *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 335–349, 2005.

42. A. R. Bradley. SAT-based model checking without unrolling. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 6538 of *Lecture Notes in Computer Science*, pages 70–87, 2011.
43. A. R. Bradley and Z. Manna. *The Calculus of Computation*. Springer, 2007.
44. A. R. Bradley, Z. Manna, and H. B. Sipma. What's decidable about arrays? In *Verification, Model Checking, and Abstract Interpretation, 7th International Conference (VMCAI)*, volume 3855 of *LNCS*, pages 427–442. Springer, 2006.
45. M. Brain, V. D'Silva, A. Griggio, L. Haller, and D. Kroening. Deciding floating-point logic with abstract conflict driven clause learning. *Formal Methods in System Design*, 45(2):213–245, 2014.
46. M. Brain, L. Hadarean, D. Kroening, and R. Martins. Automatic generation of propagation complete SAT encodings. In B. Jobstmann and K. R. M. Leino, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 9583 of *LNCS*, pages 536–556. Springer, 2016.
47. M. Brain, C. Tinelli, P. Rümmer, and T. Wahl. An automatable formal semantics for IEEE-754 floating-point arithmetic. In *22nd IEEE Symposium on Computer Arithmetic (ARITH)*, pages 160–167. IEEE, 2015.
48. A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.
49. A. Brillout, D. Kroening, and T. Wahl. Mixed abstractions for floating-point arithmetic. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 69–76. IEEE, 2009.
50. R. Brinkmann and R. Drechsler. RTL-datapath verification using integer linear programming. In *Proceedings of VLSI Design*, pages 741–746. IEEE, 2002.
51. R. Brummayer and A. Biere. C32SAT: Checking C expressions. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 294–297. Springer, 2007.
52. R. Brummayer and A. Biere. Boolector: An efficient SMT solver for bit-vectors and arrays. In S. Kowalewski and A. Philippou, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 5505 of *LNCS*, pages 174–177. Springer, 2009.
53. R. Brummayer and A. Biere. Lemmas on demand for the extensional theory of arrays. *JSAT*, 6(1-3):165–201, 2009.
54. R. Bruttomesso, A. Cimatti, A. Franzén, A. Griggio, Z. Hanna, A. Nadel, A. Palti, and R. Sebastiani. A lazy and layered SMT($\mathcal{BV}$) solver for hard industrial verification problems. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 547–560. Springer, 2007.
55. R. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(12):1035–1044, 1986.
56. R. Bryant, S. German, and M. Velev. Exploiting positive equality in a logic of equality with uninterpreted functions. In *11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *LNCS*. Springer, 1999.
57. R. Bryant and M. Velev. Boolean satisfiability with transitivity constraints. In *12th International Conference on Computer Aided Verification (CAV)*, volume 1855 of *LNCS*. Springer, 2000.
58. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*. Springer, 2002.

59. H. K. Büning, M. Karpinski, and A. Flögel. Resolution for quantified Boolean formulas. *Information and Computation*, 117(1):12–18, 1995.
60. R. Burstall. Some techniques for proving correctness of programs which alter data structures. *Machine Intelligence*, 7:23–50, 1972.
61. W. R. Bush, J. D. Pincus, and D. J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw., Pract. Exper.*, 30(7):775–802, 2000.
62. C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *OSDI*, pages 209–224. USENIX Association, 2008.
63. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. In *Static Analysis, 13th International Symposium (SAS)*, volume 4134 of *LNCS*, pages 182–203. Springer, 2006.
64. C. Calcagno, H. Yang, and P. W. O'Hearn. Computability and complexity results for a spatial assertion language for data structures. In *Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2245 of *LNCS*, pages 108–119. Springer, 2001.
65. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In L. A. Clarke, L. Dillon, and W. F. Tichy, editors, *ICSE*, pages 385–395. IEEE Computer Society, 2003.
66. S. Chaki, A. Gurfinkel, and O. Strichman. Regression verification for multi-threaded programs (with extensions to locks and dynamic thread creation). *Formal Methods in System Design*, 47(3):287–301, 2015.
67. P. Cheeseman, B. Kanefsky, and W. M. Taylor. Where the really hard problems are. In *12th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 331–337, 1991.
68. H. Chen, C. David, D. Kroening, P. Schrammel, and B. Wachter. Synthesising interprocedural bit-precise termination proofs. In M. B. Cohen, L. Grunske, and M. Whalen, editors, *Automated Software Engineering (ASE)*, pages 53–64. IEEE, 2015.
69. Y. Chen, N. Dalchau, N. Srinivas, A. Phillips, L. Cardelli, D. Soloveichik, and G. Seelig. Programmable chemical controllers made from DNA. *Nature nanotechnology*, 8(10):755–762, 2013.
70. J. Christ and J. Hoenicke. Weakly equivalent arrays. In P. Rümmer and C. M. Wintersteiger, editors, *Satisfiability Modulo Theories (SMT)*, volume 1163 of *CEUR Workshop Proceedings*, pages 39–49, 2014.
71. A. Cimatti, A. Griggio, B. Schaafsma, and R. Sebastiani. The MathSAT5 SMT solver. In N. Piterman and S. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
72. E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.
73. E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods in System Design*, 25(2-3):105–127, 2004.
74. B. Cook, D. Kroening, P. Rümmer, and C. M. Wintersteiger. Ranking function synthesis for bit-vector relations. *Formal Methods in System Design*, 43(1):93–120, 2013.

75. B. Cook, D. Kroening, and N. Sharygina. Accurate theorem proving for program verification. In *Leveraging Applications of Formal Methods, First International Symposium (ISoLA)*, volume 4313 of *LNCS*, pages 96–114. Springer, 2006.
76. B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In M. I. Schwartzbach and T. Ball, editors, *ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation (PLDI)*, pages 415–426. ACM, 2006.
77. S. Cook. The complexity of theorem-proving procedures. In *3rd Annual ACM Symposium on Theory of Computing*, pages 151–158, 1971.
78. D. C. Cooper. Theorem proving in arithmetic without multiplication. In *Machine Intelligence*, pages 91–100. Edinburgh University Press, 1972.
79. F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 436–453, 2001.
80. L. Cordeiro, B. Fischer, and J. Marques-Silva. SMT-based Bounded Model Checking for embedded ANSI-C software. In *ASE*, pages 137–148. IEEE Computer Society, 2009.
81. F. Corella, Z. Zhou, X. Song, M. Langevin, and E. Cerny. Multiway decision graphs for automated hardware verification. *Formal Methods in System Design*, 10(1):7–46, 1997.
82. T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*, chapter 25.3, pages 532–536. MIT Press, 2000.
83. D. W. Currie, A. J. Hu, and S. P. Rajan. Automatic formal verification of DSP software. In *Design Automation Conference (DAC)*, pages 130–135. ACM, 2000.
84. G. B. Dantzig. *Linear Programming and Extensions.* Princeton University Press, 1963.
85. C. David, D. Kroening, and M. Lewis. Propositional reasoning about safety and termination of heap-manipulating programs. In J. Vitek, editor, *Programming Languages and Systems, 24th European Symposium on Programming (ESOP)*, volume 9032 of *LNCS*, pages 661–684. Springer, 2015.
86. C. David, D. Kroening, and M. Lewis. Unrestricted termination and non-termination arguments for bit-vector programs. In J. Vitek, editor, *Programming Languages and Systems, 24th European Symposium on Programming (ESOP)*, volume 9032 of *LNCS*, pages 183–204. Springer, 2015.
87. M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5:394–397, 1962.
88. M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7:201–215, 1960.
89. H. de Jong. Modeling and simulation of genetic regulatory systems: A literature review. *Journal of Computational Biology*, 9(1):67–103, 2002.
90. L. de Moura. System description: Yices 0.1. Technical report, Computer Science Laboratory, SRI International, 2005.
91. L. de Moura and N. Bjørner. Model-based theory combination. In *Satisfiability Modulo Theories (SMT)*, 2007.
92. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and*

Analysis of Systems, 14th International Conference (TACAS), volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
93. L. de Moura and H. Ruess. Lemmas on demand for satisfiability solvers. In *Theory and Applications of Satisfiability Testing (SAT)*, 2002.
94. L. de Moura and H. Ruess. An experimental evaluation of ground decision procedures. In *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 162–174. Springer, 2004.
95. L. de Moura, H. Ruess, and N. Shankar. Justifying equality. In *Second Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR)*, 2004.
96. L. M. de Moura and N. Bjørner. Efficient E-matching for SMT solvers. In *Automated Deduction, 21st International Conference on Automated Deduction (CADE)*, volume 4603 of *LNCS*, pages 183–198. Springer, 2007.
97. L. M. de Moura and N. Bjørner. Generalized, efficient array decision procedures. In *9th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 45–52, 2009.
98. R. Dechter. *Constraint Processing*. The Morgan Kaufmann Series in Artificial Intelligence. Morgan Kaufmann, 2003.
99. N. Dershowitz, Z. Hanna, and A. Nadel. A clause-based heuristic for SAT solvers. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing, 8th International Conference (SAT)*, volume 3569 of *LNCS*, pages 46–60. Springer, 2005.
100. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science (Volume B): Formal Models and Semantics*, pages 243–320. MIT Press, 1990.
101. D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3):365–473, May 2005.
102. A. Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Programming Language Design and Implementation (PLDI)*, pages 230–241. ACM, 1994.
103. D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 3920 of *LNCS*, pages 287–302. Springer, 2006.
104. A. Dolzmann, T. Sturm, and V. Weispfenning. Real quantifier elimination in practice. Technical Report MIP9720, FMI, Universität Passau, Dec. 1997.
105. P. J. Downey. Undecidability of Presburger arithmetic with a single monadic predicate letter. Technical Report TR-18-72, Center for Research in Computing Technology, Harvard University, 1972.
106. P. J. Downey, R. Sethi, and R. E. Tarjan. Variations on the common subexpression problem. *J. ACM*, 27(4):758–771, October 1980.
107. S. Dunn, G. Martello, B. Yordanov, S. Emmott, and A. Smith. Defining an essential transcription factor program for naïve pluripotency. *Science*, 344(6188), 2014.
108. B. Dutertre and L. de Moura. A fast linear-arithmetic solver for DPLL(T). In *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 81–94. Springer, 2006.
109. B. Dutertre and L. de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, Stanford Research Institute (SRI), 2006.

110. N. Eén and N. Sörensson. An extensible SAT-solver [ver 1.2]. In *Theory and Applications of Satisfiability Testing*, volume 2919 of *LNCS*, pages 512–518. Springer, 2003.
111. D. R. Engler, D. Y. Chen, and A. Chou. Bugs as inconsistent behavior: A general approach to inferring errors in systems code. In *18th ACM Symposium on Operating System Principles (SOSP)*, pages 57–72. ACM, 2001.
112. J. Ferrante and C. Rackoff. A decision procedure for the first order theory of real addition with order. *SIAM J. Comput.*, 4(1):69–76, 1975.
113. J. Filliatre, S. Owre, H. Ruess, and N. Shankar. ICS: Integrated canonizer and solver. In *13th International Conference on Computer Aided Verification (CAV)*, volume 2102 of *LNCS*, pages 246–249. Springer, 2001.
114. M. J. Fischer and M. O. Rabin. Super-exponential complexity of Presburger arithmetic. In *SIAM-AMS Symposium in Applied Mathematics*, volume 7, pages 27–41, 1974.
115. C. Flanagan, R. Joshi, X. Ou, and J. B. Saxe. Theorem proving using lazy proof explication. In *Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 355–367. Springer, 2003.
116. R. Floyd. Assigning meanings to programs. *Symposia in Applied Mathematics*, 19:19–32, 1967.
117. V. Ganesh, S. Berezin, and D. Dill. Deciding Presburger arithmetic by model checking and comparisons with other methods. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 of *LNCS*, pages 171–186. Springer, 2002.
118. V. Ganesh and D. L. Dill. A decision procedure for bit-vectors and arrays. In *Computer Aided Verification (CAV)*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.
119. H. Ganzinger. Shostak light. In A. Voronkov, editor, *Automated Deduction, 18th International Conference on Automated Deduction (CADE)*, volume 2392 of *LNCS*, pages 332–346. Springer, 2002.
120. H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. DPLL(T): Fast decision procedures. In *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 175–188. Springer, 2004.
121. Y. Ge, C. W. Barrett, and C. Tinelli. Solving quantified verification conditions using satisfiability modulo theories. *Ann. Math. Artif. Intell.*, 55(1-2):101–122, 2009.
122. A. V. Gelder and Y. Tsuji. Incomplete thoughts about incomplete satisfiability procedures. In *2nd DIMACS Challenge Workshop: Cliques, Coloring and Satisfiability*, 1993.
123. R. Gershman, M. Koifman, and O. Strichman. Deriving small unsatisfiable cores with dominators. In *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 109–122. Springer, 2006.
124. R. Gershman and O. Strichman. HaifaSat: A new robust SAT solver. In *1st International Haifa Verification Conference*, volume 3875 of *LNCS*, pages 76–89. Springer, 2005.
125. M. Ghasemzadeh, V. Klotz, and C. Meinel. Embedding memoization to the semantic tree search for deciding QBFs. In *Australian Conference on Artificial Intelligence*, pages 681–693, 2004.

126. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Deciding extensions of the theory of arrays by integrating decision procedures and instantiation strategies. In *Logics in Artificial Intelligence (JELIA)*, volume 4160 of *LNCS*, pages 177–189. Springer, 2006.
127. S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision procedures for extensions of the theory of arrays. *Ann. Math. Artif. Intell.*, 50(3-4):231–254, 2007.
128. E. Giunchiglia, M. Narizzano, and A. Tacchella. Quantified Boolean Formulas satisfiability library (QBFLIB), 2001. `www.qbflib.org`.
129. F. Giunchiglia and R. Sebastiani. Building decision procedures for modal logics from propositional decision procedures – the case study of modal K. In *Automated Deduction (CADE)*, volume 1104 of *LNCS*, pages 583–597. Springer, 1996.
130. P. Godefroid, J. de Halleux, A. V. Nori, S. K. Rajamani, W. Schulte, N. Tillmann, and M. Y. Levin. Automating software testing using program analysis. *IEEE Software*, 25(5):30–37, 2008.
131. P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *PLDI*, pages 213–223. ACM, 2005.
132. A. Goel, K. Sajid, H. Zhou, A. Aziz, and V. Singhal. BDD based procedures for a theory of equality with uninterpreted functions. In *Computer Aided Verification, 10th International Conference (CAV)*, volume 1427 of *LNCS*, pages 244–255. Springer, 1998.
133. E. Goldberg and Y. Novikov. Berkmin: A fast and robust SAT-solver. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, page 142, 2002.
134. R. Gomory. An algorithm for integer solutions to linear problems. In *Recent Advances in Mathematical Programming*, pages 269–302, New York, 1963. McGraw-Hill.
135. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In O. Grumberg, editor, *Computer Aided Verification, 9th International Conference (CAV)*, volume 1254 of *LNCS*. Springer, 1997.
136. L. Hadarean, K. Bansal, D. Jovanovic, C. Barrett, and C. Tinelli. A tale of two solvers: Eager and lazy approaches to bit-vectors. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 680–695. Springer, 2014.
137. A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.
138. J. Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2008.
139. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Symposium on Principles of Programming Languages*, pages 58–70, 2002.
140. M. Heule, O. Kullmann, S. Wieringa, and A. Biere. Cube and conquer: Guiding CDCL SAT solvers by lookaheads. In *Haifa Verification Conference*, volume 7261 of *LNCS*, pages 50–65. Springer, 2011.
141. J. Heusser and P. Malacaria. Quantifying information leaks in software. In C. Gates, M. Franz, and J. P. McDermott, editors, *Twenty-Sixth Annual Computer Security Applications Conference (ACSAC)*, pages 261–269. ACM, 2010.
142. F. Hillier and G. Lieberman. *Introduction to Mathematical Programming*. McGraw-Hill, 1990.

143. E. A. Hirsch and A. Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.
144. C. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–580, 1969.
145. C. A. R. Hoare and N. Wirth. An axiomatic definition of the programming language PASCAL. *Acta Informatica*, 2(4):335–355, December 1973.
146. R. Hojati, A. Isles, D. Kirkpatrick, and R. Brayton. Verification using uninterpreted functions and finite instantiations. In *1st International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, volume 1166 of *LNCS*, pages 218–232. Springer, 1996.
147. R. Hojati, A. Kuehlmann, S. German, and R. Brayton. Validity checking in the theory of equality using finite instantiations. In *International Workshop on Logic Synthesis*, 1997.
148. J. N. Hooker. Solving the incremental satisfiability problem. *Journal of Logic Programming*, 15:177–186, 1993.
149. I. Horrocks. The FaCT system. In H. de Swart, editor, *TABLEAUX-98*, volume 1397 of *LNAI*, pages 307–312. Springer, 1998.
150. J. Huang. MUP: A minimal unsatisfiability prover. In *10th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 432–437, 2005.
151. G. Huet and D. Oppen. Equations and rewrite rules: A survey. In *Formal Language Theory: Perspectives and Open Problems*, pages 349–405. Academic Press, 1980.
152. A. P. Hurst, P. Chong, and A. Kuehlmann. Physical placement driven by sequential timing analysis. In *International Conference on Computer-Aided Design (ICCAD)*, pages 379–386. IEEE Computer Society/ACM, 2004.
153. N. Immerman, A. M. Rabinovich, T. W. Reps, S. Sagiv, and G. Yorsh. The boundary between decidability and undecidability for transitive-closure logics. In *Computer Science Logic (CSL)*, volume 3210 of *LNCS*, pages 160–174. Springer, 2004.
154. O. Inverso, E. Tomasco, B. Fischer, S. La Torre, and G. Parlato. Bounded model checking of multi-threaded C programs via lazy sequentialization. In A. Biere and R. Bloem, editors, *Computer Aided Verification (CAV)*, volume 8559 of *LNCS*, pages 585–602. Springer, 2014.
155. F. Ivancic, I. Shlyakhter, A. Gupta, and M. K. Ganai. Model checking C programs using F-SOFT. In *International Conference on Computer Design (ICCD)*, pages 297–308. IEEE Computer Society, 2005.
156. J. Jaffar. Presburger arithmetic with array segments. *Inf. Process. Lett.*, 12(2):79–82, 1981.
157. G. Johnson. Separating the insolvable and the merely difficult. New York Times, July 13 1999.
158. D. Jovanovic and C. Barrett. Polite theories revisited. In C. G. Fermüller and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 17th International Conference (LPAR)*, volume 6397 of *LNCS*, pages 402–416. Springer, 2010.
159. D. Jovanovic and C. Barrett. Sharing is caring: Combination of theories. In C. Tinelli and V. Sofronie-Stokkermans, editors, *Frontiers of Combining Systems, 8th International Symposium (FroCoS)*, volume 6989 of *LNCS*, pages 195–210. Springer, 2011.

160. T. Jussila, A. Biere, C. Sinz, D. Kroening, and C. M. Wintersteiger. A first step towards a unified proof checker for QBF. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4501 of *LNCS*, pages 201–214. Springer, 2007.
161. A. Kaiser, D. Kroening, and T. Wahl. Efficient coverability analysis by proof minimization. In M. Koutny and I. Ulidowski, editors, *CONCUR*, volume 7454 of *LNCS*, pages 500–515. Springer, 2012.
162. M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Kluwer Academic, 2000.
163. S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference (TACAS)*, pages 553–568, 2003.
164. J. Kim, J. Whittemore, J. Silva, and K. Sakallah. Incremental Boolean satisfiability and its application to delay fault testing. In *IEEE/ACM International Workshop on Logic Synthesis (IWLS)*, June 1999.
165. J. C. King. A new approach to program testing. In C. Hackl, editor, *Programming Methodology*, volume 23 of *LNCS*, pages 278–290. Springer, 1974.
166. S. Kirkpatrick and B. Selman. Critical behavior in the satisfiability of random Boolean expressions. *Science*, 264(5163):1297–1301, 1994.
167. D. E. Knuth. *The Art of Computer Programming Volume 4, Fascicle 1*. Addison-Wesley Professional, 2009. Bitwise tricks & techniques; Binary Decision Diagrams.
168. D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 6*. Pearson Education, 2016. Satisfiability.
169. L. Kovács and A. Voronkov. First-order theorem proving and Vampire. In N. Sharygina and H. Veith, editors, *Computer Aided Verification, 25th International Conference (CAV)*, volume 8044 of *LNCS*, pages 1–35. Springer, 2013.
170. D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Design Automation Conference (DAC)*, pages 368–371. ACM, 2003.
171. D. Kroening and O. Strichman. A framework for satisfiability modulo theories. *Formal Aspects of Computing*, 21(5):485–494, 2009.
172. V. Kuncak and M. C. Rinard. Existential heap abstraction entailment is undecidable. In *Static Analysis (SAS)*, volume 2694 of *LNCS*, pages 418–438. Springer, 2003.
173. R. Kurshan. *Computer Aided Verification of Coordinating Processes*. Princeton University Press, 1994.
174. S. K. Lahiri, R. E. Bryant, A. Goel, and M. Talupur. Revisiting positive equality. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of *LNCS*, pages 1–15. Springer, 2004.
175. S. K. Lahiri and S. Qadeer. Verifying properties of well-founded linked lists. In *Principles of Programming Languages (POPL)*, pages 115–126. ACM, 2006.
176. S. K. Lahiri and S. Qadeer. Back to the future: Revisiting precise program verification using SMT solvers. In *Principles of Programming Languages (POPL)*, pages 171–182. ACM, 2008.
177. A. Lal and S. Qadeer. Powering the Static Driver Verifier using Corral. In S. Cheung, A. Orso, and M. D. Storey, editors, *22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 202–212. ACM, 2014.

178. A. Lal, S. Qadeer, and S. K. Lahiri. A solver for reachability modulo theories. In P. Madhusudan and S. A. Seshia, editors, *Computer Aided Verification (CAV)*, volume 7358 of *LNCS*, pages 427–443. Springer, 2012.
179. A. Lal and T. W. Reps. Reducing concurrent analysis under a context bound to sequential analysis. In A. Gupta and S. Malik, editors, *Computer Aided Verification (CAV)*, volume 5123 of *LNCS*, pages 37–51. Springer, 2008.
180. S. Lee and D. Plaisted. Eliminating duplication with the hyper-linking strategy. *Journal of Automated Reasoning*, 9:25–42, 1992.
181. K. R. M. Leino. *Toward reliable modular programs.* PhD thesis, CalTech, 1995. Available as Technical Report Caltech-CS-TR-95-03.
182. M. Levine and E. Davidson. Gene regulatory networks for development. *Proceedings of the National Academy of Sciences*, 102(14):4936–4942, 2005.
183. R. Loos and V. Weispfenning. Applying linear quantifier elimination. *Comput. J.*, 36(5):450–462, 1993.
184. I. Lynce and J. Marques-Silva. On computing minimum unsatisfiable cores. In *International Symposium on Theory and Applications of Satisfiability Testing (SAT)*, pages 305–310, 2004.
185. M. Mahfoudh. *On Satisfiability Checking for Difference Logic.* PhD thesis, Verimag, France, 2003.
186. M. Mahfoudh, P. Niebert, E. Asarin, and O. Maler. A satisfiability checker for difference logic. In *5th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 222–230, 2002.
187. R. Manevich, E. Yahav, G. Ramalingam, and S. Sagiv. Predicate abstraction and canonical abstraction for singly-linked lists. In *Verification, Model Checking, and Abstract Interpretation, 6th International Conference (VMCAI)*, volume 3385 of *LNCS*, pages 181–198. Springer, 2005.
188. P. Manolios, S. K. Srinivasan, and D. Vroon. Automatic memory reductions for RTL model verification. In *International Conference on Computer-Aided Design (ICCAD)*, pages 786–793. ACM, 2006.
189. P. Manolios and D. Vroon. Efficient circuit to CNF conversion. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 4501 of *LNCS*, pages 4–9. Springer, 2007.
190. P. Mateti. A decision procedure for the correctness of a class of programs. *J. ACM*, 28(2):215–232, 1981.
191. J. McCarthy and J. Painter. Correctness of a compiler for arithmetic expressions. In *Proceedings Symposium in Applied Mathematics, Volume 19, Mathematical Aspects of Computer Science*, pages 33–41. American Mathematical Society, 1967.
192. K. McMillan. *Symbolic Model Checking.* Kluwer Academic, 1993.
193. K. L. McMillan. Interpolation and SAT-based model checking. In *15th International Conference on Computer Aided Verification (CAV)*, volume 2725 of *LNCS*, pages 1–13. Springer, Jul 2003.
194. O. Meir and O. Strichman. Yet another decision procedure for equality logic. In *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 307–320. Springer, 2005.
195. F. Merz, S. Falke, and C. Sinz. LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In R. Joshi, P. Müller, and A. Podelski, editors, *VSTTE*, volume 7152 of *LNCS*, pages 146–161. Springer, 2012.
196. M. Mezard, G. Parisi, and R. Zecchina. Analytic and algorithmic solution of random satisfiability problems. *Science*, 297(5582):812–815, 2002.

197. D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions for SAT problems. In *Tenth National Conference on Artificial Intelligence*, pages 459–465. AAAI Press, 1992.
198. A. Møller and M. I. Schwartzbach. The pointer assertion logic engine. In *2001 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 221–231. ACM, 2001.
199. J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *13th International Conference on Computer Science Logic*, volume 1683 of *LNCS*, pages 111–125. Springer, 1999.
200. R. Monasson, R. Zecchina, S. Kirkpatrick, B. Selman, and L. Troyansky. Determining computational complexity from characteristic phase transitions. *Nature*, 400(8):133–137, 1999.
201. U. Montanari. Networks of constraints: fundamental properties and applications to picture processing. *Information Science*, 7, 1976.
202. M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference (DAC)*, June 2001.
203. A. Nadel. Boosting minimal unsatisfiable core extraction. In R. Bloem and N. Sharygina, editors, *10th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 221–229. IEEE, 2010.
204. G. Nelson. Verifying reachability invariants of linked structures. In *Tenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, pages 38–47. ACM, 1983.
205. G. Nelson and D. C. Oppen. Fast decision procedures based on UNION and FIND. In *18th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 114–119. IEEE, 1977.
206. G. Nelson and D. C. Oppen. Simplification by cooperating decision procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, Oct 1979.
207. G. Nelson and F. F. Yao. Solving reachability constraints for linear lists. Technical report, 1982.
208. G. Nemhauser and L. Wosley. *Integer and Combinatorial Optimization*. Wiley-Interscience, 1999.
209. R. Nieuwenhuis and A. Oliveras. DPLL(T) with exhaustive theory propagation and its application to difference logic. In *17th International Conference on Computer Aided Verification (CAV)*, volume 3576 of *LNCS*, pages 321–334. Springer, 2005.
210. R. Nieuwenhuis and A. Oliveras. Proof-producing congruence closure. In *16th International Conference on Term Rewriting and Applications (RTA)*, volume 3467 of *LNCS*, pages 453–468. Springer, 2005.
211. R. Nieuwenhuis, A. Oliveras, and C. 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.
212. E. Nudelman, K. Leyton-Brown, H. H. Hoos, A. Devkar, and Y. Shoham. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Principles and Practice of Constraint Programming (CP)*, pages 438–452, 2004.
213. C. Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In M. Heule and S. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT*, volume 9340 of *LNCS*, pages 307–323, 2015.

214. Y. Oh, M. N. Mneimneh, Z. S. Andraus, K. A. Sakallah, and I. L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor. In *Design Automation Conference (DAC)*, pages 518–523, 2004.
215. D. C. Oppen. Complexity, convexity and combinations of theories. *Theoretical Computer Science*, 12(3):291–302, 1980.
216. N. Paoletti, B. Yordanov, Y. Hamadi, C. Wintersteiger, and H. Kugler. Analyzing and synthesizing genomic logic functions. In *CAV*, volume 8559 of *LNCS*. Springer, 2014.
217. M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 247–258. ACM, 2005.
218. G. Parthasarathy, M. K. Iyer, K.-T. Cheng, and L.-C. Wang. An efficient finite-domain constraint solver for circuits. In *Design Automation Conference (DAC)*, pages 212–217. ACM Press, 2004.
219. P. F. Patel-Schneider. DLP system description. In E. Franconi, G. D. Giacomo, R. M. MacGregor, W. Nutt, and C. A. Welty, editors, *International Workshop on Description Logics (DL)*, volume 11 of *CEUR Workshop Proceedings*, pages 87–89, 1998.
220. J. Petke and P. Jeavons. The order encoding: From tractable CSP to tractable SAT. In *Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *Lecture Notes in Computer Science*, pages 371–372. Springer, 2011.
221. A. Phillips and L. Cardelli. A programming language for composable DNA circuits. *Journal of The Royal Society Interface*, 6(4):1470–1485, 2009.
222. K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In *10th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 294–299, 2007.
223. K. Pipatsrisawat and A. Darwiche. RSAT 2.0: SAT solver description, 2007. SAT solvers competition.
224. R. Piskac, L. de Moura, and N. Bjørner. Deciding effectively propositional logic with equality. Technical Report MSR-TR-2008-181, Microsoft Research, 2008.
225. D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2(3):293–304, September 1986.
226. A. Pnueli, Y. Rodeh, and O. Shtrichman. Range allocation for equivalence logic. In R. Hariharan, M. Mukund, and V. Vinay, editors, *21st Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, volume 2245 of *LNCS*, pages 317–333. Springer, 2001.
227. A. Pnueli, Y. Rodeh, O. Shtrichman, and M. Siegel. Deciding equality formulas by small-domains instantiations. In *11th International Conference on Computer Aided Verification (CAV)*, volume 1633 of *LNCS*. Springer, 1999.
228. A. Pnueli, Y. Rodeh, O. Strichman, and M. Siegel. The small model property: How small can it be? *Information and Computation*, 178(1):279–293, Oct. 2002.
229. A. Pnueli and O. Strichman. Reduced functional consistency of uninterpreted functions. In *Pragmatics of Decision Procedures for Automated Reasoning (PDPAR)*, number 898 in Electronic Notes in Computer Science, 2005.
230. A. Podelski and T. Wies. Boolean heaps. In *Static Analysis, 12th International Symposium (SAS)*, volume 3672 of *LNCS*, pages 268–283. Springer, 2005.
231. V. Pratt. Two easy theories whose combination is hard. Technical report, Massachusetts Institute of Technology, 1977.

232. M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du I congrès de Mathématiciens des Pays Slaves*, pages 92–101, Warszawa, 1929.
233. W. Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.
234. L. Qian and E. Winfree. Scaling up digital circuit computation with DNA strand displacement cascades. *Science*, 332(6034):1196–1201, 2011.
235. L. Qian, E. Winfree, and J. Bruck. Neural network computation with DNA strand displacement cascades. *Nature*, 475(7356):368–372, 2011.
236. M. O. Rabin. Decidability of second-order theories and automata on infinite trees. *Transactions of the American Mathematical Society*, 141:1–35, July 1969.
237. I. Rabinovitz and O. Grumberg. Bounded model checking of concurrent programs. In K. Etessami and S. K. Rajamani, editors, *Computer Aided Verification, 17th International Conference (CAV)*, volume 3576 of *LNCS*, pages 82–97. Springer, 2005.
238. S. Ranise, C. Ringeissen, and C. G. Zarba. Combining data structures with nonstably infinite theories using many-sorted logic. In B. Gramlich, editor, *Frontiers of Combining Systems, 5th International Workshop (FroCoS)*, volume 3717 of *LNCS*, pages 48–64. Springer, 2005.
239. S. Ranise and C. Tinelli. The SMT-LIB standard: Version 1.2. Technical report, Department of Computer Science, The University of Iowa, 2006. Available at `www.SMT-LIB.org`.
240. T. W. Reps, S. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In *Computer Aided Verification, 16th International Conference (CAV)*, volume 3114 of *LNCS*, pages 15–30. Springer, 2004.
241. J. C. Reynolds. Reasoning about arrays. *Communications of the ACM*, 22(5):290–299, 1979.
242. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.
243. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.
244. J. A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.
245. Y. Rodeh. *Techniques in Decision Procedures for Equality Logic and Improved Model Checking Methods*. PhD thesis, Weizmann Institute of Science, 2003.
246. Y. Rodeh and O. Shtrichman. Finite instantiations in equivalence logic with uninterpreted functions. In *Computer Aided Verification (CAV)*, 2001.
247. M. Rozanov and O. Strichman. Generating minimum transitivity constraints in P-time for deciding equality logic. In *Satisfiability Modulo Theories (SMT)*, 2007.
248. L. Ryan. Efficient algorithms for clause-learning SAT solvers. Master's thesis, Simon Fraser University, 2004.
249. V. Ryvchin and O. Strichman. Faster extraction of high-level minimal unsatisfiable cores. In K. A. Sakallah and L. Simon, editors, *Theory and Applications of Satisfiability Testing (SAT)*, volume 6695 of *LNCS*, pages 174–187. Springer, 2011.

250. A. Sabharwal, C. Ansótegui, C. P. Gomes, J. W. Hart, and B. Selman. QBF modeling: Exploiting player symmetry for simplicity and efficiency. In *9th International Conference on Theory and Applications of Satisfiability Testing (SAT)*, pages 382–395, 2006.
251. S. Sagiv, T. W. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *POPL*, pages 105–118, 1999.
252. A. Schrijver. *Theory of Linear and Integer Programming*. Wiley, 1998.
253. R. Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation, JSAT*, 3, 2007.
254. B. Selman, H. J. Levesque, and D. G. Mitchell. A new method for solving hard satisfiability problems. In *10th National Conference on Artificial Intelligence (AAAI)*, pages 440–446, 1992.
255. K. Sen and G. Agha. CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *18th International Conference on Computer Aided Verification (CAV)*, volume 4144 of *LNCS*, pages 419–423. Springer, 2006.
256. S. Seshia, S. Lahiri, and R. Bryant. A hybrid SAT-based decision procedure for separation logic with uninterpreted functions. In *Proc. of Design Automation Conference (DAC)*, pages 425–430, 2003.
257. N. Shankar and H. Ruess. Combining Shostak theories. In S. Tison, editor, *Rewriting Techniques and Applications, 13th International Conference (RTA)*, volume 2378 of *LNCS*, pages 1–18. Springer, 2002.
258. R. E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.
259. R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31(1):1–12, January 1984.
260. O. Shtrichman. Sharing information between instances of a propositional satisfiability (SAT) problem, Dec 2000. US Patent 2002/0123867 A1.
261. O. Shtrichman. Pruning techniques for the SAT-based bounded model checking problem. In *11th Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME)*, volume 2144 of *LNCS*, pages 58–70. Springer, 2001.
262. J. Silva and K. Sakallah. GRASP – a new search algorithm for satisfiability. Technical Report TR-CSE-292996, University of Michigan, 1996.
263. E. Singerman. Challenges in making decision procedures applicable to industry. In PDPAR 2005. *Electronic Notes in Computer Science*, 144(2), 2006.
264. L. Stockmeyer. The polynomial-time hierarchy. *Theoretical Computer Science*, 3:1–22, 1977.
265. L. Stockmeyer and A. Meyer. Word problems requiring exponential time. In *5th Annual ACM Symposium on Theory of Computing (STOC)*, pages 1–9. ACM, 1973.
266. O. Strichman. On solving Presburger and linear arithmetic with SAT. In *Formal Methods in Computer-Aided Design (FMCAD)*, volume 2517 of *LNCS*, pages 160–170. Springer, 2002.
267. O. Strichman, S. Seshia, and R. Bryant. Deciding separation formulas with SAT. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 209–222. Springer, July 2002.
268. A. Stump, C. Barrett, and D. Dill. CVC: a cooperating validity checker. In *14th International Conference on Computer Aided Verification (CAV)*, volume 2404 of *LNCS*, pages 500–504. Springer, 2002.

269. A. Stump, C. W. Barrett, D. L. Dill, and J. R. Levitt. A decision procedure for an extensional theory of arrays. In *16th Annual IEEE Symposium on Logic in Computer Science (LICS)*, pages 29–37. IEEE, 2001.
270. A. Stump and L.-Y. Tan. The algebra of equality proofs. In *Term Rewriting and Applications, 16th International Conference (RTA)*, volume 3467 of *LNCS*, pages 469–483. Springer, 2005.
271. G. Sutcliffe and C. Suttner. The state of CASC. *AI Communications*, 19(1):35–48, 2006.
272. N. Suzuki and D. Jefferson. Verification decidability of Presburger array programs. *J. ACM*, 27(1):191–205, 1980.
273. M. Talupur, N. Sinha, O. Strichman, and A. Pnueli. Range-allocation for separation logic. In *16th International Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 148–161. Springer, July 2004.
274. C. Tinelli. A DPLL-based calculus for ground satisfiability modulo theories. In *Proc. 8th European Conference on Logics in Artificial Intelligence*, volume 2424 of *LNAI*, pages 308–319. Springer, 2002.
275. C. Tinelli and M. T. Harandi. A new correctness proof of the Nelson–Oppen combination procedure. In *Frontiers of Combining Systems: 1st International Workshop*, pages 103–120. Kluwer Academic, 1996.
276. C. Tinelli and C. Zarba. Combining nonstably infinite theories. *Journal of Automated Reasoning*, 34(3):209–238, 2005.
277. G. Tseitin. On the complexity of proofs in propositional logics. In *Automation of Reasoning: Classical Papers in Computational Logic 1967–1970*, volume 2. Springer, 1983. Originally published 1970.
278. R. J. Vanderbei. *Linear Programming: Foundations and Extensions*. Kluwer, 1996.
279. M. Veksler and O. Strichman. Learning general constraints in CSP. *Artificial Intelligence*, 238:135–153, 2016.
280. W. Visser, S. Park, and J. Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In M. P. E. Heimdahl, editor, *Third Workshop on Formal Methods in Software Practic (FMSP)*, pages 3–182. ACM, 2000.
281. J. Whittemore, J. Kim, and K. Sakallah. SATIRE: A new incremental satisfiability engine. In *IEEE/ACM Design Automation Conference (DAC)*, 2001.
282. R. Wilhelm, S. Sagiv, and T. W. Reps. Shape analysis. In *Compiler Construction (CC)*, volume 1781 of *LNCS*, pages 1–17. Springer, 2000.
283. H. P. Williams. Fourier-Motzkin elimination extension to integer programming problems. *Journal of Combinatorial Theory*, 21:118–123, July 1976.
284. R. Williams, C. P. Gomes, and B. Selman. Backdoors to typical case complexity. In *IJCAI*, pages 1173–1178. Morgan Kaufmann, 2003.
285. J. Wilson. Compact normal forms in propositional logic. *Computers and Operations Research*, 90:309–314, 1990.
286. S. A. Wolfman and D. S. Weld. The LPSAT engine & its application to resource planning. In T. Dean, editor, *International Joint Conference on Artificial Intelligence (IJCAI)*, pages 310–317. Morgan Kaufmann, 1999.
287. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 1785 of *LNCS*, pages 1–19. Springer, 2000.
288. L. Wolsey. *Integer Programming*. Wiley-Interscience, 1998.

289. L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Principles and Practice of Constraint Programming (CP)*, pages 712–727, 2007.
290. B. Yordanov, S.-J. Dunn, H. Kugler, A. Smith, G. Martello, and S. Emmott. A method to identify and analyze biological programs through automated reasoning. *Nature Systems Biology and Applications, In Press*, 2016.
291. B. Yordanov, C. Wintersteiger, Y. Hamadi, and H. Kugler. SMT-based analysis of biological computation. In *NFM*, volume 7871 of *LNCS*. Springer, 2013.
292. B. Yordanov, C. Wintersteiger, Y. Hamadi, and H. Kugler. Z34Bio: An SMT-based framework for analyzing biological computation. In *SMT*, 2013.
293. B. Yordanov, C. Wintersteiger, Y. Hamadi, A. Phillips, and H. Kugler. Functional analysis of large-scale DNA strand displacement circuits. In *Proc. 19th International Conference on DNA Computing and Molecular Programming (DNA 2013)*, volume 8141 of *LNCS*, pages 189–203, 2013.
294. H. Zantema and J. F. Groote. Transforming equality logic to propositional logic. *Electronic Notes in Computer Science*, 86(1), 2003.
295. L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a Boolean satisfiability solver. In *International Conference on Computer Aided Design (ICCAD)*, pages 279–285. IEEE, 2001.
296. L. Zhang and S. Malik. Conflict driven learning in a quantified Boolean satisfiability solver. In *International Conference on Computer Aided Design (ICCAD)*, 2002.
297. L. Zhang and S. Malik. Towards symmetric treatment of conflicts and satisfaction in quantified Boolean satisfiability solver. In P. V. Hentenryck, editor, *8th International Conference on Principles and Practice of Constraint Programming (CP)*, volume 2470 of *LNCS*, pages 200–215. Springer, 2002.
298. L. Zhang and S. Malik. Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In *Theory and Applications of Satisfiability Testing (SAT)*, 2003. Presentation only.
299. L. Zhang and S. Malik. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In *Design and Test in Europe Conference (DATE)*, pages 10880–10885. IEEE, 2003.

Tools index

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

Algorithms index

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0

Index

Note that there is a separate index page for tools on page 347, and a separate index page for algorithms on page 349.

D. Kroening and O. Strichman, *Decision Procedures*,
Texts in Theoretical Computer Science. An EATCS Series,
DOI 10.1007/978-3-662-50497-0