SRI International

April 9, 2014

Yices 2 Manual

Bruno Dutertre
Computer Science Laboratory
SRI International
Menlo Park CA 94025 USA

Computer Science Laboratory • 333 Ravenswood Ave. • Menlo Park, CA 94025 • (650) 326-6200 • Facsimile: (650) 859-2844

Contents

Chapter 1

Introduction

This manual is an introduction to the logic, language, and architecture of the Yices 2 SMT solver. Yices is developed in SRI International's Computer Science Laboratory and is distributed free-of-charge for personal use, under the terms of the Yices License (reproduced in Chapter 7 of this manual). To discuss alternative license terms, please contact us at `fm-license@csl.sri.com`.

1.1 Download and Installation

The latest version of Yices 2 can be downloaded at `http://yices.csl.sri.com/download-yices2.shtml`. We provide binary distributions for the platforms and operating systems listed in Table 1.1. To download Yices 2, go to `http://yices.csl.sri.com/download-yices2.shtml` and select the distribution that you want to install. This will open a web page showing the license terms. If you agree to the terms, click on the "accept" button to download a tarfile or zip file. Untar or unzip the file and follow the instructions in the included README file.

OS/Hardware	**Notes**
Linux (32 and 64bit)	Requires Kernel 2.6.8 or more recent
Mac OS X (Intel 32 and 64bit)	For MacOS X 10.5 (Leopard) and more recent
Windows (32bits and 64bit)	Compatible with Windows XP, Vista, 7 and 8
Cygwin (32bit Intel)	
FreeBSD 9.0 (32 and 64bits)	
Solaris 2.10 (Sparc 64bits)	

Table 1.1: Supported Platforms

1.2 Content of the Distribution

The distribution includes the Yices executables, the Yices library and header files, and examples and documentation. The Yices library and header files allows you to use Yices via its API, as explained in Chapter 6.

Three solvers are currently included in the distribution:

- `yices` is the main SMT solver. It can read and process input given in Yices 2's specification language. This language is explained in Chapter 4.
- `yices-smt` is a solver for input in the SMT-LIB 1.2 notation [RT06].
- `yices-sat` is a Boolean satisfiability solver that can read input in the DIMACS CNF format.

We are developing a new solver that will support the more recent SMT-LIB 2 notation, but this solver is not complete yet, and it is not included in the distribution.

1.3 Library Dependencies

Yices uses the GNU Multiple Precision Arithmetic Library (GMP). We recommend most people to download a Yices distribution that is statically linked against GMP. However, we also provide Yices builds that are linked dynamically against GMP. To use such distributions, you have to install a compatible version of GMP on your machine. GMP-5.0.x or more recent should work. To see the exact GMP version used by Yices, check the README file. The GMP library can be installed using common package managers in most Linux distributions. It can also be built and installed from source. For more information, please visit the GMP website `http://gmplib.org`.

1.4 Supported Logics

The current Yices 2 release supports quantifier-free combinations of linear integer and real arithmetic, uninterpreted function, arrays, and bitvectors. Currently, Yices 2 supports all SMT-LIB logics that do not involve quantifiers or nonlinear arithmetic as summarized in Table 1.2. The meaning of the logics and theories in this table is explained at the SMT-LIB website (`http://www.smtlib.org`). In addition, Yices 2 support a more general set of array operations than required by SMT-LIB, and Yices 2 has support for tuple and enumeration types, which are not part of SMT-LIB.

Logic	Description	Supported
AUFLIA	Arrays, Linear Integer Arithmetic Quantifiers, Uninterpreted Functions	no
AUFLIRA	Arrays, Mixed Linear Arithmetic Quantifiers, Uninterpreted Functions	no
AUFNIRA	Arrays, Nonlinear Integer Arithmetic Quantifiers, Uninterpreted Functions	no
LIA	Linear Integer Arithmetic, Quantifiers	no
LRA	Linear Real Arithmetic, Quantifiers	no
QF_A	Arrays (without extensionality)	yes
QF_AUFBV	Arrays, Bitvectors Uninterpreted Functions	yes
QF_AUFLIA	Arrays, Linear Integer Arithmetic Uninterpreted Functions	yes
QF_AX	Arrays (with extensionality)	yes
QF_BV	Bitvectors	yes
QF_IDL	Integer Difference Logic	yes
QF_LIA	Linear Integer Arithmetic	yes
QF_LIA	Linear Real Arithmetic	yes
QF_NIA	Nonlinear Integer Arithmetic	no
QF_RDL	Real Difference Logic	yes
QF_UF	Uninterpreted Functions	yes
QF_UFIDL	Uninterpreted Functions, Integer Difference Logic	yes
QF_UFBV	Uninterpreted Functions, Bitvectors	yes
QF_UFLIA	Uninterpreted Functions, Linear Integer Arithmetic	yes
QF_UFLRA	Uninterpreted Functions, Linear Real Arithmetic	yes
QF_UFNRA	Uninterpreted Functions, Nonlinear Real Arithmetic	yes
UFNIA	Nonlinear Integer Arithmetic, Quantifiers Uninterpreted Functions	no

Table 1.2: Logics Supported by Yices 2

1.5 Getting Help and Reporting Bugs

The Yices website provides the latest release and information about Yices. For bug reports and questions about Yices, please contact us via the Yices mailing lists:

- If you have questions about Yices usage or installation, send e-mail to `yices-help@csl.sri.com`.

  This mailing list is moderated, but you do not need to register to post to it.

- To report a bug, send e-mail to `yices-bugs@csl.sri.com`.

  Please include enough information in your bug report to enable us to reproduce the problem.

Chapter 2

Yices 2 Logic

Yices 2 specifications are written in a typed logic. The language is intended to be simple enough for efficient processing by the tool and expressive enough for most applications. The Yices 2 language is similar to the logic supported by Yices 1, but the most complex type constructs have been removed.

2.1 Type System

Yices 2 has a few built-in types for primitive objects:

- The arithmetic types `int` and `real`
- The Boolean type `bool`
- The type `(bitvector k)` of bitvectors of size k, where k is a positive integer.

All these built-in types are *atomic*. The set of atomic types can be extended by declaring new *uninterpreted types* and *scalar types*. An uninterpreted type denotes a nonempty collection of objects with no cardinality constraint. A scalar type denotes a nonempty, *finite* set of objects. The cardinality of a scalar type is defined when the type is created.

In addition to the atomic types, Yices 2 provides constructors for tuple and function types. The set of all Yices 2 types can be defined inductively as follows:

- Any atomic type τ is a type.
- If $n > 0$ and $\sigma_1, \ldots, \sigma_n$ are n types, then $\sigma = (\sigma_1 \times \ldots \times \sigma_n)$ is a type. Objects of type σ are tuples $(x_1, \ldots, x_n)$ where x_i is an object of type σ_i.
- If $n > 0$ and $\sigma_1, \ldots, \sigma_n$ and τ are types, then $\sigma = (\sigma_1 \times \ldots \times \sigma_n \to \tau)$ is a type. Objects of type σ are functions of domain $\sigma_1 \times \ldots \times \sigma_n$ and range τ.

By construction, all the types are nonempty. Yices does not have a specific type constructor for arrays since the logic does not distinguish between arrays and functions. For example, an array indexed by integers is simply a function of domain `int`.

Yices 2 uses a simple form of subtyping. Given two types σ and τ, let $\sigma \sqsubset \tau$ denote that σ is a subtype of τ. Then the subtype relation is defined by the following rules:

- $\tau \sqsubset \tau$ (any type is a subtype of itself)
- $\texttt{int} \sqsubset \texttt{real}$ (the integers form a subtype of the reals)
- If $\sigma_1 \sqsubset \tau_1, \ldots, \sigma_n \sqsubset \tau_n$ then $(\sigma_1 \times \ldots \times \sigma_n) \sqsubset (\tau_1 \times \ldots \times \tau_n)$.
- If $\tau \sqsubset \tau'$ then $(\sigma_1 \times \ldots \times \sigma_n \to \tau) \sqsubset (\sigma_1 \times \ldots \times \sigma_n \to \tau')$.

For example, the type $(\texttt{int} \times \texttt{int})$ (pairs of integers) is a subtype of $(\texttt{real} \times \texttt{real})$ (pairs of reals).

Two types, τ and τ', are said to be *compatible* if they have a common supertype, that is, if there exists a type σ such that $\tau \sqsubset \sigma$ and $\tau' \sqsubset \sigma$. If that is the case, then there exists a unique minimal supertype among all the common supertypes. We denote the minimal supertype of τ and τ' by $\tau \sqcup \tau'$. By definition, we then have

$$\tau \sqsubset \sigma \ \texttt{and} \ \tau' \sqsubset \sigma \ \Rightarrow \ \tau \sqcup \tau' \ \sqsubset \ \sigma.$$

For example, the tuple types $\tau = (\texttt{int} \times \texttt{real} \times \texttt{int})$ and $\tau = (\texttt{int} \times \texttt{int} \times \texttt{real})$ are compatible. Their minimal supertype is $\tau \sqcup \tau' = (\texttt{int} \times \texttt{real} \times \texttt{real})$. The type $(\texttt{real} \times \texttt{real} \times \texttt{real})$ is also a common supertype of τ and τ' but it is not minimal.

2.2 Terms and Formulas

In Yices 2, the atomic terms include the Boolean constants (`true` and `false`) as well as arithmetic and bitvector constants.

When a scalar type τ of cardinality n is declared, n distinct constant $c_1, \ldots, c_n$ of type τ are also implicitly defined. In the Yices 2 syntax, this is done via a declaration of the form:

```
(define-type tau (scalar c1 ... cn))
```

An equivalent functionality is provided by the Yices API. The API allows one to create a new scalar type and to access n constants of that type indexed by integers between 0 and $n-1$ (check file `include/yices.h` for explanations).

The user can also declare *uninterpreted constants* of arbitrary types. Informally, uninterpreted constants of type τ can be considered like global variables, but Yices (in particular the Yices API) makes a distinction between *variables* of type τ and *uninterpreted constants* of type τ. In the Yices API, variables are used to build quantified expressions and to support term substitutions. Free variables are not allowed to occur in assertions.

The term constructors include the common Boolean operators (conjunction, disjunction, negation, implication, etc.), an if-then-else constructor, equality, function application, and tuple constructor and projection. In addition, Yices provides an `update` operator that can be applied to arbitrary functions. The type-checking rules for these primitive operators are described in Figure 2.1, where the notation $t :: \tau$ means "term t has type τ".

There are no separate syntax or constructors for formulas. In Yices 2, a formula is simply a term of Boolean type.

The semantics of most of these operators is standard. The update operator for functions is characterized by the following axioms[1]:

$$\begin{array}{rcl} ((\texttt{update}\ f\ t_1 \ldots t_n\ v)\ t_1 \ldots t_n) & = & v \\ u_1 \neq t_1 \vee \ldots \vee u_n \neq t_n \Rightarrow ((\texttt{update}\ f\ t_1 \ldots t_n\ v)\ u_1 \ldots u_n) & = & (f\ u_1 \ldots u_n) \end{array}$$

In other words, $(\texttt{update}\ f\ t_1 \ldots t_n\ v)$ is the function equal to f at all points except $(t_1, \ldots, t_n)$. Informally, if f is interpreted as an array then the update corresponds to "storing" v at position $t_1, \ldots, t_n$ in the array. Reading the content of the array is nothing other than function application: $(f\ i_1 \ldots i_n)$ is the content of the array at position $i_1, \ldots, i_n$.

The full Yices 2 language has a few more operators not described here, and it includes existential and universal quantifiers. We do not describe the type-checking rules for quantifiers here since Yices 2 does not have a solver for quantified formulas at this point.

2.3 Theories

In addition to the generic operators presented previously, the Yices language includes the standard arithmetic operators and a rich set of bitvector operators.

2.3.1 Arithmetic

Arithmetic constants are arbitrary precision integers and rationals. Although Yices uses exact arithmetic, rational constants can be written using standard floating-point notation. Internally, Yices converts floating-point input to rationals. For example, the floating-point expression $3.04e - 1$ is converted to $304/1000$.

The Yices language supports the traditional arithmetic operators (i.e., addition, subtraction, multiplication) with the exception that it does not allow division by a non constant, to avoid issues related to division by zero. For example, the expression $(x + 4y)/3$ is allowed, but $3/(x + 4y)$ is not. The arithmetic predicates are the usual comparison operators, including both strict and nonstrict inequalities.

[1]These are the main axioms of the McCarthy theory of arrays.

Boolean Operators

$$\frac{t :: \texttt{bool}}{(\texttt{not}\ t) :: \texttt{bool}} \qquad \frac{t_1 :: \texttt{bool} \quad t_2 :: \texttt{bool}}{(\texttt{implies}\ t_1\ t_2) :: \texttt{bool}}$$

$$\frac{t_1 :: \texttt{bool} \ldots t_n :: \texttt{bool}}{(\texttt{or}\ t_1 \ldots t_n) :: \texttt{bool}} \qquad \frac{t_1 :: \texttt{bool} \ldots t_n :: \texttt{bool}}{(\texttt{and}\ t_1 \ldots t_n) :: \texttt{bool}}$$

Equality

$$\frac{t_1 :: \tau_1 \quad t_2 :: \tau_2}{(t_1 = t_2) :: \texttt{bool}} \text{ provided } \tau_1 \text{ and } \tau_2 \text{ are compatible}$$

If-then-else

$$\frac{c :: \texttt{bool} \quad t_1 :: \tau_1 \quad t_2 :: \tau_2}{(\texttt{ite}\ c\ t_1\ t_2) :: \tau_1 \sqcup \tau_2} \text{ provided } \tau_1 \text{ and } \tau_2 \text{ are compatible}$$

Tuple Constructor and Projection

$$\frac{t_1 :: \tau_1 \ldots t_n :: \tau_n}{(\texttt{tuple}\ t_1 \ldots t_n) :: (\tau_1 \times \ldots \times \tau_n)} \qquad \frac{t :: (\tau_1 \times \ldots \times \tau_n)}{(\texttt{select}_i\ t) :: \tau_i}$$

Function Application

$$\frac{f :: (\tau_1 \times \ldots \times \tau_n \to \tau) \quad t_1 :: \sigma_1 \ldots t_n :: \sigma_n \quad \sigma_1 \sqsubset \tau_1 \ldots \sigma_n \sqsubset \tau_n}{(f\ t_1 \ldots t_n) :: \tau}$$

Function Update

$$\frac{f :: (\tau_1 \times \ldots \times \tau_n \to \tau) \quad t_1 :: \sigma_1 \ldots t_n :: \sigma_n \quad v :: \sigma \quad \sigma_i \sqsubset \tau_i \quad \sigma \sqsubset \tau}{(\texttt{update}\ f\ t_1 \ldots t_n\ v) :: (\tau_1 \times \ldots \times \tau_n \to \tau)}$$

Figure 2.1: Primitive Operators and Type Checking

The language allows nonlinear polynomials but this is not fully supported by the tool at this time. Yices 2 can solve problems involving real and integer linear arithmetic, but it does not yet include a solver for nonlinear arithmetic.

2.3.2 Bitvectors

Yices supports all the bitvector operators defined in the SMT-LIB standard [RT06]. The most commonly used operators are listed in Table 2.1. They include bitvector arithmetic (where bitvectors are interpreted either as unsigned integers or as signed integers in two's complement representation), logical operators such as bitwise OR or AND, logical and arithmetic shifts, concatenation, and extraction of subvectors. Other operators are defined in the theory QF_BV of SMT-LIB (cf. `http://www.smtlib.org`); all of them are supported by Yices 2.

The semantics of all the bitvector operators is defined in the SMT-LIB 1.2 standard. Yices 2 follows the standard except for the case of division by zero. In SMT-LIB, the result of a division by zero is an unspecified value, but one must ensure that the division operators are functional. In other words, SMT-LIB does not specify the result of $(\texttt{bvudiv}\ a\ b)$ if b is the zero vector, but $(\texttt{bvudiv}\ a\ b)$ and $(\texttt{bvudiv}\ c\ b)$ must be equal whenever $a = c$, even if b is the zero vector. Yices 2 uses a simpler semantics (inspired from the BTOR format [BBL08]):

- **Unsigned Division:** If b is the zero bitvector of n bits then

$$
\begin{aligned}
(\texttt{bvudiv}\ a\ b) &= \texttt{0b111...1} \\
(\texttt{bvurem}\ a\ b) &= a
\end{aligned}
$$

  In general, the quotient $(\texttt{bvudiv}\ a\ b)$ is the largest unsigned integer that can be represented on n bits, and is smaller than a/b, and the following identity holds for all bitvectors a and b

$$
a = (\texttt{bvadd}\ (\texttt{bvmul}\ (\texttt{bvudiv}\ a\ b)\ b)\ (\texttt{bvurem}\ a\ b)).
$$

- **Signed Division:** If b is the zero bitvector of n bits then

$$
\begin{aligned}
(\texttt{bvsdiv}\ a\ b) &= \texttt{0b000..01}\ \text{ if } a \text{ is negative} \\
(\texttt{bvsdiv}\ a\ b) &= \texttt{0b111...1}\ \text{ if } a \text{ is non-negative} \\
(\texttt{bvsrem}\ a\ b) &= a \\
(\texttt{bvsmod}\ a\ b) &= a
\end{aligned}
$$

Operator and Type	Meaning
`bvadd` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	addition
`bvsub` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	subtraction
`bvmul` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	multiplication
`bvneg` :: $\texttt{bv}\ n) \to (\texttt{bv}\ n))$	2's complement opposite
`bvudiv` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	quotient in unsigned division
`bvudiv` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	remainder in unsigned division
`bvsdiv` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	quotient in signed division with rounding toward zero
`bvsrem` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	remainder in signed division with rounding toward zero
`bvsmod` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	remainder in signed division with rounding toward $-\infty$
`bvule` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned less than or equal
`bvuge` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned greater than or equal
`bvult` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned less than
`bvugt` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	unsigned greater than
`bvsle` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed less than or equal
`bvsge` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed greater than or equal
`bvslt` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed less than
`bvsgt` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to \texttt{bool}$	signed greater than
`bvand` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise and
`bvor` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise or
`bvnot` :: $((\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise negation
`bvxor` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	bitwise exclusive or
`bvshl` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	shift left
`bvlshr` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	logical shift right
`bvashr` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ n) \to (\texttt{bv}\ n))$	arithmetic shift right
`bvconcat` :: $((\texttt{bv}\ n) \times (\texttt{bv}\ m) \to (\texttt{bv}\ n+m))$	concatenation
$\texttt{bvextract}_{i,j}((\texttt{bv}\ n) \to (\texttt{bv}\ m))$	extract bits i down to j form a bitvector of size n

Table 2.1: Bitvector Operators

Chapter 3

Yices 2 Architecture

Yices 2 relies on a simpler language and type system than Yices 1. We have also completely redesigned the architecture to make Yices 2 easier to maintain and develop. The new architecture supports new features, such as the possibility to maintain several contexts in parallel.

3.1 Main Components

The Yices 2 software can be conceptually decomposed into three main modules:

Term Database Yices 2 maintains a global database in which all terms and types are stored. Yices 2 provides an API for constructing terms, formulas, and types in this database.

Context Management A context is a central data structure that stores asserted formulas. Each context contains a set of assertions to be checked for satisfiability. The context-management API supports operations for creating and initializing contexts, for asserting formulas into a context, and for checking the satisfiability of the asserted formulas. Several contexts can be constructed and manipulated independently.

Contexts are highly customizable. Each context can be configured to support a specific theory, and to use a specific solver or combination of solvers.

Model Management If the set of formulas asserted in a context is satisfiable, then one can construct a model of the formulas. The model maps symbols of the formulas to concrete values (e.g., integer or rational values or bitvector constants). The API provides functions to build and query models.

Figure 3.1 shows the top-level architecture of Yices 2, divided into the three main modules. Each context consists of two separate components: The *solver* employs a Boolean satisfiability solver and decision procedures for determining whether the formulas asserted

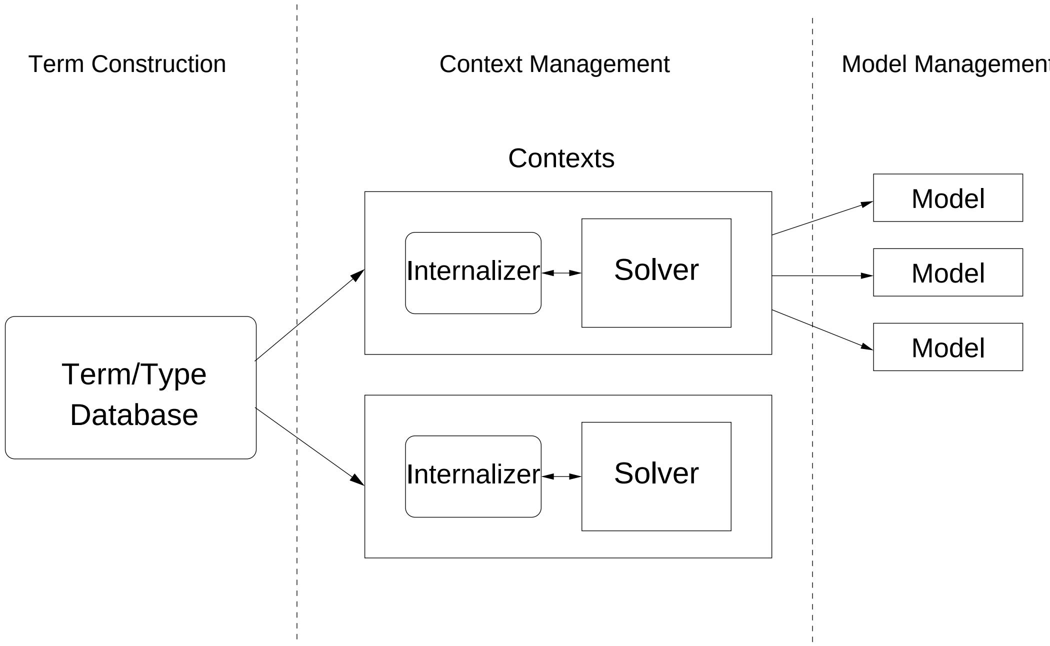

Figure 3.1: Top-level Yices 2 Architecture

in the context are satisfiable. The *internalizer* converts the format used by the term database into the internal format used by the solver. In particular, the internalizer rewrites all formulas in conjunctive normal form, which is used by the internal SAT solver.

3.2 Solvers

In Yices 2, it is possible to select a different solver (or combination of solvers) for the problem of interest. Each context can thus be configured for a specific class of formulas. For example, one can use a solver specialized for linear arithmetic, or use a solver that supports the full Yices 2 language. Figure 3.2 shows how the most general solver is built. A major component of all solvers is a SAT solver based on the Davis-Putnam-Logemann-Loveland (DPLL) procedure. The SAT solver is coupled with one or more so-called *theory solvers*. Each theory solver implements a decision procedure for a particular theory. Currently, Yices 2 includes four main theory solvers:

- The *UF Solver* deals with the theory of uninterpreted functions with equality[1]. It implements a decision procedure based on computing congruence closures, similar to the Simplify system [DNS05].

[1]UF stands for uninterpreted functions.

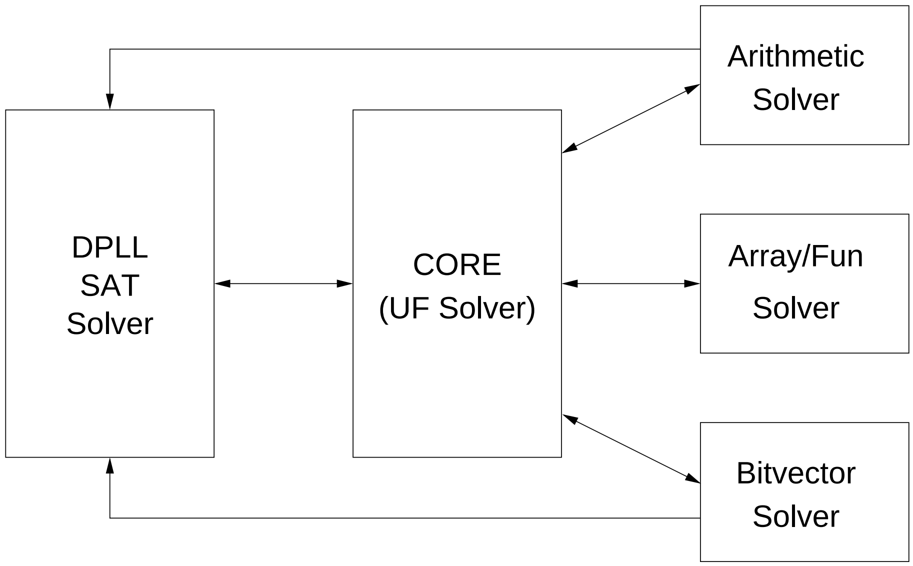

Figure 3.2: Solver Components

- The *Arithmetic Solver* deals with linear integer and real arithmetic. It implements a decision procedure based on the Simplex algorithm [DdM06a, DdM06b].
- The *Bitvector Solver* deals with the theory of bitvectors.
- The *Array Solver* implements a decision procedure for McCarthy's theory of arrays.

Yices 2 employs a modular solver architecture. It is possible to remove some of the components of Figure 3.2 to build simpler and more efficient solvers that are specialized for specific classes of formulas. For example, a solver for pure arithmetic can be built by directly attaching the arithmetic solver to the DPLL SAT solver. Similarly, Yices 2 can be specialized for pure bitvector problems, or for problems combining uninterpreted functions, arrays, and bitvectors (by removing the arithmetic solver).

Yices 2 combines several theory solver using the Nelson-Oppen method [NO79]. The UF solver is essential for this purpose; it coordinates the different theory solvers and ensures global consistency. The other solvers (for arithmetic, arrays, and bitvectors) communicate only with the central UF solver and never directly with each other. This property considerably simplifies the design and implementation of theory solvers.

Chapter 4

yices

The Yices 2 distribution includes a tool for processing input written in the Yices 2 language. This tool is called `yices` (or `yices.exe` in the Windows and Cygwin distributions). The syntax and the set of commands supported by `yices` are explained in the file `doc/YICES-LANGUAGE` included in the distribution. Several example specifications are also included in the `examples/` directory.

4.1 Example

To illustrate the tool usage, consider file `examples/bv_test2.ys` shown in Figure 4.1. The first line defines a type called `BV`. In this case, `BV` is a synonym for bitvectors of size 32. Then four terms are declared of type `BV`. The three constants `a`, `b`, and `d` are uninterpreted, while `c` is defined as the bitvector representation of the integer 1008832. The next line of the file is an assertion expressing a constraint between `a`, `b`, `c`, and `d`, using bitvector operators. The command `(check)` checks whether the assertion is satisfiable. Since it is, command `(show-model)` asks for a satisfying model to be displayed. The next commands ask for the value of four terms in the model.

To run `yices` on this input file, just type

```
yices examples/bv_test2.ys
```

The tool will output something like this:

```
sat
(= d 0b00000000000000000000000000000000)
(= b 0b00000000000000000000000000000000)
(= a 0b00000000000000000000000011000000)

0b00000000000000000000000011000000
0b00000000000000000000000000000000
0b00000000000011110110010011000000
0b00000000000000000000000000000000
```

```
(define-type BV (bitvector 32))

(define a::BV)
(define b::BV)
(define c::BV (mk-bv 32 1008832))
(define d::BV)

(assert (= a (bv-or (bv-and (mk-bv 32 255) (bv-not (bv-or b (bv-not c))))
                    (bv-and c (bv-xor d (mk-bv 32 1023))))))

(check)

(show-model)
(eval a)
(eval b)
(eval c)
(eval d)
```

Figure 4.1: Example Yices Script (from `examples/bv_test2.ys`)

The result of the `(check)` command is shown on the first line (i.e., `sat` for satisfiable). The next three lines show the model as an assignment to the three uninterpreted terms `a`, `b`, and `d`. Then, the tool displays one bitvector constant for each of the `(eval ...)` command.

Since this example contains only terms and constructs from the bitvector theory, we could specify logic `QF_BV` on the command line as follows:

```
yices --logic=QF_BV examples/bv_test2.ys
```

To get a more detailed output, give option `--verbose`:

```
yices --verbose examples/bv_test2.ys
```

4.2 Tool Invocation

Yices is invoked on an input file by typing

```
yices [option] <filename>
```

If no `<filename>` is given, `yices` will run in interactive mode and will read the standard input. The following options are supported

`--logic=<name>` Select an SMT-LIB logic.

`<name>` must either be an SMT-LIB logic name such as `QF_UFLIA` or the special name `NONE`.

Yices recognizes the logics defined at http://www.smtlib.org (as of December 2012). Option `--logic=NONE` configures `yices` for propositional logic.

By default—that is, if no logic is given—`yices` includes all the theory solvers described in Section 3.2. In this default configuration, `yices` supports linear arithmetic, bitvectors, uninterpreted functions, and arrays. If a logic is specified, `yices` uses a specialized solver or combination of solvers that is appropriate for the given logic. Some of the search parameters will also be set to values that seem to work well for this logic (based on extensive benchmarking). All the search parameters can also be modified individually using the command `(set-param ...)`.

If option `--logic=NONE` is given, then `yices` includes no theory solvers at all. All assertions must be purely propositional (i.e., involve only Boolean terms).

`--arith-solver=<solver>` Select one of the possible arithmetic solvers.

`<solver>` must be one of `simplex`, `floyd-warshall`, or `auto`.

If the logic is `QF_IDL` (integer difference logic) or `QF_RDL` (real difference logic), then this option can be used to select the arithmetic solver: either the generic Simplex-based solver or a specialized solver based on the Floyd-Warshall algorithm. If option `--arith-solver=auto` is given, then the arithmetic solver is determined automatically; the default is `auto`.

This option has no effect for logics other than `QF_IDL` or `QF_RDL`.

`--mode=<mode>` Select solver features.

`<mode>` must be one of `one-shot`, `multi-checks`, `push-pop`, or `interactive`.

This option selects the set of functionalities supported by the solver as follows:

- `one-shot`: no assertions are allowed after `(check)`, so only one call to `(check)` is possible.
- `multi-checks`: several calls to `(assert)` and `(check)` are allowed.
- `push-pop`: like `multi-checks` but with support for adding and retracting assertions via the commands `(push)` and `(pop)`.
- `interactive`: supports the same features as `push-pop` mode, but with a different behavior when `(check)` is interrupted.

In the first two modes, `yices` employs more aggressive simplifications when processing assertions; this can lead to better performance on some problems.

In interactive mode, the solver context is saved before every call to `(check)` and it is restored if `(check)` is interrupted. This introduces some overhead, but the solver recovers gracefully if `(check)` is interrupted or times out. In the non-interactive modes, the solver exits after the first interruption or timeout.

The default mode is `push-pop` if the solver is run with an input file, or `interactive` if no input file is given.

Mode `one-shot` is required if the Floyd-Warshall solvers are used.

`--version, -V` Display version information then exit.

This displays the Yices version number, the GMP version linked with Yices, and information about build date and platform. For example, here is the output for Yices 2.1.0 built for MacOS X

```
Yices 2.1.0. Copyright SRI International.
GMP 5.0.4. Copyright Free Software Foundation.
Build date: Fri Oct 5 10:32:43 PDT 2012
Platform: x86_64-apple-darwin10.8.0 (release/static)
```

If you ever have to report a bug, please include this version information in your bug report.

`--help, -h` Print a summary of options

`--verbose, -v` Run in verbose mode

4.3 Input Language

The syntax of the Yices input language is summarized in Figures 4.2, 4.3, and 4.4.

4.3.1 Lexical Elements

Comments

Input files may contain comments, which start with a semi-colon **`';'`** and extend to the end of the line.

Strings

Strings are similar to strings in C. They are delimited by double quotes `"` and may contain escaped characters:

- The characters `\n` and `\t` are replaced by newline and tab, respectively.
- The character \ followed by at most three octal digits (i.e., from `0` to `7`) is replaced by the character whose ASCII code is the octal number.
- In all other cases, `\<char>` is replaced by `<char>` (including if `<char>` is a newline or \).
- A newline cannot occur inside the string, unless preceded by \.

```
<command>  ::=
           ( define-type <symbol> )
         | ( define-type <symbol> <typedef> )
         | ( define <symbol> :: <type> )
         | ( define <symbol> :: <type> <expression> )
         | ( assert <expression> )
         | ( exit )
         | ( check )
         | ( push )
         | ( pop )
         | ( reset )
         | ( show-model )
         | ( eval <expression> )
         | ( echo <string> )
         | ( include <string> )
         | ( set-param <symbol> <immediate-value> )
         | ( show-param <symbol> )
         | ( show-params )
         | ( show-stats )
         | ( reset-stats )
         | ( set-timeout <number> )
         | ( dump-context )
         | ( help )
         | ( help <symbol> )
         | ( help <string> )
         | EOS

<immediate-value> ::=
           true
         | false
         | <number>
         | <symbol>

<number> ::=
           <rational>
         | <float>
```

Figure 4.2: Yices Syntax: Commands

```
<typedef> ::=
            <type>
          | ( scalar <symbol> ... <symbol> )

<type> ::=
            <symbol>
          | ( tuple <type> ... <type> )
          | ( -> <type> ... <type> <type> )
          | ( bitvector <rational> )
          | int
          | bool
          | real
```

Figure 4.3: Yices Syntax: Types

```
<expr> ::=
            true
          | false
          | <symbol>
          | <rational>
          | <float>
          | <binary bv>
          | <hexa bv>
          | ( forall ( <var_decl> ... <var_decl> ) <expr> )
          | ( exists ( <var_decl> ... <var_decl> ) <expr> )
          | ( lambda ( <var_decl> ... <var_decl> ) <expr> )
          | ( let ( <binding> ... <binding> ) <expr> )
          | ( update <expr> ( <expr> ... <expr> ) <expr> )
          | ( <function> <expr> ... <expr> )

<function> ::=
            <function-keyword>
          | <expr>

<var_decl> ::= <symbol> :: <type>

<binding> ::= ( <symbol> <expr> )
```

Figure 4.4: Yices Syntax: Expressions

Numerical Constants

Numerical constants can be written as decimal integers (e.g., 44 or -3), rational (e.g., -1/3), or using a floating-point notation (e.g., 0.07 or -1.2e+2). Positive constants can start with an optional + sign. For example +4 and 4 denote the same number.

Bitvector Constants

Bitvector constants can be written in a binary format using the prefix 0b or in hexadecimal using the prefix 0x. For example, the expressions 0b01010101 and 0x55 denote the same bitvector constant of eight bits.

Symbols

A symbol is any character string that's not a keyword (see Table 4.1) and doesn't start with a digit, a space, or one of the characters (,), ;, :, and ". If the first character is + or -, then it must not be followed by a digit. Symbols end by a space, or by any of the characters (,), ;, :, or ". Here are some examples:

```
a_symbol __another_one  X123  &&&  +z203  t\12
```

All the predefined keywords and symbols are listed in Table 4.1.

4.3.2 Declarations

Type Declaration

A type declaration is a command of the following two forms.

```
(detine-type <name>)
(define-type <name> <type>)
```

The fist form creates a new uninterpreted type called <name>. The second form gives a <name> to an existing <type>. After this definition, every occurrence of <name> refers to <type>. A variant of this second form is used to define scalar types. In these two commands, <name> must be a symbol that's not already used as a type name.

Term Declaration

A term is declared using one for the following two commands.

```
(define <name> :: <type>)
(define <name> :: <type> <term>)
```

The first form declares a new uninterpreted term of the given <type>. The second form assigns a <name> to the given <term>, which must be of type <type>. The <name> must be a symbol that's not already used as a term name.

Yices uses different name spaces for types and terms. It is then permitted to use the same name for a type and for a term.

*	+	-	->
/	/=	<	<=
<=>	=	=>	>
>=	^	and	assert
bitvector	bool	bv-add	bv-and
bv-ashift-right	bv-ashr	bv-comp	bv-concat
bv-div	bv-extract	bv-ge	bv-gt
bv-le	bv-lshr	bv-lt	bv-mul
bv-nand	bv-neg	bv-nor	bv-not
bv-or	bv-pow	bv-redand	bv-redor
bv-rem	bv-repeat	bv-rotate-left	bv-rotate-right
bv-sdiv	bv-sge	bv-sgt	bv-shift-left0
bv-shift-left1	bv-shift-right0	bv-shift-right1	bv-shl
bv-sign-extend	bv-sle	bv-slt	bv-smod
bv-srem	bv-sub	bv-xnor	bv-xor
bv-zero-extend	check	define	define-type
distinct	dump-context	dump-context	echo
eval	exists	exit	false
forall	help	if	include
int	ite	lambda	let
mk-bv	mk-tuple	not	or
pop	push	real	reset
reset-stats	scalar	select	set-param
set-timeout	show-model	show-param	show-params
show-stats	true	tuple	tuple-update
update	xor		

Table 4.1: Keywords and predefined symbols

4.3.3 Types

Predefined Types

The predefined types are `bool`, `int`, `real`, and `(bitvector k)` where k is a positive integer. For example a bit-vector variable `b` of 32 bits is declared using the command

```
(define b::(bitvector 32))
```

The number of bits must be positive so `(bitvector 0)` is not a valid type. There is also a hard-coded limit on the size of bitvectors that Yices can handle (namely, $2^{28} - 1$).

Uninterpreted Types

A new uninterpreted type T can be introduced using the command

```
(define-type T)
```

This command will succeed provided `T` is a fresh type name, that is, if there is no existing type called `T`. As explained in Section 2.1, an uninterpreted type denotes a nonempty collection of objects. There is no cardinality constraint on `T`, except that `T` is not empty.

Scalar Type

A scalar type is defined by enumerating its elements. For example, the following declaration

```
(define-type P (scalar A B C))
```

defines a new scalar type called `P` that contains the three distinct constants `A`, `B`, and `C`. Such a declaration is valid provided `P` is a fresh type name and `A`, `B`, and `C` are all fresh term names.

The enumeration must include at least one element, but singleton types are allowed. For example, the following declaration is valid.

```
(define-type Unit (scalar One))
```

It introduces a new type `Unit` of cardinality one, and which contains `One` as its unique element. Thus, any term of type `Unit` is known to be equal to `One`.

Tuple Types

A tuple type is written (tuple <tau_1> ... <tau_n>) where <tau_i> is a type. For example, the type of pairs of integer can be declared as follows:

```
(define-type Pairs (tuple int int))
```

Then one can declare an uninterpreted constant `x` of this type as follows

```
(define x::Pairs)
```

This is equivalent to the declaration

```
(define x::(tuple int int))
```

Tuple types with a single component are allowed. For example, the following declaration is legal.

```
(define-type T (tuple bool))
```

Function Types

A function type is written (-> <tau_1> ... <tau_n> <sigma>), where <tau_i> and <sigma> are types. The types <tau_1>, ..., <tau_n> define the domain of the function type, and <sigma> is the range. For example, a function defined over the integers and that returns a Boolean can be declared as follows:

```
(define f::(-> int bool))
```

Yices does not have a distinct type construct for arrays. In Yices, arrays are the same as functions.

4.3.4 Terms

Yices uses a lisp-like syntax for terms and formulas (i.e., Boolean terms). Here are examples of Boolean terms:

```
(distinct x y z t u)
(= x y)
(/= x y)
(< (* 2 x) -1)
(and (P (f a) b)
```

and has support for all common Boolean and arithmetic operations. Yices also supports all the bitvector operators

4.3.5 Commands

Chapter 5

yices-smt

Another tool included in the distribution can process input written in the SMT-LIB notation. This tool is called `yices-smt` (or `yices-smt.exe`). It is included in the `bin` directory. Currently, this tool supports version 1.2 of SMT-LIB. Support for the more recent SMT-LIB 2 will be provided in future releases.

Chapter 6

Yices API

The distribution includes a library and header files for embedding Yices in other software. The main header file is `yices.h` which includes all the API. The API functions are documented in this header file. More complete and detailed documentation on the Yices 2 API will be provided at the Yices website `http://yices.csl.sri.com/`.

Chapter 7

Yices License Terms

Before downloading and using Yices, you will be asked to agree to the Yices license terms reproduced below. SRI is open to distributing Yices under other agreements. Contact us at `fm-licensing@csl.sri.com` to discuss alternative license terms.

```
END-USER LICENSE AGREEMENT

IMPORTANT - READ CAREFULLY.  Be  sure to carefully read and understand
all of the rights and  restrictions described in this End-User License
Agreement ("EULA").  You will be  asked to review and either accept or
not accept the terms of the EULA.  You will not be permitted to access
or use the Software unless or  until you accept the terms of the EULA.
Alternative  license  terms may  be  available  to  you by  contacting
fm-licensing@csl.sri.com.

This EULA is a legal agreement  between you (either an individual or a
single entity) and SRI International ("SRI") for the software referred
to by SRI as "Yices",  which includes the computer software accessible
via  this web  browser interface,  and may  include  associated media,
printed  materials  and   any  "online"  or  electronic  documentation
("Software").  By utilizing the Software, you agree to be bound by the
terms of this  EULA.  If you do  not agree to the terms  of this EULA,
you may not access or use the Software.

GRANT  OF LIMITED  LICENSE.   SRI  hereby grants  to  you a  personal,
non-exclusive,  non-transferable, royalty-free  license to  access and
use  the Software  for your  own internal  purposes.  The  Software is
licensed to  you, and such license  does not constitute a  sale of the
Software.   SRI  reserves the  right  to  release  the Software  under
different license terms or to stop distributing or providing access to
the Software at any time.

RESTRICTIONS.  You may not:  (i) distribute, sublicense, rent or lease
the  Software;  (ii)  modify,  adapt,  translate,  reverse  engineer,
decompile,  disassemble  or  create  derivative  works  based  on  the
```

Software; or (iii) create more than one (1) copy of the Software or any related documentation.

OWNERSHIP. SRI is the sole owner of the Software. You agree that SRI retains title to and ownership of the Software and that you will keep confidential and use your best efforts to prevent and protect the Software from unauthorized access, use or disclosure. All trademarks, service marks, and trade names are proprietary to SRI. All rights not expressly granted herein are hereby reserved.

TERMINATION. The EULA is effective upon the date you first use the Software and shall continue until terminated as specified below. You may terminate the EULA at any time prior to the natural expiration date by destroying the Software and any and all related documentation and copies and installations thereof, whether made under the terms of these terms or otherwise. SRI may terminate the EULA if you fail to comply with any condition of the EULA or at SRI's discretion for good cause. Upon termination, you must destroy the Software in your possession, if any, and any and all copies thereof. In the event of termination for any reason, the provisions set forth under the paragraphs entitled DISCLAIMER OF ALL WARRANTIES, EXCLUSION OF ALL DAMAGES, and LIMITATION AND RELEASE OF LIABILITY shall survive.

U.S. GOVERNMENT RESTRICTED RIGHTS. The Software is deemed to be "commercial software" and "commercial computer software documentation", respectively, pursuant to DFARS 227.7202 and FAR 12.212, as applicable. Any use, modification, reproduction, release, performance, display, or disclosure of the Software by the U.S. Government or any of its agencies or by a U.S. Government prime contractor or subcontractor (at any tier) shall have only "Restricted Rights", shall be governed solely by the terms of this EULA, and shall be prohibited except to the extent expressly permitted by the terms of this EULA.

DISCLAIMER OF ALL WARRANTIES. SRI PROVIDES THE SOFTWARE "AS IS" AND WITH ALL FAULTS, AND HEREBY DISCLAIMS ALL OTHER WARRANTIES AND CONDITIONS, EITHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY (IF ANY) IMPLIED WARRANTIES OR CONDITIONS OF MERCHANTABILITY, OF FITNESS FOR A PARTICULAR PURPOSE, OF LACK OF VIRUSES AND OF LACK OF NEGLIGENCE OR LACK OF WORKMANLIKE EFFORT. ALSO, THERE IS NO WARRANTY OR CONDITION OF TITLE, OF QUIET ENJOYMENT OR OF NON-INFRINGEMENT. THE ENTIRE RISK ARISING OUT OF THE USE OR PERFORMANCE OF THE SOFTWARE IS WITH YOU.

EXCLUSION OF ALL DAMAGES. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL SRI BE LIABLE FOR ANY CONSEQUENTIAL, INCIDENTAL, DIRECT, INDIRECT, SPECIAL, PUNITIVE OR OTHER DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR ANY INJURY TO PERSON OR PROPERTY, DAMAGES FOR LOSS OF PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, FOR LOSS OF PRIVACY FOR

FAILURE TO MEET ANY DUTY INCLUDING OF GOOD FAITH OR OF REASONABLE CARE, FOR NEGLIGENCE AND FOR ANY PECUNIARY OR OTHER LOSS WHATSOEVER) ARISING OUT OF OR IN ANY WAY RELATED TO THE USE OF OR INABILITY TO USE THE SOFTWARE, EVEN IF SRI HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. THIS EXCLUSION OF DAMAGES SHALL BE EFFECTIVE EVEN IF ANY REMEDY FAILS OF ITS ESSENTIAL PURPOSE.

LIMITATION AND RELEASE OF LIABILITY. SRI has included in this EULA terms that disclaim all warranties and liability for the Software. To the full extent allowed by law, YOU HEREBY RELEASE SRI FROM ANY AND ALL LIABILITY ARISING FROM OR RELATED TO ALL CLAIMS CONCERNING THE SOFTWARE OR ITS USE. If you do not wish to accept access to the Software under the terms of this EULA, do not access or use the Software. No refund will be made because the SOFTWARE was provided to you at no charge. Independent of, severable from, and to be enforced independently of any other provision of this EULA, UNDER NO CIRCUMSTANCE SHALL SRI'S aggregate LIABILITY TO YOU (INCLUDING LIABILITY TO ANY THIRD PERSON OR PERSONS WHOSE CLAIM OR CLAIMS ARE BASED ON OR DERIVED FROM A RIGHT OR RIGHTS CLAIMED BY YOU), WITH RESPECT TO ANY AND ALL CLAIMS AT ANY AND ALL TIMES ARISING FROM OR RELATED TO THE SUBJECT MATTER OF THIS EULA, IN CONTRACT, TORT, OR OTHERWISE, EXCEED THE TOTAL AMOUNT ACTUALLY PAID BY YOU to SRI pursuant to THIS EULA, IF ANY.

JURISDICTIONAL ISSUES. This Software is controlled by SRI from its offices within the State of California. SRI makes no representation that the Software is appropriate or available for use in other locations. Those who choose to access this Software from other locations do so at their own initiative and are responsible for compliance with local laws, if and to the extent local laws are applicable. You hereby acknowledge that the rights and obligations of the EULA are subject to the laws and regulations of the United States relating to the export of products and technical information. Without limitation, you shall comply with all such laws and regulations, including the restriction that the Software may not be accessed from, used or otherwise exported or reexported (i) into (or to a national or resident of) any country to which the U.S. has embargoed goods; or (ii) to anyone on the U.S. Treasury Department's list of Specialty Designated Nationals or the U.S. Commerce Department's Table of Deny Orders. By accessing or using the Software, you represent and warrant that you are not located in, under the control of, or a national or resident of any such country on any such list.

Notice and Procedure for Making Claims of Copyright Infringement. Pursuant to Title 17, United States Code, Section 512(c)(2), notifications of claimed copyright infringement should be sent to SRI International, Office of the General Counsel, 333 Ravenswood Ave., Menlo Park, CA 94025.

SUPPORT, UPDATES AND NEW RELEASES. The EULA does not grant you any rights to any software support, enhancements or updates. Any updates or new releases of the Software which SRI chooses at its own discretion to distribute or provide access to shall be subject to the terms hereof.

GENERAL INFORMATION. The EULA constitutes the entire agreement between you and SRI and governs your access to and use of the Software. The EULA shall not be modified except in writing by both parties.

The EULA shall be governed by and construed in accordance with the laws of the State of California, without regard to the conflicts of law principles thereof. The parties shall resolve any disputes arising out of this EULA, including disputes about the scope of this arbitration provision, by final and binding arbitration seated and held in San Francisco, California before a single arbitrator. JAMS shall administer the arbitration under its comprehensive arbitration rules and procedures. The arbitrator shall aware the prevailing party its reasonable attorney's fees and expenses, and its arbitration fees and associated costs. Any court of competent jurisdiction may enter judgment on the award.

If any provision of the EULA shall be deemed unlawful, void, or for any reason unenforceable, then that provision shall be deemed severable from these terms and shall not affect the validity and enforceability of any remaining provisions.

In consideration of your use of the Software, you represent that you are of legal age to form a binding contract and are not a person barred from receiving services under the laws of the United States or other applicable jurisdiction.

The failure of SRI to exercise or enforce any right or provision of the EULA shall not constitute a waiver of such right or provision.

Bibliography

[BBL08] R. Brummayer, A. Biere, and F. Lonsing. BTOR: Bit-Precise Modelling of Word-Level Problems for Model Checking. In *First International Workshop on Bit-Precise Reasoning*, pages 53–64, 2008. Available at `http://fmv.jku.at/BrummayerBiereLonsing-BPR08.pdf`.

[DdM06a] Bruno Dutertre and Leonardo de Moura. A fast linear-arithmetic solver for DPLL(T). In *Computer-Aided Verification (CAV'2006)*, volume 4144 of *Lecture Notes in Computer Science*, pages 81–94. Springer Verlag, August 2006.

[DdM06b] Bruno Dutertre and Leonardo de Moura. Integrating Simplex with DPLL(T). Technical Report SRI-CSL-06-01, Computer Science Laboratory, SRI International, May 2006. Available at `http://yices.csl.sri.com/sri-csl-06-01.pdf`.

[DNS05] 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.

[NO79] G. Nelson and D. C. Oppen. Simplification by Cooperating Decision Procedures. *ACM Transactions on Programming Languages and Systems*, 1(2):245–257, 1979.

[RT06] Silvio Ranise and Cesare Tinelli. The SMT-LIB Standard: Version 1.2. Technical report, SMT-LIB Initiative, 2006. Available at `http://www.smtlib.org`.