

**Java and Web Extensions of the Yices Little
Engine of Proof**

Sokharith Sok

Supervisor: Dr. Christelle Scharff

12/14/2006

Abstract

Automated Deduction, also known as Theorem Proving, is the study of programs that prove theorems. The domain evolved into two main trends: uniform proof search procedures which are guided by heuristics and based on the John Alan Robinson's resolution method, and combination of domain specific decision procedures called little engines of proof introduced by Hao Wang. Yices is a Satisfiability Modulo Theories little engine developed at Stanford Research Institute (SRI) capable of handling theories including uninterpreted functions, integer and real arithmetics, lambda expressions, arrays, bitvectors, tuples, records and recursive datatypes (e.g. lists). Yices is the 2006 winner of the Satisfiability Modulo Theories Competition (SMT-COMP), a competition that benchmarks the state-of-the-art little engines.

Based on the needs expressed by the developers of Yices at SRI, for this Software Development and Engineering Master Thesis, we developed documentation and material to make Yices more accessible and encourage its use and adoption. We developed applicative examples that are useful to anyone learning Yices. We designed a Java API that permits calls to Yices within a Java program. We built a Web extension for Yices that permits us to use it interactively from a browser. With the set of tools that we have developed, we hope to introduce Yices to members in the Computer Science and Software Engineering community who are interested in formal verification of Java programs.

All the work that we produced for this project is available at:
<http://atlantis.seidenberg.pace.edu/wiki/lep>.

Contents

Contents	1
List of Tables	3
List of Figures	4
1 Automated Deduction and Theorem Proving	5
1.1 General Context	5
1.2 Automated Deduction and Theorem Proving	5
1.3 Uniform Engines versus Little Engines	6
1.4 Organization of the Report	6
1.5 Contributions	7
2 Little Engines of Proof	8
2.1 Definitions	8
2.1.1 Signature, Terms, Equalities	8
2.1.2 Propositional Logic	9
2.1.3 First-order Predicate Logic	9
2.2 Theories	9
2.2.1 Uninterpreted Function Symbols	9
2.2.2 Integers	9
2.2.3 Reals	10
2.2.4 Arrays	10
2.2.5 Lists	11
2.2.6 Bitvectors	11
2.3 SMT-LIB	11
2.4 Existing Little Engine of Proof Software	12
2.5 SMT-COMP	13
3 Yices	15
3.1 General Description	15
3.2 Important Features	16
3.2.1 Inputs	16
3.2.2 Types	17
3.2.3 Outputs	18
3.2.4 Logical Context	18
3.3 Examples	19
3.3.1 Integer Linear Arithmetic	19

3.3.2	Real Linear Arithmetic	19
3.3.3	Bitvectors	20
3.3.4	Arrays	20
3.3.5	Lists	21
3.3.6	Program Verification	22
3.3.7	The Graph Coloring Problem	23
3.4	Java Lite API	27
3.4.1	Methods	27
3.4.2	Example	28
3.4.3	Implementation Details	29
3.4.4	Implementation Difficulties	30
3.5	Web Based Extension	31
3.5.1	Example	32
3.5.2	Implementation Details	33
3.5.3	Implementation Difficulties	34
4	Conclusion and Future Work	35
	Bibliography	36

List of Tables

2.1	SMT-COMP 2006 - Scores (1)	13
2.2	SMT-COMP 2006 - Scores (2)	14
2.3	SMT-COMP 2006 - Scores (3)	14
2.4	SMT-COMP 2006 - Scoring Rules	14

List of Figures

3.1	Yices Architecture (Source: http://yices.csl.sri.com/)	16
3.2	Coloring Problem Graph	23
3.3	Yices Java Lite API Architecture	30
3.4	Screenshot of Yices Web Interface	32
3.5	Yices Web Extension Architecture	33

Chapter 1

Automated Deduction and Theorem Proving

Contents

1.1	General Context	5
1.2	Automated Deduction and Theorem Proving . .	5
1.3	Uniform Engines versus Little Engines	6
1.4	Organization of the Report	6
1.5	Contributions	7

1.1 General Context

An important research area in programming language technology today is the improvement of the ability to discover and ascertain properties of programs and, a fortiori, to prove the correctness of a program (i.e. to prove that program does what we want it to do). Testing can validate software but there is no guarantee. Formally verifying software is more reliable. The use of formal verification is more than justified in general but also particularly for critical systems (e.g. air traffic control information systems, railways signaling systems, spacecraft systems and medical control systems) where failure must be avoided (e.g. physical damage or threats in human life, and economic losses). Proving properties or correctness is a difficult problem (undecidable in general). It takes a great deal of time and expertise, and requires specialized tools such as theorem provers and deduction systems.

1.2 Automated Deduction and Theorem Proving

Automated Deduction also goes by the name of Theorem Proving [14, 15]. It is the study of programs that prove theorems; these programs are called theorem provers. This research area is sometimes called automated reasoning because

provers are using a form of reasoning guided by the logical inference rules and strategies on the application of these rules [4].

The difficulty of the research in automated deduction relies on the fact that most problems are *undecidable* (i.e. no algorithm exists to solve them). For example, the problem known as the *word problem* [12] of verifying that an equality is true if a set of equalities is true is undecidable. The word problem for ground equalities is decidable [1]. Efficient algorithms have been developed only recently [2].

Two important properties concerning theorem provers are *soundness* and *completeness*. The soundness property is required. This property guarantees that a prover does not prove that *true = false* and that it only proves true formulae. The completeness property is optional. This property states that if a formula is true then it can be proved. The answer that all Robbins algebras are Boolean was proved by an incomplete theorem prover (EQP) and is considered as one of the major accomplishments in automated deduction [13, 16].

1.3 Uniform Engines versus Little Engines

Program verification theorem provers and deduction systems must be able to reason proficiently on types such as integers and reals, and constructs such as equalities, arrays and lists since almost all programs use them. Equalities are of particular interest because proofs require reasoning about equalities. These domain-specific provers are called *little engines of proof*.

The domain of automated reasoning evolved into 2 main categories:

- uniform proof search procedures (*big engines of proof*) which are guided by heuristics and based on the Alan Robinson's resolution method [20], and
- combination of domain/theory-specific decision procedures¹ (*little engines of proof* [23]) introduced by Hao Wang [24].

The resolution uniform proof search method has been very popular for more than 40 years after its first introduction by Robinson in 1965, mainly because the method can be easily implemented. In recent years, some examples of little engines of proof targeting particular theories and their combinations have been developed [21]. The most popular solvers are Barcelogic [19], CVC lite [3], haRVey [8], ICS [10], MathSAT [5], and Yices [9]. Little Engines of proof are in general integrated in uniform engines. For example, Yices is the backend of the SAL model checker [7].

1.4 Organization of the Report

This report is organized in the following way:

- Chapter 2 provides some definitions useful for the comprehension of this report and background information concerning the area of little engines

¹A decision problem is a question in some formal system (theory) with a yes-or-no answer. A decision procedure is a method used to solve a decision problem (http://en.wikipedia.org/wiki/Decision_problem).

of proof for theorem proving. Theories of interest including uninterpreted function symbols, integers, reals, arrays, lists and bitvectors are described. The theories with problems for benchmarking are gathered in the Satisfiability Modulo Theories Library (SMT-LIB), a standard library developed by the little engine of proof community. A list of solvers/provers is then provided as well as the results of the Satisfiability Modulo Theories Competition (SMT-COMP), a competition that ranks the most popular solvers.

- Chapter 3 describes the Yices little engine of proof. Yices is a SMT solver/prover developed at Stanford Research Institute (SRI) capable of handling theories including uninterpreted functions, integer and real arithmetics, lambda expressions, (extensional) arrays, bitvectors, tuples, records and recursive datatypes (e.g. lists). In this chapter our contributions concerning Yices are described.
- We conclude in chapter 4.

1.5 Contributions

The contributions of this Software Engineering and Design master thesis are:

- the study of the Yices little engine of proof and the development of some Yices examples that will be useful to people interested in learning Yices;
- the development of a JAVA API for Yices that permits calls to Yices within a Java program; and
- the development of a Web extension for Yices that permits to use it interactively from the Web.

The examples, the JAVA API and the Web extension of Yices are accessible at: <http://atlantis.seidenberg.pace.edu/wiki/lep>.

The work described in this master thesis is funded by an NSF ITR grant (NSF #0326540) entitled 'Little Engines of Proof'. The contributions described above are interesting for the Yices community and were developed with the support of Dr. Natarajan Shankar (SRI) and Dr. Leonardo de Moura (Microsoft Research) whose objectives are to facilitate access and promote the use and adoption of Yices.

Chapter 2

Little Engines of Proof

Contents

2.1	Definitions	8
2.1.1	Signature, Terms, Equalities	8
2.1.2	Propositional Logic	9
2.1.3	First-order Predicate Logic	9
2.2	Theories	9
2.2.1	Uninterpreted Function Symbols	9
2.2.2	Integers	9
2.2.3	Reals	10
2.2.4	Arrays	10
2.2.5	Lists	11
2.2.6	Bitvectors	11
2.3	SMT-LIB	11
2.4	Existing Little Engine of Proof Software	12
2.5	SMT-COMP	13

In this chapter we provide definitions and background useful to the comprehension of the work presented in this report. We describe the theories of interest to the little engine of proof community, the most popular little engines of proof software and their results at the Satisfiability Modulo Theories Competition (SMT-COMP).

2.1 Definitions

2.1.1 Signature, Terms, Equalities

\mathcal{F} is a countable set of function symbols. \mathcal{X} is a set of variables. *Arity* is a function from \mathcal{F} to \mathbb{N} that defines the number of parameters of a function symbol.

Definition A *signature* Σ defines a set of function symbols and their arity.

Definition The set of *terms* is $\mathcal{T} = \mathcal{T}(\mathcal{F}, \mathcal{X})$ such that:

- $\mathcal{X} \subseteq \mathcal{T}$,
- $\forall f \in \mathcal{F}$, such that $\text{arity}(f) = n$, $f(t_1, \dots, t_n) \in \mathcal{T}$ if $t_1, \dots, t_n \in \mathcal{T}$.

Definition Terms without variables are called *ground*.

Definition An equality is $l = r$ such that l and r are terms.

2.1.2 Propositional Logic

Proposition logic basic units are *propositions* that are evaluated to true or false. The logical connectives used on propositions are *not* (\neg), *and* (\wedge), *or* (\vee) and *implies* (\rightarrow).

Example $p \wedge \neg q$ is a proposition.

Propositional logic is decidable. For example, one can determine whether a proposition is a tautology using a truth table.

Definition A formula is in *conjunctive normal form* (CNF) if it is a conjunction of *clauses*, where a clause is a disjunction of literals.

Example $(p) \wedge (\neg q \vee r)$ and $(p \wedge r)$ are CNFs. $p \vee q$ is not a CNF.

2.1.3 First-order Predicate Logic

First-order predicate logic extends propositional logic with quantification (\forall , \exists). The basic units are *predicates* (propositions with parameters).

Example $Mortal(x)$ is a predicate.

First order predicate logic is undecidable. This was proved by Gödel in 1931 and is known as the Gödel's incompleteness theorem.

2.2 Theories

2.2.1 Uninterpreted Function Symbols

Uninterpreted function symbols are symbols that do not have any meaning. For example in the equality $g(a) = b$, g denote a function symbol of arity 1 and a and b denote constants.

2.2.2 Integers

Integers deal with positive and negative natural numbers (including 0). The addition, subtraction, multiplication and unary subtraction operators and the comparison operators ($<$, $>$, \leq , \geq) are part of the theory. Properties such as associativity and commutativity of the addition operator are defined. Properties such as the transitivity of \leq are also defined. 0 is the identify element for addition and 1 is the identify element for multiplication.

The Integers are described in the following way in the Satisfiability Modulo Theories Library (SMT-LIB) at <http://combination.cs.uiowa.edu/smtlib>.

```

attribute | possible value | meaning
-----
:assoc      //          the symbol is associative
:comm       //          the symbol is commutative
:unit       a constant c  c is the symbol's left and right unit
:trans      //          the symbol is transitive
:refl       //          the symbol is reflexive
:irref      //          the symbol is irreflexive
:antisym    //          the symbol is antisymmetric
"

:funs ((0 Int)
      (1 Int)
      (~ Int Int)          ; unary minus
      (- Int Int Int :assoc) ; binary minus
      (+ Int Int Int :assoc :comm :unit {0})
      (* Int Int Int :assoc :comm :unit {1})
      )

:preds ((<= Int Int :refl :trans :antisym)
        (< Int Int :trans :irref)
        (>= Int Int :refl :trans :antisym)
        (> Int Int :trans :irref)
        )

```

2.2.3 Reals

Reals deals with positive and negative real numbers (including 0).

2.2.4 Arrays

An array is a sequence of elements of the same type. Each element is accessed by its position called *index*. An array can have one or multiple dimensions. The most common operations on arrays are: *selection* (*read*) and *store* (*write*).

Arrays without Extensionality

$$\begin{aligned} \text{select}(\text{store}(A, I, E), I) &= E \\ I \neq J &\rightarrow \text{select}(\text{store}(A, I, E), J) = \text{select}(A, J) \end{aligned}$$

Arrays with Extensionality

This theory extends the theory of arrays with an axiom stating that any two arrays with the same elements are in fact the same array.

$$\begin{aligned} \text{select}(\text{store}(A, I, E), I) &= E \\ I \neq J &\rightarrow \text{select}(\text{store}(A, I, E), J) = \text{select}(A, J) \\ \forall I, \text{select}(A, I) = \text{select}(B, I) &\rightarrow A = B \end{aligned}$$

2.2.5 Lists

List are ordered collections of entities. The constructors of lists are the empty list and the *cons* operator that adds an element at the beginning of a list. The most common operators on lists are: *car* (head of a list) and *cdr* (tail of a list).

The theory of Lists *à la* Shostak is defined by:

$$\begin{aligned} \text{car}(\text{cons}(X, Y)) &\approx X \\ \text{cdr}(\text{cons}(X, Y)) &\approx Y \\ \text{cons}(\text{car}(X), \text{cdr}(X)) &\approx X. \end{aligned}$$

2.2.6 Bitvectors

Bitvectors are arrays of bits (0 or 1). The common operations on bitvectors are: *concatenation*, *extraction*, *not*, *and*, *or*, *xor*, *left-shift* and *right-shift*.

2.3 SMT-LIB

Satisfiability Modulo Theories Library (SMT-LIB) is an initiative of the SMT community to establish a library of benchmark for SMT which will facilitate the assessment and comparison of existing SMT solvers. Moreover this library contributes as a standard format for all SMT tools; each tool comes with its own input language. This library has been used in the Satisfiability Modulo Theories Competition (SMT-COMP). SMT-COMP is an initiative that started in 2005 to stimulate the development of SMT tools. It is also a jump start to the SMT-LIB initiative's standardization and benchmark collection activities. The SMT-LIB library consists of descriptions of theories, logics and problems. The complete description is available at: <http://combination.cs.uiowa.edu/smtlib>. The list of logics from <http://combination.cs.uiowa.edu/smtlib> is provided below.

- **AUFLIA:** Closed linear formulas over the theory of integer arrays with free sort, function and predicate symbols.
- **AUFLIRA:** Closed linear formulas with free function and predicate symbols over a theory of arrays of arrays of integer index and real value.
- **AUFNIRA:** Closed formulas with free function and predicate symbols over a theory of arrays of arrays of integer index and real value.
- **QF_A:** Closed quantifier-free formulas over the theory of arrays without extensionality.
- **QF_AUFLIA:** Closed quantifier-free and linear formulas over the theory of integer arrays with free sort, function and predicate symbols.
- **QF_AX:** Closed quantifier-free formulas over the theory of arrays with extensionality.
- **QF_IDL:** Difference Logic over the integers. In essence, Boolean combinations of inequations of the form $x - y < b$ where x and y are integer variables and b is an integer constant.

- **QF_LIA**: Unquantified integer linear arithmetic. In essence, Boolean combinations of inequations between linear polynomials over integer variables.
- **QF_LRA**: Unquantified real linear arithmetic. In essence, Boolean combinations of inequations between linear polynomials over real variables.
- **F_RDL**: Difference Logic over the reals. In essence, Boolean combinations of inequations of the form $x - y < b$ where x and y are real variables and b is a rational constant.
- **QF_UF**: Unquantified formulas built over a signature of uninterpreted sort, function and predicate symbols.
- **QF_UFIDL**: Difference Logic over the integers but with uninterpreted sort, function, and predicate symbols.
- **QF_UFBV32**: Unquantified formulas over bit vectors of size up to 32 bits, with uninterpreted function, and predicate symbols.
- **QF_UFLIA**: Unquantified integer linear arithmetic with uninterpreted sort, function, and predicate symbols.
- **QF_UFLRA**: Unquantified real linear arithmetic with uninterpreted sort, function, and predicate symbols.

2.4 Existing Little Engine of Proof Software

The following is a list of state-of-the-art little engines of proof software.

- **Barcelogic**, <http://www.lsi.upc.es/oliveras/bclt-main.html>, Technical University of Catalonia, Spain.
Barcelogic is an SMT solver developed at the Technical University of Catalonia in Spain. Barcelogic is written in C++ and implements the Davis-Putnam-Logemann-Loveland (DPLL) algorithm used to decide propositional logic formulae expressed in conjunctive normal forms. This tool is capable of handling different theories such as: integers, reals, equalities with uninterpreted functions and the interpreted function symbols such as predecessor and successor, or a combination of these theories.
- **CVC Lite**, <http://www.cs.nyu.edu/acsys/cvcl>, New York University, USA.
CVC Lite is written in C++. This tool is capable of handling logics such as : equalities with uninterpreted functions, linear arithmetic (integer, real), arrays and bitvectors.
- **haRVey**, <http://www.loria.fr/equipes/cassis/software/haRVey>, INRIA (The French Institute for Research in Computer Science and Control), France.
haRVey is written in C. This tool is capable of handling theories such as: arrays, lists, Presburger arithmetic and combination of these theories.

- **ICS**, <http://fm.csl.sri.com/simplics>, Stanford Research Institute (SRI), USA.

ICS is a decision procedure system developed at SRI. This tool is written in OCaml. This tool is capable of handling different theories such as: equality with uninterpreted and interpreted functions, linear arithmetic (real and integer), tuples, coproducts, arrays and bitvectors. ICS has been superseded by Yices.

- **MathSAT**, <http://mathsat.itc.it>, The Trentino Cultural Institute, Italia.

MathSAT is written in C and C++. This tool is an extension of propositional satisfiability where formulae are either boolean expressions or built on linear arithmetic for reals, integers and uninterpreted symbols.

- **Yices**, <http://fm.csl.sri.com/yices>, Stanford Research Institute (SRI), USA.

Yices is described in section 3.

Other systems include Ario (<http://www.eecs.umich.edu/ario>, University of Michigan, USA), HTP (<http://www.fordocsys.com/htp.htm>, Formal Documentation System Inc, USA), Sammy (<http://goedel.cs.uiowa.edu/Sammy>, University of Iowa, USA), SIMPLICS (<http://fm.csl.sri.com/simplics>, SRI, USA), and Zap (<http://research.microsoft.com/research/srr>, Microsoft, USA).

2.5 SMT-COMP

The following tables summarize the results of SMT-COMP 2006 and the rules that were used to score the results of the solvers. The solvers were benchmarked on a certain number of problems for each logic. For example, for QF_UF, there are 100 problems. The scores are computed as showed in table 2.4. For example, if a problem that is satisfiable is reported as unsatisfiable by a solver, the score is decreased of 8 points. Barcelogic was considered the best solver in 2005. Yices was considered the best solver in 2006. It won in all categories except in AUFLIRA (Closed linear formulas with free function and predicate symbols over a theory of arrays of arrays of integer index and real value). The complete results of SMT-COMP 2005 and SMT-COMP 2006 are available at: <http://combination.cs.uiowa.edu/smtlib>.

Logics	QF_UF	QF_RDL	QF_IDL	QF_UFIDL	QF_LRA
Total Number of Problems	100	102	103	102	102
Barcelogic	81	95	96	102	N/A
CVC Lite	47	27	70	58	29
MathSAT	69	53	95	100	48
Yices	88	99	97	102	101

Table 2.1: SMT-COMP 2006 - Scores (1)

Logics	QF_LIA	QF_UFLIA	QF_UFBV32	QF_AUFLIA
Total Number of Problems	105	102	100	106
Barcelogic	N/A	N/A	N/A	N/A
CVC Lite	43	63	98	60
MathSAT	85	96	100	N/A
Yices	92	102	100	106

Table 2.2: SMT-COMP 2006 - Scores (2)

Logics	AUFLIA	AUFLIRA
Total Number of Problems	101	107
Barcelogic	N/A	N/A
CVC Lite	50	96
MathSAT	N/A	N/A
Yices	96	96

Table 2.3: SMT-COMP 2006 - Scores (3)

Reported	Correct	Incorrect
Unsat	1	-8
Sat	1	-8
Unknown	0	0
Timeout	0	0

Table 2.4: SMT-COMP 2006 - Scoring Rules

Chapter 3

Yices

Contents

3.1	General Description	15
3.2	Important Features	16
3.2.1	Inputs	16
3.2.2	Types	17
3.2.3	Outputs	18
3.2.4	Logical Context	18
3.3	Examples	19
3.3.1	Integer Linear Arithmetic	19
3.3.2	Real Linear Arithmetic	19
3.3.3	Bitvectors	20
3.3.4	Arrays	20
3.3.5	Lists	21
3.3.6	Program Verification	22
3.3.7	The Graph Coloring Problem	23
3.4	Java Lite API	27
3.4.1	Methods	27
3.4.2	Example	28
3.4.3	Implementation Details	29
3.4.4	Implementation Difficulties	30
3.5	Web Based Extension	31
3.5.1	Example	32
3.5.2	Implementation Details	33
3.5.3	Implementation Difficulties	34

In this chapter we describe the Yices little engine or proof, some important features of Yices including inputs, outputs, types and logical contexts. We also present our contributions: Yices examples and extensions.

3.1 General Description

Yices is a SMT solver/prover developed by Leonardo de Moura at SRI capable of handling theories including uninterpreted functions, integer and real arithmetics, lambda expressions, (extensional) arrays, bitvectors, tuples, records and

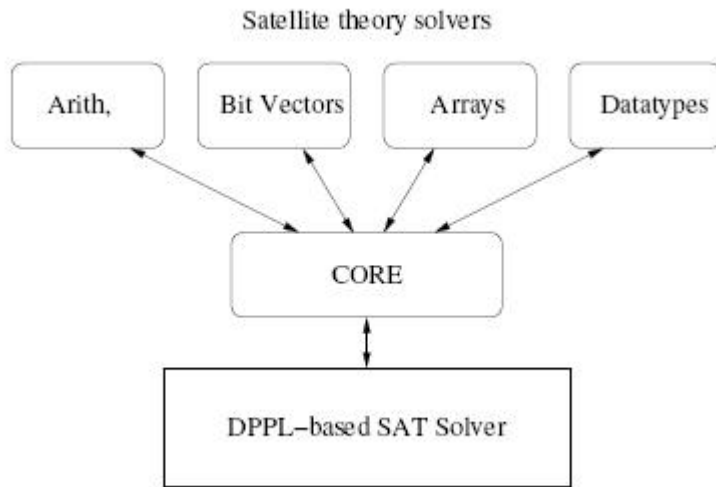


Figure 3.1: Yices Architecture (Source: <http://yices.csl.sri.com/>)

recursive datatypes (e.g. lists). It was extended to deal with quantified expressions. Yices is written in C++.

Yices comes with its own input language based on the SAL languages; SAL [7] is a model checker developed SRI. It also supports the SMT-LIB format. Its concrete syntax is similar to the one of Scheme and Lisp. It is the backend default solver of the SAL [7] model checker.

Yices implements the Davis-Putnam-Logemann-Loveland (DPLL) algorithm used to decide propositional logic formulae expressed in conjunctive normal forms. The Yices architecture relies on the *core theory solver* (to handle equalities and uninterpreted functions) and *satellite solvers* (to handle the theories of arithmetic, bitvectors...). The combination of the core and satellite solvers is facilitated by the Nelson-Oppen method, the standard method used to combine theories [17, 18]. The architecture of Yices is described in figure 3.1

The Yices prototype (Yices 0.1) entered the SMT-COMP 2005. It ranked first for linear integer arithmetic (QF LIA), and arrays, uninterpreted functions, and linear integer arithmetic (QF AUFLIA). In 2006, Yices 1.0 ranked first at SMT-COMP 2006 in all categories except in AUFLIRA (Closed linear formulas with free function and predicate symbols over a theory of arrays of arrays of integer index and real value).

A short tutorial on Yices is available at: <http://yices.csl.sri.com>.

3.2 Important Features

3.2.1 Inputs

The input language of Yices permits us to declare types, constants and functions, to assert formulae and to check their (un)satisfiability (in a particular context).

How to define types is described in section 3.2.2. Typed constants and functions are defined using the *define* keyword. Asserting a formula implies the

addition of the formula (as true) to the current context (see section 3.2.4). This is done with the keyword *assert* or *assert+*. The difference between *assert* and *assert+* is that a formula that is asserted using *assert+* can be removed from the context. The command *check* checks whether the current logical context is satisfiable or not.

Example

```
(define x::bool) ;; x is declared as a bool

(define y::bool)

(assert+ (/= x y))
```

3.2.2 Types

Yices uses typed expressions. A type is defined using the keyword *define-type*.

- The primitive types are *nat*, *int*, *real* and *bool*.
- The types of functions are expressed by: $\rightarrow T_p T_r$, where T_p is the type of the parameters and T_r is the type of the result.
- The types of tuples are expressed by: *(tuple $T_1 \dots T_n$)*, where T_i is the type of the i^{th} component of the tuple.
- The types of records are expressed by: *(record $ID_1 :: T_1 \dots ID_n :: T_n$)* where ID_i is the name of the record field and T_i is the type of that field.
- The types of recursive datatypes are expressed using constructors and accessors. The definition is of the form:

(datatype CONSTRUCTORDEF₁ ... CONSTRUCTORDEF_n)

CONSTRUCTORDEF_i is of the form:

CONSTNAME or *(CONSTNAME ACCESSOR₁ :: T₁ ... ACCESSOR_n :: T_n)*

where:

CONSTNAME is a constructor name and *ACCESSOR_i* is an accessor name.

- The bitvector type is introduced with the keyword *bitvector*.

Example

```
(define i::int)

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

(define t::(tuple int bool))
```

```
(define complexnum::(record r::real i::real))

(define-type intlist (datatype nil (cons car::int cdr::list)))

(define b::(bitvector 5)) ;; bitvector of size 5
```

3.2.3 Outputs

Yices has three kinds of outputs:

- *sat* (satisfiable)
- *unsat* (unsatisfiable)
- *unsat* with unsatisfiable core (the minimum set of formulae that are unsatisfiable)

A model or a counter-example is provided when the level of verbosity of Yices is set to an int different from 0. The verbosity level specifies the amount of feedback messages Yices will produce.

Example `(set-verbosity! 3)`

3.2.4 Logical Context

Yices maintains the logical context of a given problem. The logical context contains the following information:

- Terms: all the terms identified within the given problem.
- Linear arithmetic: all linear arithmetic formulae identified within the given problem.
- Bitvector theory: all bitvector formulae identified within the given problem.
- Boolean constraints: all boolean constraints and clauses identified within the given problem.

A context can be saved using *(push)*, reset using *(reset)* and display the status of the logical context using *(status)*.

Example Below is an example of logical context (see example 3.3.4 in section 3.3).

```
terms:
t!12 := (a i) ;; t!12 is a name of variable in the internal
;; representation used by Yices. It is like t12 = a(i)
;; this representation is used because of DPLL and the combination
t!14 := (b k)
t!15 := (a j)
t!17 := (b l)
t!18 := (a m)
```

```

linear arithmetic:
(0 = 0) [0]
bitvector theory:
boolean constraints:
p!4 := (= i j)
;; p!4 is i=j
p!6 := (= k 1)
p!8 := (= t!12 t!14)
p!10 := (= j t!15)
p!12 := (= m t!17)
p!14 := (= t!14 t!18)
 clause p!3 p!4)
 clause p!5 p!6)
 clause p!7 p!8)
 clause p!9 p!10)
 clause p!11 p!12)
 clause p!13 (not p!14))

```

3.3 Examples

In this section we propose a list of examples we designed to showcase the use of Yices in different theories.

3.3.1 Integer Linear Arithmetic

In this example we solve the following equational system in the integers.

$$\begin{cases} 16x + 2y = 18 \\ x + y = 2 \end{cases}$$

Input:

```

(set-evidence! true)

(define x::int)

(define y::int)

(assert+ (= 18 (+ (* x 16) (* y 2))))

(assert+ (= 2 (+ (* x 1) (* y 1))))

(check)

```

Output:

```
sat (= x 1) (= y 1) ;; the solution
```

3.3.2 Real Linear Arithmetic

In this example we solve the following equational system in the integers.

$$\begin{cases} 17/2x + 2y = 18 \\ -x + 2y = 2 \end{cases}$$

Input:

```
(set-evidence! true)

;; Enables the construction of evidence: models
;; and unsatisfiable cores (define x::real) (define y::real)

(assert+ (= 18 (+ (* x (/ 17 2)) (* y 2))))

(assert+ (= 2 (+ (* x -1) (* y 2))))

(check)
```

Output:

```
sat (= x 32/19) (= y 35/19) ;; the solution
```

3.3.3 Bitvectors

This example checks if a bitvector represents an even number.

Input:

```
(define b::(bitvector 8))

(assert+ (= b (mk-bv 8 10)))

;; last bit is at position 0. If the last bit in the bitvector is 0,
;; the bitvector represents an even number

(assert+ (= (bv-extract 0 0 b) 0b0))

(check)
```

Output:

```
sat
```

3.3.4 Arrays

Given 2 arrays of integers a and b and integers i , j , k , l and m , we have: $i = j \wedge k = l \wedge a[i] = b[k] \wedge b[l] = m \rightarrow a[m] = b[k]$ [22]. To verify this, we will check that the negation is unsatisfiable i.e. $i = j \wedge k = l \wedge a[i] = b[k] \wedge b[l] = m \wedge a[m] \neq b[k]$ is unsatisfiable.

Input:

```

(set-evidence! true)

(define a::(-> int int))
(define b::(-> int int))
(define i::int)
(define j::int)
(define k::int)
(define l::int)
(define m::int)

(assert+ (= i j)) ;; 1

(assert+ (= k l)) ;; 2

(assert+ (= (a i) (b k))) ;; 3

(assert+ (= j (a j))) ;; 4

(assert+ (= m (b l))) ;; 5

(assert+ (/= (a m) (b k))) ;; 6

(check)

```

Output:

unsat

unsat core ids: 1 2 3 4 5 6

;; Yices provides the minimum number of expressions involved in the

;; unsatisfiability.

3.3.5 Lists

This example checks whether two lists *l1* and *l2* are equal.

Input:

```

(define-type list (datatype (cons car::int cdr::list) nil))

(define L1::list (cons 1 nil))

(define L2::list (cdr (cons 2 (cons 1 nil))))

(assert+ (= (car L1) (car L2)))

(assert+ (= (cdr L1) (cdr L2)))

```

```
(assert+ (= L1 12))
```

```
(check)
```

Output:

```
sat
```

3.3.6 Program Verification

Let us consider the following code that computes the maximum of a and b .

```
// a and b are ints
int max = 0;
// Pre-condition: true
if (a <= b){
    max := b;
}
else max := a;
// Post-condition
assert (max >= a && max >= b && (max = a || max = b));
return max;
```

We propose to check that this program is correct by using the Hoare Logic axiomatic semantics. The central feature of Hoare logic is the Hoare triple $\{P\} C \{Q\}$. P is a pre-condition, and Q is a post-condition. P and Q are assertions, which are formulae written in predicate logic and claimed to be true during program runtime execution. C is a sequence of statements of the program. $\{P\} C \{Q\}$ describes how the execution of a piece of code changes the state of the computation. Hoare logic has axioms and inference rules for all the common constructs of modern programming languages (e.g. sequence, assignment, conditional and loop statements).

Proving the correctness of program fragment corresponds to proving that if the pre-condition stands before the execution of the code, then the post-condition stands after the execution of the code. This is equivalent to proving that the initial pre-condition implies the generated pre-condition [11]. Proving that $p \rightarrow q$ is valid using Yices is done by (refutationally) proving that $\sim p \rightarrow q$ (or equivalently $p \ \& \ \sim q$) is unsatisfiable.

To prove that the *maximum* program is correct we need to prove that the negations of the 2 following formulae are unsatisfiable:

- $a \leq b \rightarrow (b \geq a \text{ and } b \geq b \text{ and } (b = a \text{ or } b = b))$
- $a > b \rightarrow (a \geq a \text{ and } a \geq b \text{ and } (a = a \text{ or } a = b))$

Input:

```
(set-evidence! true)
```

```
(define a::int)
```

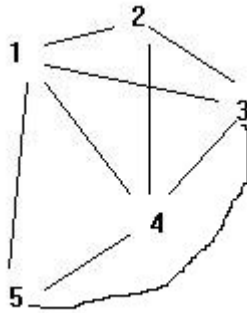


Figure 3.2: Coloring Problem Graph

```
(define b::int)

(define Precond1::bool (<= a b))
(define Precond2::bool (not (<= a b)))

(define Precondprime1::bool (and (and (>= b a) (>= b b)) (or (= b a) (= b b))))
(define Precondprime2::bool (and (and (>= a a) (>= a b)) (or (= a a) (= a b))))

(assert (not (=> Precond1 Precondprime1)))

(assert (not (=> Precond2 Precondprime2)))

(check)
```

Output:

unsat

3.3.7 The Graph Coloring Problem

This example deals with the graph coloring problem. This problem requires that, whatever colors are actually used, no two adjacent vertices may not have the same color.

We consider the graph in figure 3.2

The vertices of the graph are represented by integers between 1 and 5. Adjacent vertices are defined by the *adj* function. We color the vertices of the graph with the red, blue, yellow and green colors. Vertices colors are defined by the *c* function.

Input:

```
;; inspired from
;; http://www.csupomona.edu/~jrfisher/www/prolog_tutorial/2_1.html
```

```
(set-evidence! true)
```

```

(define-type color (scalar red blue yellow green))

;; vertices are represented by ints from 1 to 5
(define adj::(-> (subrange 1 5) (subrange 1 5) bool))

;; function that associates a color to a vertex
(define c::(-> (subrange 1 5) color))

;; Colors of the vertices

(assert+ (= (c 1) red)) ;; 1

(assert+ (= (c 2) red)) ;;2

(assert+ (= (c 3) red)) ;; 3

(assert+ (= (c 4) red)) ;; 4

(assert+ (= (c 5) red)) ;; 5

;; Description of the graph

(assert+ (= (adj 1 1) false)) ;; 6

(assert+ (= (adj 1 2) true)) ;; 7

(assert+ (= (adj 1 3) true)) ;; 8

(assert+ (= (adj 1 4) true)) ;; 9

(assert+ (= (adj 1 5) true)) ;; 10

(assert+ (= (adj 2 1) true)) ;; 11

(assert+ (= (adj 2 2) false)) ;; 12

(assert+ (= (adj 2 3) true)) ;; 13

(assert+ (= (adj 2 4) true)) ;; 14

(assert+ (= (adj 2 5) false)) ;; 15

(assert+ (= (adj 3 1) true)) ;; 16

(assert+ (= (adj 3 2) true)) ;; 17

(assert+ (= (adj 3 3) false)) ;; 18

(assert+ (= (adj 3 4) true)) ;; 19

```

```

(assert+ (= (adj 3 5) false)) ;; 20
(assert+ (= (adj 4 1) true)) ;; 21
(assert+ (= (adj 4 2) true)) ;; 22
(assert+ (= (adj 4 3) true)) ;; 23
(assert+ (= (adj 4 4) false)) ;; 24
(assert+ (= (adj 4 5) true)) ;; 25
(assert+ (= (adj 5 1) true)) ;; 26
(assert+ (= (adj 5 2) false)) ;; 27
(assert+ (= (adj 5 3) true)) ;; 28
(assert+ (= (adj 5 4) true)) ;; 29
(assert+ (= (adj 5 5) false)) ;; 30

;; QUERY 1

;; 1 is adjacent to all the other vertices

(assert (forall (v1::(subrange 1 5)) (= (adj 1 v1) true)))

;; Output 1:

;;unsat

unsat
core ids: 6 ;;
(assert+ (= (adj 1 1) false)) ;; 6

;; QUERY 2

;; 1 is adjacent to 2, 3, 4 and 5

(define v1::(subrange 2 5))
(assert (= (adj 1 v1) true))

;; Output 2:

sat (= v1 2) (= (c 1) red) (= (c 2) red) (= (c 3) red) (= (c 4) red)
(= (c 5) red) (= (adj 1 1) false) (= (adj 1 2) true) (= (adj 1 3)
true) (= (adj 1 4) true) (= (adj 1 5) true) (= (adj 2 1) true) (=
(adj 2 2) false) (= (adj 2 3) true) (= (adj 2 4) true) (= (adj 2 5)
false) (= (adj 3 1) true) (= (adj 3 2) true) (= (adj 3 3) false) (=

```

```

(adj 3 4) true) (= (adj 3 5) false) (= (adj 4 1) true) (= (adj 4 2)
true) (= (adj 4 3) true) (= (adj 4 4) false) (= (adj 4 5) true) (=
(adj 5 1) true) (= (adj 5 2) false) (= (adj 5 3) true) (= (adj 5 4)
true) (= (adj 5 5) false)

;; QUERY 3

;; using forall is incomplete

;; It is true but Yices' answer is unknown because Yices is

;; incomplete in the case with variables

(assert (forall (v1::(subrange 2 5)) (= (adj 1 v1) true)))

;; Output 3:

unknown (= (c 1) red) (= (c 2) red) (= (c 3) red) (= (c 4) red) (=
(c 5) red) (= (adj 1 1) false) (= (adj 1 2) true) (= (adj 1 3) true)
(= (adj 1 4) true) (= (adj 1 5) true) (= (adj 2 1) true) (= (adj 2
2) false) (= (adj 2 3) true) (= (adj 2 4) true) (= (adj 2 5) false)
(= (adj 3 1) true) (= (adj 3 2) true) (= (adj 3 3) false) (= (adj 3
4) true) (= (adj 3 5) false) (= (adj 4 1) true) (= (adj 4 2) true)
(= (adj 4 3) true) (= (adj 4 4) false) (= (adj 4 5) true) (= (adj 5
1) true) (= (adj 5 2) false) (= (adj 5 3) true) (= (adj 5 4) true)
(= (adj 5 5) false)

;; QUERY 4

;; 1 is adjacent to 5. 1 is colored in red. 5 is colored in red.

(assert+ (and (and (= (adj 5 1) true) (= (c 5) red)) (= (c 1) red)))

;; Output 4:

sat (= (c 1) red) (= (c 2) red) (= (c 3) red) (= (c 4) red) (= (c 5)
red) (= (adj 1 1) false) (= (adj 1 2) true) (= (adj 1 3) true) (=
(adj 1 4) true) (= (adj 1 5) true) (= (adj 2 1) true) (= (adj 2 2)
false) (= (adj 2 3) true) (= (adj 2 4) true) (= (adj 2 5) false) (=
(adj 3 1) true) (= (adj 3 2) true) (= (adj 3 3) false) (= (adj 3 4)
true) (= (adj 3 5) false) (= (adj 4 1) true) (= (adj 4 2) true) (=
(adj 4 3) true) (= (adj 4 4) false) (= (adj 4 5) true) (= (adj 5 1)
true) (= (adj 5 2) false) (= (adj 5 3) true) (= (adj 5 4) true) (=
(adj 5 5) false)

;; QUERY 6

;; 1 and 5 are colored by different colors

(define c1::color)

```

```
(define c2::color)
(assert+ (and (and (and (= (adj 5 1) true) (= (c 5) c1)) (= (c 1) c2)) (/= c1 c2)))

;; Output 6:

unsat
unsat core ids: 1 5 31
```

3.4 Java Lite API

Yices was released with two types of C libraries: the C Lite API and the C API. The C Lite API uses the Yices input language. The C API is under development and does not require users to know the Yices syntax. We designed a Java Lite API for Yices based on the C Lite API. It is available at: <http://atlantis.seidenberg.pace.edu/wiki/lep>.

3.4.1 Methods

The Yices Java Lite API contains the same methods as the Yices C Lite API. The signatures changed because Java was used. We implemented the following public methods:

- `void yicesl_del_context(int ctx)`
This public method deletes an existing context *ctx* provided in the parameter.
- `void yicesl_enable_log_file(Java.lang.String filename)`
This method logs all Yices input commands into a file specified in the parameter.
- `void yicesl_enable_type_checker(short flag)`
This method enables the type checker. By default, Yices engine assumes that the inputs have the appropriate type.
- `Java.lang.String Yicesl_get_last_error_message()`
This method gets the last error message generated by Yices.
- `int Yicesl_inconsistent(int ctx)`
This method checks the inconsistency of the current logical context referenced by *ctx*.
- `int Yicesl_mk_context()`
This method creates a new logical context.
- `int Yicesl_read(int ctx, Java.lang.String cmd)`
This method reads and executes the Yices input commands in the context *ctx*.

- `void Yicesl_set_output_file(Java.lang.String filename)`
This method redirects the result of the execution of the Yices input commands into an output file *filename*.
- `void Yicesl_set_verbosity(short l)`
This method sets the verbosity level. By default, the verbosity level is 0 (zero), which prevents it from producing any model. Increasing the verbosity level increases the information printed.
- `Java.lang.String Yicesl_version()`
This method returns the version number of Yices.

3.4.2 Example

Yices Code

```
(define x::int)
(define y::int)
(set-evidence! true)
(assert (and (> x 0) (> y 0)))
(assert (> (+ x y) 0))
(check)
(dump-context)
```

Java Code

```
import yices.*;

public class TestYicesLite {

    public static void main(String[] args) {
        YicesLite yices = new YicesLite();
        int ctx = yices.yicesl_mk_context();
        String str = yices.yicesl_version();
        System.out.println("My version is: "+str);
        short level = 2;
        yices.yicesl_set_verbosity(level);
        yices.yicesl_read(ctx,"(define x::int)");
        yices.yicesl_read(ctx,"(define y::int)");
        yices.yicesl_read(ctx,"(set-evidence! true)");
        yices.yicesl_read(ctx,"(assert (and (> x 0) (> y 0)))");
        yices.yicesl_read(ctx,"(assert (> (+ x y) 0))");
        yices.yicesl_read(ctx,"(check)");
        yices.yicesl_read(ctx,"(dump-context)");
        yices.yicesl_del_context(ctx);
    }
}
```

Output

```

[sokharith@tabletfc API]$ Java TesstYicesLite

My version is: 1.0

sat

(= x 1)

(= y 1)

terms:

linear arithmetic:

0: s!3 (= (+ (* -1 x) (* -1 y) (* 1 s!3)) 0)

(0 = 0) [0]

1 <= (x = 1) [0]:i

1 <= (y = 1) [0]:i

1 <= (s!3 = 2) [0]:i

bitvector theory:

boolean constraints:

p!3 := (<= x 0)

p!4 := (<= y 0)

p!5 := (<= s!3 0)

(not p!3)

(not p!4)

(not p!5)

```

3.4.3 Implementation Details

The Yices Java Lite API was designed by reusing the existing C library that is included within the Yices installation package provided by SRI. The Java API is implemented using the Java Native Interface (JNI) to have a bridge from the Java code to the native code implemented in C.

In figure 3.3, the Java API of YicesLite is encapsulated inside *YicesLite.java*, which calls *libYicesLite.so* a dynamic C library, which works together with the existing dynamic Yices C Lite API library provided by SRI.

The JNI has been deployed to facilitate the reuse of the existing Yices C

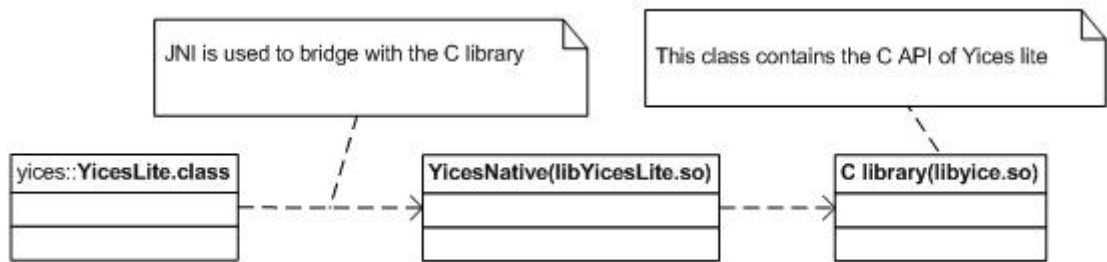


Figure 3.3: Yices Java Lite API Architecture

API for the development of the Yices Java Lite API.

The implementation of the Yices Java Lite API was done in the following steps:

- **Implementation of Java code** For Java, *YicesLite.java* contains all declaration of the Yices lite API as native code with the keyword "native". Once the Java code is finished, it is compiled into bytecode (*.class* extension) by using the Java compiler.
- **Generation of the C header file** The next step is to generate the C header file to be included in the real implementation of the native methods declared in the previous Java file. The generation of the header file can be done as follows:

```
javah -jni Yices.YicesLite
```

- **Implementation of C code** The real implementation of the methods declared in the previous Java file needs to be coded using the methods (functions) generated from the previous step (from the header file).
- **Compilation** The implementation of the native code needs to be compiled to create a new library necessary for the Yices Java Lite API. A makefile was used to facilitate the compilation.

Makefile:

```
libYicesLite.so : YicesLite.c YicesLite.h
gcc -o libYicesLite.so -shared -static
-Wall -lyices -I/usr/Java/jdk1.5.0_09/include
-I/usr/Java/jdk1.5.0_09/include/linux
-shared -lc YicesLite.c -lgmp
```

3.4.4 Implementation Difficulties

There were many difficulties we encountered during the development of the Java Lite API for Yices:

- **Lack of documentation** There is a real lack of documentation on JNI technology, especially the document on compiling the code, loading the C library and packaging the Java API library.

- **Lack of Tools** The lack of Development Environment was another challenge for the development of this API. To date, there are a lot of Integrated Development Environment (IDE) for Java and C, but there is no IDE available to support the development using the JNI technology.
- **Datatypes** Datatypes were another challenge for the development of this API. Datatypes are represented differently in programming languages (in this case Java and C). For example, an *int* is coded in 4 bytes in Java but only 2 bytes in C. Another problem is the pointer type in C which does not exist in Java.
- **Testing Environment** During the testing phase of the developed API, testing environment set up became a major challenge.

3.5 Web Based Extension

Installing, configuring and working with Yices is a challenge for many users. Yices is only provided for certain versions of Linux; it does not work on Windows. This work looks at the development of a web based interface for Yices that will permit users to use Yices directly from their browsers. In this section, we focus on the development of the web based extension as well as the difficulties we encountered during the development.

With our web based extension of Yices, the user can interact directly with Yices. The user types in Yices input commands and the results are printed back on the screen. This web based interface is inspired from the interface of Try Ruby! in your browser available at: <http://tryruby.hobix.com>. We used a similar web design.

The Yices web based interface provides the same functionalities as the interactive version of Yices (command line). The following Yices input commands can be used through the web interface:

- `(assert [expr])` : asserts the formula [expr] into the current logical context.
- `(assert+ [expr])` : similar to `assert`, but allows retraction and unsat core extraction.
- `(assert+ [expr] [weight])` : similar to `assert+`, but the weight is used for max-sat.
- `(retract [idx])` : retracts the assertion with index `idx` from the logical context.
- `(check)` : checks whether the logical context is satisfiable or not.
- `(max-sat)` : extracts the maximal satisfying model.
- `(set-evidence! true)` : enables the construction of evidence: models and unsatisfiable cores.
- `(set-evidence! false)` : disables the construction of evidence: models and unsatisfiable cores.
- `(set-verbosity! [num])` : sets the verbosity level.

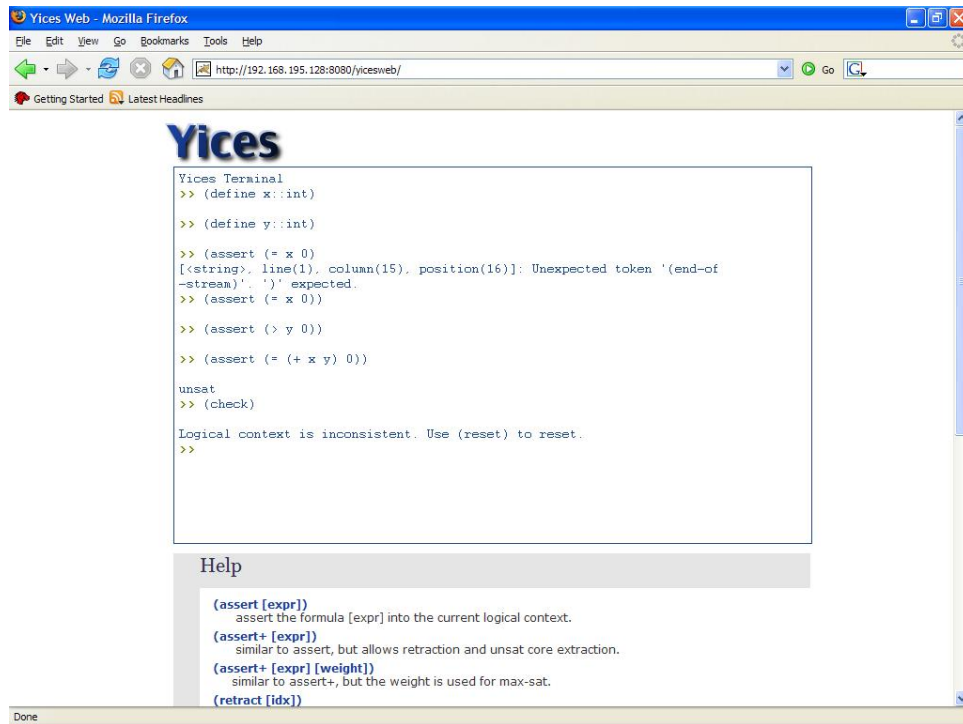


Figure 3.4: Screenshot of Yices Web Interface

- (set-arith-only! true) : tells Yices that only boolean combinations of arithmetic constraints are going to be asserted.
- (push) : saves the current logical context on the stack.
- (pop) : restores the context from the top of the stack, and pop it off the stack.
- (echo [string]) : prints the string [string].
- (reset) : resets the logical context.
- (status) : displays the status of the logical context.
- (dump-context) : displays the logical context.
- (exit) : *not Applicable for Yices web based.*
- (help) : display this message.

3.5.1 Example

Figure 3.4 is a screenshot of the Yices Web extension. You can try Yices at: <http://atlantis.seidenberg.pace.edu/wiki/lep>.

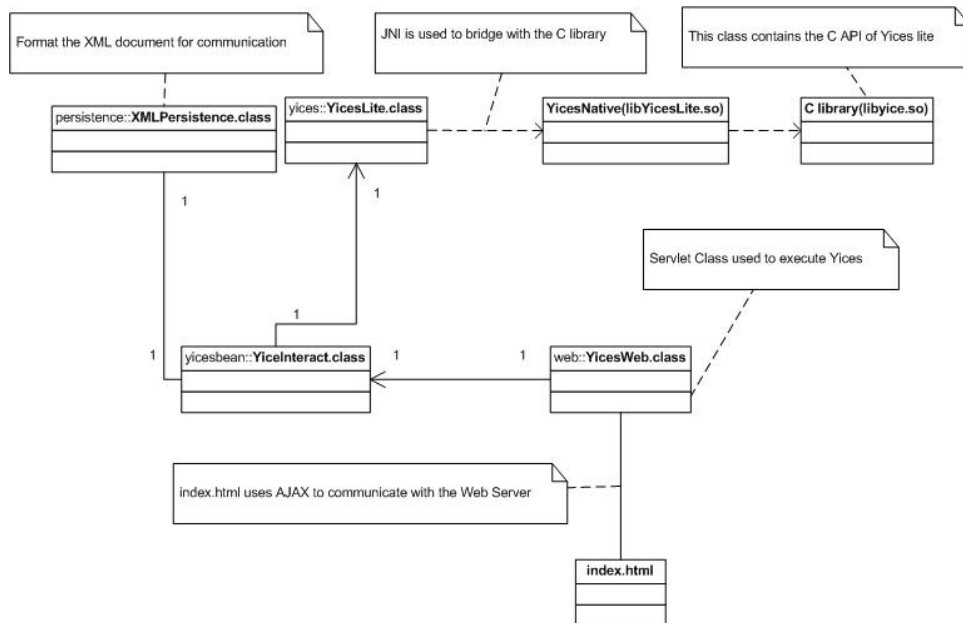


Figure 3.5: Yices Web Extension Architecture

3.5.2 Implementation Details

The Eclipse Integrated Development Environment (IDE) and the Web Tool Project (WTP) plug-in have been used to support the development of this web based application.

Architecture

The web based interface of Yices is developed using a client server architecture. It is built using the Yices Java Lite API. The client uses the browser to contact the application server (tomcat) and interacts with the server using Ajax. The user uses the browser to load the Yices web based interface and can interact with the system using the Yices input language in the interactive mode. In the background, Ajax communicates with the web server. User inputs commands are sent to the webservice (Servlets) to be run. Ajax analyzes the resulting XML document and puts it back into the screen.

Figure 3.5 shows the architecture of the web interface of Yices. The main page (*index.html*) of Yices is built using HTML to describe and organize the view of the web page. Javascript handles the onscreen interaction with the user and the communication with the server (Servlets).

Domain Model

The application handling the execution of the web based Yices is organized into packages: *web*, *Yicesbean*, *Yices* and *persistence* (see figure 3.5).

- **Web package** contains the Servlets class which handles the requests from the browser (using Ajax) and returns an XML document containing the

input and the output Yices produced.

- **Yicesbean package** contains the bean class which handles the requests from the Servlets and analyzes and executes the Yices commands.
- **Yices package** contains the Yices Java Lite API. It handles the requests from the bean class and calls the existing Yices C library.
- **Persistence package** is responsible for formatting the input and output of the Yices input command into the appropriate format (XML).

The XML document used by Ajax to communicate with the server consists of the root element of Yices, which is composed of many child *commands*. A *command* node has 2 different types of children *input* and *output*.

3.5.3 Implementation Difficulties

- **Interactive Editor** There is no editor component in the HTML specification.
- **Technology** The complication of the project relies on the use of many technologies: XML, Ajax, and Servlets.
- **Tools** The *vi* editor was used to edit the code and some manual compilations is needed as the Web Tool Project (WTP) plug-in does not work well under Linux platform.

Chapter 4

Conclusion and Future Work

To make Yices more accessible to the community, we have developed and gathered documentation, material, applicative examples and extended Yices by developing a Java API and a Web extension.

This work resulted in:

- **Examples** We developed applicative examples for linear arithmetic (integer and real), bitvectors, arrays, lists, program verification and the coloring graph problem.
- **Java API** We developed and documented a Java API for Yices which is equivalent to the existing Yices C Lite API.
- **Web extensions** We developed a Web interface for Yices to facilitate access to Yices.

This work will help anyone who is interested in using Yices or integrate it in Java based applications.

Our future plans include the development of an Eclipse plug-in for Yices and use Yices in static analysis of Java code based on the Hoare Logic approach like it is done in ESC/Java [6].

Bibliography

- [1] W. Ackerman. Solvable cases of the decision problem. *Studies in logic and the foundations of mathematics*, 1954.
- [2] Leo Bachmair, I. V. Ramakrishnan, Ashis Tiwari, and Laurent Vigneron. Congruence closure modulo associativity and commutativity. In H. Kirchner and Ch. Ringeissen, editors, *Proceedings of the 3rd International Workshop on Frontiers of Combining Systems, FroCoS'2000, Nancy (France)*, volume 1794, pages 245–259. Springer-Verlag, 2000.
- [3] Clark Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In Rajeev Alur and Doron A. Peled, editors, *Proceedings of the 16th International Conference on Computer Aided Verification (CAV '04)*, volume 3114 of *Lecture Notes in Computer Science*, pages 515–518. Springer-Verlag, July 2004. Boston, Massachusetts.
- [4] Maria Paola Bonacina. A taxonomy of theorem-proving strategies. In Michael J. Wooldridge and Manuela Veloso, editors, *Artificial Intelligence Today – Recent Trends and Developments*, volume 1600 of *Lecture Notes in Artificial Intelligence*, pages 43–84. Springer, August 1999.
- [5] Marco Bozzano, Roberto Bruttomesso, Alessandro Cimatti, Tommi Junttila, Peter van Rossum, Stephan Schulz, and Roberto Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. In Enrico Giunchiglia and Toby Walsh, editors, *SAT 2005; Satisfiability Research in the Year 2005*. Springer, to appear.
- [6] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, and Erik Poll. *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2*. Springer-Verlag, 2006.
- [7] Leonardo de Moura, Sam Owre, Harald Rueß, John Rushby, N. Shankar, Maria Sorea, and Ashish Tiwari. SAL 2. In Rajeev Alur and Doron Peled, editors, *Computer-Aided Verification, CAV 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 496–500, Boston, MA, July 2004. Springer-Verlag.
- [8] David Dharbe and Pascal Fontaine. haRVey: combining reasoners. In *Sixth International Workshop on Automated Verification of Critical Systems*, 2006.
- [9] Bruno Dutertre and Leonardo Mendonça de Moura. A fast linear-arithmetic solver for dpll(t). In *CAV*, pages 81–94, 2006.

- [10] Jean-Christophe Filliâtre, Sam Owre, Harald Rueß, and N. Shankar. ICS: integrated canonizer and solver. To be presented at CAV'2001, 2001.
- [11] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.
- [12] D. E. Knuth and P. B. Bendix. Simple word problems in universal algebras. In J. Siekmann and G. Wrightson, editors, *Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970*, pages 342–376. Springer, Berlin, Heidelberg, 1983.
- [13] G. Kolata. Computer math proof shows reasoning power. *New York Times*, december 1996.
- [14] Donald W Loveland. *Automated theorem proving: A logical basis (Fundamental studies in computer science)*. sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978.
- [15] W D Loveland. Automated theorem proving: Mapping logic into ai. Technical report, Durham, NC, USA, 1986.
- [16] William McCune. Solution of the robbins problem. *Journal of Automated Reasoning*, 19(3):263–276, 1997.
- [17] Greg Nelson and Derek C. Oppen. Simplification by cooperating decision procedures. *ACM Trans. Program. Lang. Syst.*, 1(2):245–257, 1979.
- [18] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.
- [19] R. Nieuwenhuis and A. Oliveras. DPLL(T) with Exhaustive Theory Propagation and its Application to Difference Logic. In K. Etessami and S. Rajamani, editors, *17th International Conference on Computer Aided Verification, CAV'05*, volume 3576 of *Lecture Notes in Computer Science*, pages 321–334. Springer, 2005.
- [20] John Alan Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, 1965.
- [21] N. Shankar. Little engines of proof. In L.-H. Eriksson and P. Lindsay, editors, *FME 2002: Formal Methods — Getting IT Right, Copenhagen*, pages 1–20. Springer-Verlag, 2002.
- [22] Robert E. Shostak. An algorithm for reasoning about equality. *Commun. ACM*, 21(7):583–585, 1978.
- [23] Hao Wang. Proving theorems by pattern recognition i. *Commun. ACM*, 3(4):220–234, 1960.
- [24] Hao Wang. Mechanical mathematics and inferential analysis. *Computer Programming and Formal Systems*, pages 1–20, 1963.