Yices 1.0: An Efficient SMT Solver

*AFM’06 Tutorial*

Leonardo de Moura (joint work with Bruno Dutertre)

\{demoura, bruno\}@csl.sri.com.

Computer Science Laboratory
SRI International
Menlo Park, CA
Satisfiability Modulo Theories (SMT)

- SMT is the problem of determining satisfiability of formulas modulo background theories.
- Examples of background theories:
  - linear arithmetic: $x + 1 \leq y$
  - arrays: $a[i := v_1][j] = v_2$
  - uninterpreted functions: $f(f(f(x))) = x$
  - datatypes: $\text{car}(\text{cons}(v_1, v_3)) = v_2$
  - bitvectors: $\text{concat}(bv_1, bv_2) = bv_3$
- Example of formula:

$$i - 1 = j + 2, f(i + 3) \neq f(j + 6)$$
Applications of SMT

- Extended Static Checking
- Equivalence Checking (Hardware)
- Bounded Model Checking (e.g., sal-inf-bmc)
- Predicate Abstraction
- Symbolic Simulation
- Test Case Generation (e.g., sal-atg)
- AI Planning & Scheduling
- Embedded in Theorem Provers (e.g., PVS)
Yices

- Yices is an SMT Solver developed at SRI International.
- Yices is not ICS.
- It is used in SAL, PVS, and CALO.
- It is a complete reimplementation of SRI’s previous SMT solvers.
  - It has a new architecture, and uses new algorithms.
  - Counterexamples and Unsatisfiable Cores.
  - Incremental: push, pop, and retract.
  - Weighted MaxSAT/MaxSMT.
- Supports all theories in SMT-LIB and much more.
Supported Features

- Uninterpreted functions
- Linear real and integer arithmetic
- Extensional arrays
- Fixed-size bit-vectors
- Quantifiers
- Scalar types
- Recursive datatypes, tuples, records
- Lambda expressions
- Dependent types
Using Yices

- Starting yices shell: `./yices -i`
- Batch mode:
  - Yices format: `./yices ex1.ys`
  - SMT-LIB format: `./yices -smt ex1.smt`
  - Dimacs format: `./yices -d ex1.cnf`
- Increasing verbosity level: `./yices -v 3 ex1.ys`
- Producing models: `./yices -e ex1.ys`
First Example

(define f::(-> int int))
(define i::int)
(define j::int)
(assert (= (- i 1) (+ j 2)))
(assert (/= (f (+ i 3)) (f (+ j 6))))

→ unsat
Check

- `assert` gets only trivial inconsistencies.
- `(check)` should be used to test satisfiability.

```
(define x::int)
(define y::int)
(define z::int)
(assert (= (+ (* 3 x) (* 6 y) z) 1))
(assert (= z 2))
(check)
```

→ unsat
Extracting Models

- ./yices -e ex3.ys

(define x::int)
(define y::int)
(define f::(-> int int))
(assert (/= (f (+ x 2)) (f (- y 1))))
(assert (= x (- y 4)))
(check)

→ sat

(= x -2)
(= y 2)
(= (f 0) 1)
(= (f 1) 3)
Extracting Unsatisfiable Cores

- ./yices -e ex4.ys

  (define f :: (-> int int))
  (define i :: int)
  (define j :: int)
  (define k :: int)
  (assert+ (= (+ i (* 2 k)) 10))
  (assert+ (= (f k) (f i)))
  (assert+ (= (+ j 2) (f (+ i 3))) (f (+ j 6)))
  (check)

→ unsat

unsat core ids: 2 4
SMT (and SAT) solvers have a search engine:

- Case-split
- Propagate
- Conflict $\leadsto$ Backtrack

Each conflict generates a Lemma:

- It prevents a conflict from happening again.
Retracting Assertions

- Assertions asserted with `assert+` can be retracted.
- Lemmas are reused in the next call to `(check)`.
  - Yices knows which lemmas are safe to reuse.

\[
\begin{align*}
&\text{(assert+ } (= \ (+ \ i \ (* \ 2 \ k)) \ 10)) \\
&\text{(assert+ } (= \ (- \ i \ 1) \ (+ \ j \ 2))) \\
&\text{(assert+ } (= \ (f \ k) \ (f \ i))) \\
&\text{(assert+ } (= \ (f \ (+ \ i \ 3)) \ (f \ (+ \ j \ 6)))) \\
&\text{(check)} \\
\end{align*}
\]

→ unsat

\[
\begin{align*}
&\text{(retract 2)} \\
&\text{(check)} \\
\end{align*}
\]

→ sat
Stacking logical contexts

- **(push)**
  - Saves the current logical context on the stack.

- **(pop)**
  - Restores the context from the top of the stack.
  - Pops it off the stack.
  - Any changes between the matching push and pop commands are flushed.
  - The context is restored to what it was right before the push.

- **Applications (depth-first search):**
  - Symbolic Simulation
  - Extended Static Checking
Weighted MaxSAT

```plaintext
./yices -e ex5.ys

(assert+ (= (+ i (* 2 k)) 10) 10)
(assert+ (= (- i 1) (+ j 2)) 20)
(assert+ (= (f k) (f i)) 30)
(assert+ (/= (f (+ i 3)) (f (+ j 6))) 15)
(max-sat)

→ sat

unsatisfied assertion ids: 4
(= i 10) (= k 0) (= j 7) (= (f 0) 11)
(= (f 10) 11) (= (f 13) 12)
cost: 10
```
Type checking

- By default, Yices assumes the input is correct.
- It may crash if the input has type errors.
- You can force Yices to “type check” the input:
  - ./yices -tc ex1.ys
  - Performance penalty.
- Idea: use -tc only when you are developing your front-end for Yices.
Other useful commands

- `(reset)` – reset the logical context.
- `(status)` – display the status of the logical context.
- `(echo [string])` – prints the string `[string]`. 
Yices (like PVS) does not make a distinction between arrays and functions.

Function theory handles:
- Function updates.
- Lambda expressions.
- Extensionality
Example: ./yices f1.ys

(define A1::(-> int int))
(define A2::(-> int int))
(define v::int) (define w::int)
(define x::int) (define y::int)
(define g::(-> (-> int int) int))
(define f::(-> int int))
(assert (= (update A1 (x) v) A2))
(assert (= (update A1 (y) w) A2))
(assert (=/= (f x) (f y)))
(assert (=/= (g A1) (g A2)))
(check)

→ unsat
Lambda expressions

Example: ./yices -e f2.ys

(define f::(-> int int))
(assert (or (= f (lambda (x::int) 0))
 (= f (lambda (x::int) (+ x 1)))))
(define x::int)
(assert (and (>= x 1) (<= x 2)))
(assert (>= (f x) 3))
(check)

→ sat

(= x 2) (= (f 2) 3)
Recursive datatypes

- Similar to PVS and SAL datatypes.
- Useful for defining: lists, trees, etc.
- Example: ./yices dt.ys

```lisp
(define-type list
  (datatype (cons car::int cdr::list) nil))
(define l1::list)
(define l2::list)
(assert (not (nil? l2)))
(assert (not (nil? l1)))
(assert (= (car l1) (car l2)))
(assert (= (cdr l1) (cdr l2)))
(assert (/= l1 l2))

→ unsat
```
Fixed-size bit-vectors

- It is implemented as a satellite theory.

- Straightforward implementation:
  - Simplification rules.
  - Bit-blasting for all bit-vector operators but equality.
  - “Bridge” between bit-vector terms and the boolean variables.

- Example: 
  ```
  ./yices -e bv.ys
  (define b::(bitvector 4))
  (assert (= b (bv-add 0b0010 0b0011)))
  (check)
  → unsat
  (= b 0b0101)
  ```
Dependent types

- Useful for stating properties of uninterpreted functions.
- Alternative to quantifiers.
- Example: ./yices -e d.ys

```
(define x::real)
(define y::int)
(define floor::(-> x::real
               (subtype (r::int) (and (>= x r)
                                    (< x (+ r 1))))))
(assert (and (> x 5) (< x 6)))
(assert (= y (floor x)))
(check)

→ sat

(= x 11/2) (= y 5) (= (floor 11/2) 5)
```
Quantifiers

- Main approach: egraph matching (Simplify)
  - Extension for offset equalities and terms.
  - Several triggers (multi-patterns) for each universally quantified expression.
  - The triggers are fired using a heuristic that gives preference to the most conservative ones.
- Fourier Motzkin elimination to simplify quantified expressions.
- Instantiation heuristic based on:

  What’s Decidable About Arrays?,
  A. R. Bradley, Z. Manna, and H. B. Sipma, VMCAI’06.
Yices may return *unknown* for quantified formulas.

The model should be interpreted as a “potential model”.

Tuning egraph matching:

- `-mi <num>` – Maximum number of quantifier instantiations.
- `-mp <num>` – Maximum number of patterns per quantifier.
- `-pc <num>` – Pattern generation heuristic (0: liberal, 2: conservative).

Advice: try conservative setting first.
Quantifiers: example

```
./yices q.ys

(define f::(-> int int))
(define g::(-> int int))
(define a::int)
(assert (forall (x::int) (= (f x) x)))
(assert (forall (x::int) (= (g (g x)) x)))
(assert (/= (g (f (g a))) a))
(check)

→ unsat
```
C API

- Yices distribution comes with a C library.

- Two different APIs:
  - `yices_c.h`
  - `yicesl_c.h` (Lite version).
Conclusion

- Yices is an efficient and flexible SMT solver.
  - Yices supports all theories in SMT-LIB and much more.
  - It is being used in SAL, PVS, and CALO.
- Yices is not ICS.
- Yices is freely available for end-users.
  - [http://yices.csl.sri.com](http://yices.csl.sri.com)
- Supported Platforms:
  - Linux
  - Windows: Cygwin & MinGW
  - Mac OSX