ACL2 Workshop 2025

An ACL2s Interface to Z3

Andrew T. Walter, Panagiotis Manolios

Khoury College of Computer Science, Northeastern University

Z3

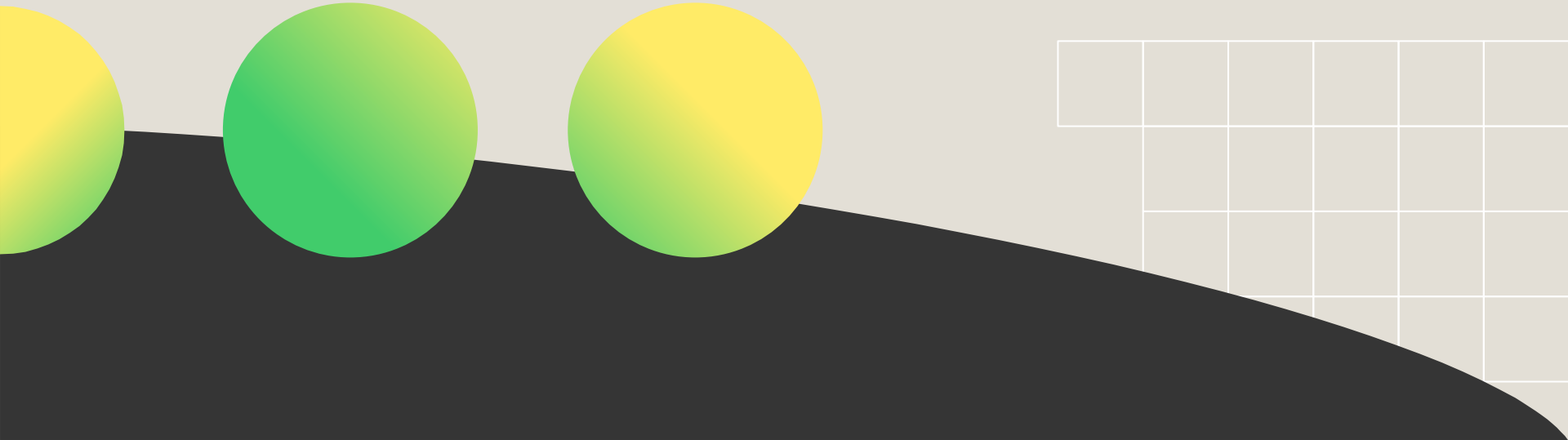
- Z3 is a Satisfiability Modulo Theories (SMT) solver
  - SMT: SAT over constraints in many-sorted FOL with equality wrt. theories
  - Decision procedures for several fragments, + support for undecidable
- SMT-LIB2: standardized S-expression input format

```
(declare-const a Int)
(declare-const b Real)
(declare-const c Int)
(assert (> a (+ b 2)))
(assert (= a (+ (* 2 c) 10)))
(assert (<= (+ c b) 1000))
(check-sat)
(get-model)
```

```
sat
(
  (define-fun b () Real
    0.0)
  (define-fun c () Int
    0)
  (define-fun a () Int
    10)
)
```

Why Z3 + ACL2?

Z3 and ACL2 have different strengths - can work together synergistically!



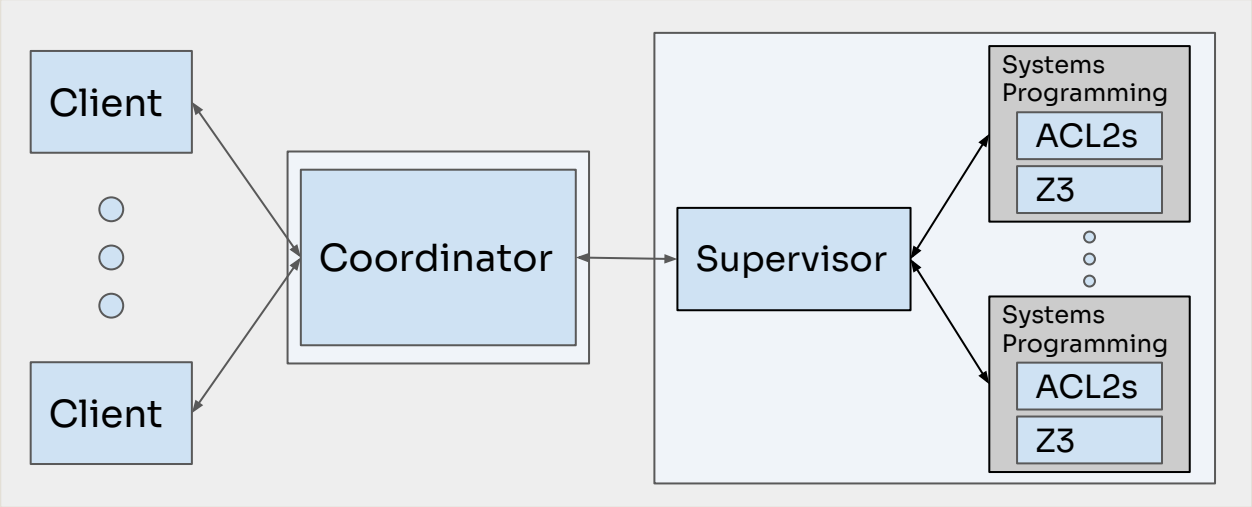
Demo

Related Work

- UCLID/BAT + ACL2 (Manolios & Srinivasan, 2004, 2005...)
  - Hardware verification
- Yices + ACL2 (Srinivasan, 2007), Satlink (Davis & Swords, 2011), Smtlink (Peng & Greenstreet, 2015)
  - Automatically convert ACL2 goals into SAT/SMT queries
  - Solver proves => ACL2 goal is a theorem (trust)
  - Solver disproves => counterexample
  - Soundness is important - trusting result of external tool + translation!
  - Yices + ACL2: QF_AUFLIA
  - Satlink: bit-blast Boolean formulae
  - Smtlink: linear and nonlinear arithmetic, user-defined data types

ACL2s Systems Programming

- Framework enabling the development of tools that use ACL2 as a black box
  - Gamified loop invariant discovery, proof checker, Python fuzzing…
- Our library adds Z3 to toolbox!
  - Framework for interfacing with reasoning tools, interacting with external world



Lisp-Z3 Library

- Goals: fast, flexible, exposes many features
  - Write constraints with SMT-LIB2 semantics
- Interface with Z3's C API
  - Fast, low overhead, high control
  - Engineering required to implement
- Written in Common Lisp
  - ASDF package
  - Accessible in ACL2 using defun-bridge
- Supports core SMT functionality (assert, check SAT, get model), plus:
  - Optimization, uninterpreted functions, user-defined sorts, quantifiers...
  - Incremental solving, solver configuration

Usage

```
;; Set up Z3. Only needs to happen once, before other code that uses Z3
(solver-init)
;; Declare variables x and y
(declare-const x Bool)
(declare-const y Int)
;; Assert a constraint over x and y
(z3-assert
  (and x (>= y 5)))
;; Check for satisfiability
(check-sat)
;; If satisfiable, get a satisfying assignment
(get-model)
```

```
#<Z3::MODEL
X -> true
Y -> 5
>
```

Usage

```
;; Set up Z3. Only needs to happen once, before other code that uses Z3
(solver-init)
;; Declare variables x and y and assert a constraint over them
(z3-assert (x :bool y :int)
  (and x (>= y 5)))
;; Check for satisfiability
(check-sat)
;; If satisfiable, get a satisfying assignment and translate it into a
;; form that is usable as Common Lisp let bindings
(get-model-as-assignment)
```

```
((X T) (Y 5))
```

Implementation: Low-Level Interface

- Use the Common Foreign Function Interface (CFFI) Common Lisp library
- Engineering effort: expose correct types and functions
- Can call Z3 C API functions, but verbose & requires memory management…

```
(let ((model (z3-solver-get-model ctx slv)))
  (loop for i below (z3-model-get-num-consts ctx model)
        for decl = (z3-model-get-const-decl ctx model i)
        for name = (z3-get-symbol-string ctx (z3-get-decl-name ctx decl))
        for value-ast = (z3-model-get-const-interp ctx model decl)
        ;; Here we assume the value is a numeral and get it as a string
        collect (list name (z3-get-numeral-string ctx value-ast))))
```

Implementation: High-Level Interface

- Build on top of low-level interface to make interaction user-friendly!
- Provide wrapper types for Z3 objects
  - Pretty-printing, automatic memory management
- Handle assertion stack: variables & constraints relative to stack level
  - Stack levels can be pushed/popped
- Translate S-expressions into Z3 ASTs
- Allow users to define sorts - enums, tuples
- Translate models into Common Lisp values

Application: Sudoku

- Sudoku: classic SMT problem
- Sudoku solver ~50 LOC using Lisp-Z3
- Straightforward to add pretty-printing
- Incremental solving is convenient here!

```
+-------+-------+-------+
| 6 _ _ | 3 _ 1 | _ 8 4 |
| _ _ _ | _ 6 9 | _ _ 7 |
| _ _ _ | _ _ 7 | _ 1 3 |
+-------+-------+-------+
| 4 _ _ | 6 9 _ | _ _ 8 |
| _ _ _ | _ 1 5 | _ _ _ |
| _ _ 8 | _ _ _ | _ 6 _ |
+-------+-------+-------+
| _ _ _ | _ _ _ | _ _ _ |
| _ _ _ | 1 _ _ | 7 _ _ |
| 2 _ 4 | _ _ 3 | 1 _ _ |
+-------+-------+-------+
```

```
+-------+-------+-------+
| 6 7 2 | 3 5 1 | 9 8 4 |
| 8 3 1 | 4 6 9 | 2 5 7 |
| 5 4 9 | 2 8 7 | 6 1 3 |
+-------+-------+-------+
| 4 1 5 | 6 9 2 | 3 7 8 |
| 7 6 3 | 8 1 5 | 4 2 9 |
| 9 2 8 | 7 3 4 | 5 6 1 |
+-------+-------+-------+
| 1 5 7 | 9 4 6 | 8 3 2 |
| 3 9 6 | 1 2 8 | 7 4 5 |
| 2 8 4 | 5 7 3 | 1 9 6 |
+-------+-------+-------+
```

```
;; Turn an index into a Sudoku grid into the variable for that square's value
(defun idx-to-cell-var (idx)
  (intern (concatenate 'string "C" (write-to-string idx))))

;; We'll encode the sudoku grid in the simplest way possible, 81 integers
(defconstant +cell-vars+
  (loop for idx below 81 append (list (idx-to-cell-var idx) :int)))

;; We limit the integers to values between 1 and 9, inclusive
(defconstant cell-range-constraints
  (loop for idx below 81
        append `((<= 1 ,(idx-to-cell-var idx)) (>= 9 ,(idx-to-cell-var idx)))))

;; The values in each row must be distinct
(defconstant row-distinct-constraints
  (loop for row below 9 collect
    `(distinct ,@(loop for col below 9 collect (idx-to-cell-var (+ (* 9 row) col))))))

;; The values in each column must be distinct
(defconstant col-distinct-constraints
  (loop for col below 9 collect
    `(distinct ,@(loop for row below 9 collect (idx-to-cell-var (+ (* 9 row) col))))))

;; The values in each 3x3 box must be distinct
(defconstant box-distinct-constraints
  ;; indices of the top-left square of each box
  (loop for box-start in '(0 3 6 27 30 33 54 57 60)
        collect `(distinct
                  ;; offsets of each square in a box from the top left square
                  ,@(loop for box-offset in '(0 1 2 9 10 11 18 19 20)
                          collect (idx-to-cell-var (+ box-start box-offset))))))

;; Set up the initial constraints on the grid
(defun init ()
  (solver-init)
  (z3-assert-fn +cell-vars+ (cons 'and cell-range-constraints))
  (z3-assert-fn +cell-vars+ (cons 'and row-distinct-constraints))
  (z3-assert-fn +cell-vars+ (cons 'and col-distinct-constraints))
  (z3-assert-fn +cell-vars+ (cons 'and box-distinct-constraints)))

;; This generates constraints based on a "starting grid".
(defun input-grid-constraints (grid)
  (loop for entry in grid for idx below 81
        when (not (equal entry '_)) collect `(= ,(idx-to-cell-var idx) ,entry)))

(defun solve-grid (input-grid)
  (solver-push)
  (let ((input-cstrs (input-grid-constraints input-grid)))
    (when input-cstrs (z3-assert-fn +cell-vars+ (cons 'and input-cstrs))))
  (let* ((sat-res (check-sat))
         (res (if (equal sat-res :sat) (get-model-as-assignment) sat-res)))
    (progn (solver-pop) res)))
```

Demo

Ankit Kumar and Panagiotis Manolios. "Mathematical Programming Modulo Strings." *FMCAD 2021.*

Application: String Solving

- String solving: constraint solving
  - Security analysis, program verification, ...
- SeqSolve (Kumar & Manolios, 2021):
  - Uses Lisp-Z3 for LIA constraints
  - Leverages Z3's incremental solving
  - Written in ACL2s, uses Lisp-Z3
- Beat all other solvers at time of publishing
  - Solved more problems than any other solver
  - Faster than any other solver
  - Solved problems that no other solver could! Ex:

$$xcyczvycya = yacwazvbux$$

A string equation over variables x and y, with constants a and b

$$xab = ay$$

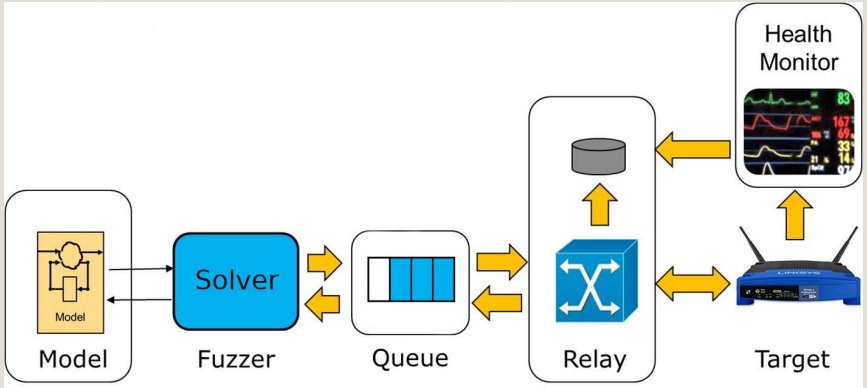
One possible solution

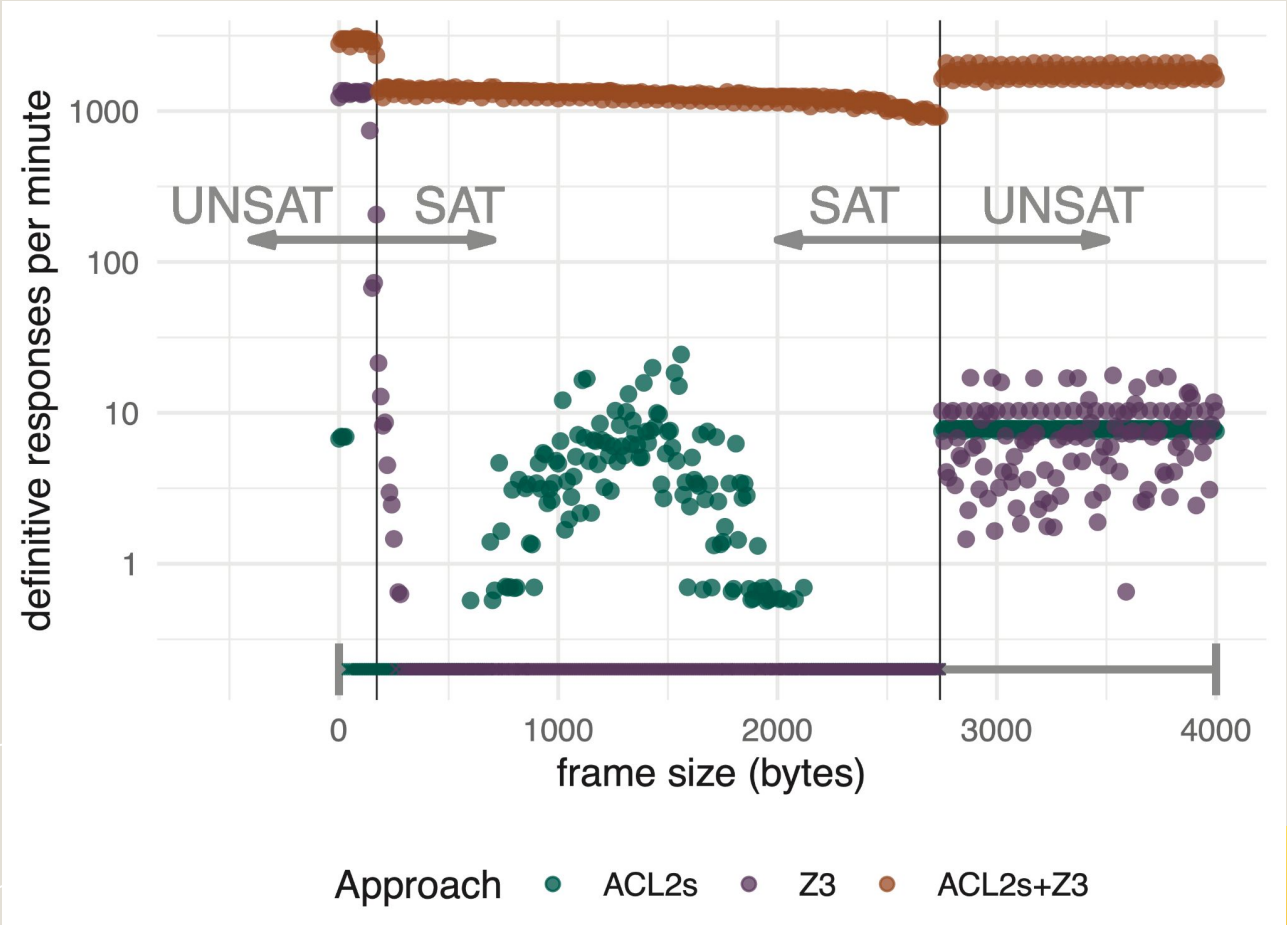
$$x = a$$
$$y = ab$$

Andrew T. Walter, David A. Greve and Panagiotis Manolios. "Enumerative Data Types with Constraints." *FMCAD 2022.*

Application: Wi-Fi Fuzzing

- Work with Greve and Manolios, 2022:
  - Generate frames for hardware-in-the-loop fuzzing
  - Model 802.11 Wi-Fi frames using ACL2s types
- Frames are complicated: many constraints on size + contents
- Hard to generate frames with good distribution of sizes
- Use of ACL2s + Lisp-Z3 to generate frames w/ a variety of sizes
  - Faster than Z3 or ACL2s alone!




definitive responses per minute
1000
100
10
1
UNSAT
SAT
SAT
UNSAT
0
1000
2000
3000
4000
frame size (bytes)
Approach
ACL2s
Z3
ACL2s+Z3

Conclusion & Future Work

- Lisp-Z3: a library for using Z3 as a service from Common Lisp
  - Integrated with ACL2s systems programming framework
  - Shown useful in: string solving, fuzzing...
  - Freely available in the ACL2 books
- Future Work
  - Support more Z3/SMT-LIB2 features
  - Integration between Z3 sorts and ACL2s defdata types
  - Backend support for other SMT solvers (e.g., CVC5)
- We'd love for others to use Lisp-Z3!
  - If you try it out, please send us your feedback and feature requests!

Any questions? Ask away!

Think of something later? Wanna talk about using Lisp-Z3? Feel free to contact me at walter.a@northeastern.edu!

https://github.com/acl2/acl2/tree/master/books/workshops/2025/walter-manolios

Thanks to the anonymous reviewers of our paper, Dave Greve & the folks at Collins we worked with, Ankit Kumar, and the students in the Fall 2021 and 2022 sections of CS4820 at Northeastern University