Computer-Aided Reasoning for Software

Solver-Aided Programming I

Emina Torlak
emina@cs.washington.edu
Topics

What is this course about?

Course logistics

Getting started with solver-aided programming!
Tools for building better software, more easily
Tools for building better software, more easily

more reliable, efficient, secure
Tools for building better software, more easily

automated verification and synthesis based on satisfiability solvers
Tools for building **better software**, more easily

- **automated verification and synthesis based on satisfiability solvers**
- **“solver-aided tools”**
By the end of this course, you’ll be able to build solver-aided tools for any domain!
By the end of this course, you’ll be able to build solver-aided tools for any domain!
logistics

Topics, structure, people
Course overview

program question

logic

automated reasoning engine
Course overview

program question

verifier, synthesizer

logic

SAT, SMT, model finders
Course overview

program → question

→ verifier, synthesizer

logic → SAT, SMT, model finders

Drawing from “Decision Procedures” by Kroening & Strichman
Course overview

program question

verifier, synthesizer

logic

SAT, SMT, model finders

Drawing from "Decision Procedures" by Kroening & Strichman
Course overview

**program**  **question**

**verifier, synthesizer**

**logic**

**SAT, SMT, model finders**

Drawing from “Decision Procedures” by Kroening & Strichman
Grading

3 individual homework assignments (75%)

• conceptual problems & proofs (TeX)
• implementations (Racket, Dafny, Alloy)
• completed on your own (may discuss HWs with course staff only)

Course project (25%)

• build a computer-aided reasoning tool for a domain of your choice
• teams of 2-3 people
• see the course web page for timeline, deliverables and other details
Reading and references

Recommended readings posted on the course web page

• Complete each reading before the lecture for which it is assigned
• If multiple papers are listed, only the first is required reading

Recommended text books

• Bradley & Manna, *The Calculus of Computation*
• Kroening & Strichman, *Decision Procedures*
Advice for doing well in 507

Come to class (prepared)
  • Lecture slides are enough to teach from, but not enough to learn from

Participate
  • Ask and answer questions

Meet deadlines
  • Turn homework in on time
  • Start homework and project sooner than you think you need to
  • Follow instructions for submitting code (we have to be able to run it)
  • No proof should be longer than a page (most are ~1 paragraph)
People

Emina Torlak
PLSE
CSE 596

Sorawee Porncharoenwase
PLSE
CSE2 253
OH W 1-2pm
People

Emina Torlak
PLSE
CSE 596
By appointment

Sorawee Porncharoenwase
PLSE
CSE2 253
OH W 1-2pm
Solver-aided programming in two parts: (1) getting started and (2) going pro
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming** in two parts: (1) getting started and (2) going pro
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) getting started and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**

1. Getting started
2. Going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Solver-aided programming in two parts:
(1) getting started and (2) going pro

A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

Solver-aided programming in two parts: (1) **getting started** and (2) going pro

How to use a solver-aided language: the **workflow**, constructs and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Classic programming: from spec to code

specification

P(x) {
  ...
  ...
}

Classic programming: test behaviors

test some behaviors against the specification

```p(x) {
  ...
  ...
}
assert safe(2, P(2))```
Solver-aided programming: query behaviors

query all behaviors against the specification

```
P(x) {
    ...
    ...
} assert safe(x, P(x))
```

Symbolic value $x$ stands for an arbitrary integer.
Find an input on which the program fails.

P(x) {
  ...
  ...
} assert safe(x, P(x))
Find an input on which the program fails.
Find values that repair the failing run.

$$\exists x . \neg \text{safe}(x, P(x))$$
$$x = 42 \land \text{safe}(x, P(x))$$

Solver-aided programming: solve
Find an input on which the program fails.
Find values that repair the failing run.
Find code that repairs the program.

assert safe(x, P(x))

∃x . ¬safe(x, P(x))
x = 42 ∧ safe(x, P(x))
∃e. ∀x. safe(x, P_e(x))
Solver-aided programming: workflow

Use **assertions**, **assumptions**, and **symbolic values** to express the specification.

Ask **queries** about program behavior (on symbolic inputs) with respect to the specification.
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**

(1) **getting started** and (2) **going pro**

How to use a solver-aided language: the workflow, **constructs, and gotchas.**

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.
Rosette extends Racket with solver-aided constructs

= +

- (define-symbolic id type)
- (define-symbolic* id type)
- (assert expr)
- (assume expr)
- (verify expr)
- (solve expr)
- (synthesize #:forall expr
#:guarantee expr)

- symbolic values
- assertions
- assumptions
- queries
Rosette extends Racket with solver-aided constructs

“A programming language for creating new programming languages”

A modern descendent of Scheme and Lisp with powerful macro-based meta programming.

(symbolic values)
(assertions)
(assumptions)
(queries)

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(assume expr)
(verify expr)
(solve expr)
(synthesize
  #:forall expr
  #:guarantee expr)
Rosette extends **Racket** with solver-aided constructs

```
#lang rosette

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(assume expr)
(verify expr)
(solve expr)
(synthesize #:forall expr #:guarantee expr)
```

symbolic values
assertions
assumptions
queries
Rosette constructs by example

(define-symbolic id type)
(define-symbolic* id type)
(assert expr)
(assume expr)
(verify expr)
(solve expr)
(synthesize
   #:forall expr
   #:guarantee expr)

demo

Common pitfalls and gotchas

“A gotcha is a valid construct in a system, program or programming language that works as documented but is counter-intuitive and almost invites mistakes because it is both easy to invoke and unexpected or unreasonable in its outcome.”

—Wikipedia

Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer \( k > 0 \) or \#f for approximate or precise reasoning, respectively.
Common pitfalls and gotchas: reasoning precision

Reasoning precision
Unbounded loops
Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer \( k > 0 \) or \#f for approximate or precise reasoning, respectively.

; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer $k > 0$ or $\#f$ for approximate or precise reasoning, respectively.

; default current-bitwidth is $\#f$
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

- Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
- Controlled by setting current-bitwidth to an integer \( k > 0 \) or \(#f\) for approximate or precise reasoning, respectively.

```scheme
; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
  (model [x 64])
> (verify (assert (not (= x 64))))
```
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.

• Controlled by setting current-bitwidth to an integer $k > 0$ or $\#f$ for approximate or precise reasoning, respectively.

; default current-bitwidth is $\#f$
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
  (model [x 64])
> (verify (assert (not (= x 64))))
  (model [x 64])
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
  
• Controlled by setting current-bitwidth to an integer $k > 0$ or #f for approximate or precise reasoning, respectively.

```scheme
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])

> (current-bitwidth 5)
> (solve (assert (= x 64)))
```
Common pitfalls and gotchas: reasoning precision

Reasoning precision

Unbounded loops

Unsafe features

• Determines if integers and reals are approximated using k-bit words or treated as infinite-precision values.
• Controlled by setting current-bitwidth to an integer k > 0 or #f for approximate or precise reasoning, respectively.

; default current-bitwidth is #f
> (define-symbolic x integer?)
> (solve (assert (= x 64)))
(model [x 64])
> (verify (assert (not (= x 64))))
(model [x 64])

> (current-bitwidth 5)
> (solve (assert (= x 64)))
(model [x 0])
> (verify (assert (not (= x 64))))
(model [x 0])
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.
Loops and recursion must be **bounded** (aka **self-finitizing**) by

- concrete termination conditions, or
- upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```scheme
(define (search x xs)
  (cond
    [(null? xs) #f]
    [(equal? x (car xs)) #t]
    [else (search x (cdr xs))]))

> (define-symbolic xs integer? #:length 5)
> (define-symbolic xl i integer?)
> (define ys (take xs xl))
> (verify
  (begin
    (assume (<= 0 i (- xl 1))
    (assert (search (list-ref ys i) ys)))))
```

**Unbounded loops**

**Unsafe features**

- Loops and recursion must be **bounded** (aka **self-finitizing**) by

**Reasoning precision**

Common pitfalls and gotchas: unbounded loops
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

```scheme
(define (search x xs)
  (cond
   [(null? xs) #f]
   [(equal? x (car xs)) #t]
   [else (search x (cdr xs))]))

> (define-symbolic xs integer? #:length 5)
> (define-symbolic xl i integer?)
> (define ys (take xs xl))
> (verify
   (begin
    (assume (<= 0 i (- xl 1))
    (assert (search (list-ref ys i) ys)))))

(unsat) Terminates because search iterates over a bounded structure.
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

(define (factorial n)
  (cond
   [(= n 0) 1]
   [else (* n (factorial (- n 1)))]))
Common pitfalls and gotchas: unbounded loops

Reasoning precision
Unbounded loops
Unsafe features

• Loops and recursion must be *bounded* (aka *self-finitizing*) by
  • concrete termination conditions, or
  • upper bounds on size of iterated (symbolic) data structures.
• Unbounded loops and recursion run forever.

(define (factorial n)
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1)))]))

> (define-symmetric k integer?)
> (solve
  (assert (> (factorial k) 10)))
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

• Loops and recursion must be bounded (aka self-finitizing) by
  • concrete termination conditions, or
  • upper bounds on size of iterated (symbolic) data structures.
• Unbounded loops and recursion run forever.

(define (factorial n)
  (cond
   [(= n 0) 1]
   [else (* n (factorial (- n 1)))]))

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k) 10)))

Unbounded because factorial termination depends on k.
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

• Loops and recursion must be bounded (aka self-finitizing) by
  • concrete termination conditions, or
  • upper bounds on size of iterated (symbolic) data structures.

• Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be **bounded** (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```scheme
(define (factorial n g)
  (assert (>= g 0))
  (cond
    [(= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))])
)
```

```scheme
> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 3) 10)))
```
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

(define (factorial n g)
  (assert (>= g 0))
  (cond
    [ (= n 0) 1]
    [else (* n (factorial (- n 1) (- g 1)))]))

> (define-symbolic k integer?)
> (solve
  (assert (> (factorial k 3) 10)))

(unsat)

UNSAT because the bound is too small to find a solution.
Common pitfalls and gotchas: unbounded loops

Reasoning precision

Unbounded loops

Unsafe features

- Loops and recursion must be bounded (aka self-finitizing) by
  - concrete termination conditions, or
  - upper bounds on size of iterated (symbolic) data structures.
- Unbounded loops and recursion run forever.

Bound the recursion with a concrete guard.

```scheme
(define (factorial n g)
  (assert (>= g 0))
  (cond
   [(= n 0) 1]
   [else (* n (factorial (- n 1) (- g 1)))])
)
```

> (define-symbolic k integer?)
> (solve
>   (assert (> (factorial k 4) 10)))

(model
  [k 4])

Make sure the bound is large enough …
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in \texttt{#lang rosette/safe}

• Unlifted constructs can be used in \texttt{#lang rosette} but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in \#lang rosette/safe

- Unlifted constructs can be used in \#lang rosette but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

;; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

• Rosette *lifts* only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`

• Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

```scheme
; vectors are lifted
>(define v (vector 1 2))
>(define-symbolic k integer?)
>(vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2)))
```
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

• Rosette *lifts* only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`

• Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

```racket
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
```
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`
- Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

```scheme
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
hash-ref: no value found for key
  key: k
```
Common pitfalls and gotchas: unsafe features

Reasoning precision

Unbounded loops

Unsafe features

- Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`
- Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

```scheme
; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (⊢ (= 0 k) 1) (⊢ (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
hash-ref: no value found for key key: k
> (hash-set! h k 3)
> (hash-ref h k)
```
Common pitfalls and gotchas: unsafe features

Reasoning precision
Unbounded loops
Unsafe features

• Rosette lifts only a core subset of Racket to operate on symbolic values. This includes all constructs in `#lang rosette/safe`

• Unlifted constructs can be used in `#lang rosette` but require care: the programmer must determine when it is okay for symbolic values to flow to unlifted code.

; vectors are lifted
> (define v (vector 1 2))
> (define-symbolic k integer?)
> (vector-ref v k)
(ite* (¬ (= 0 k) 1) (¬ (= 1 k) 2))

; hashes are unlifted
> (define h (make-hash '((0 . 1)(1 . 2))))
> (hash-ref h k)
hash-ref: no value found for key
key: k
> (hash-set! h k 3)
> (hash-ref h k)
3
A programming model that integrates solvers into the language, providing constructs for program verification, synthesis, and more.

**Solver-aided programming in two parts:**
(1) **getting started** and (2) going pro

How to use a solver-aided language: the workflow, constructs, and gotchas.

How to build your own solver-aided tool via direct symbolic evaluation or language embedding.

[emina.github.io/rosette/]
Summary

Today

• Course overview & logistics
• Getting started with solver-aided programming

Next lecture

• Going pro with solver-aided programming