Growing Solver-Aided Languages with ROSETTE

Emina Torlak & Rastislav Bodík

U.C. Berkeley
solver-aided domain-specific language

Solver-aided DSL (SDSL)

Noun

1. A high-level language in which partially implemented programs can be executed, verified, debugged and synthesized with the aid of a constraint solver.
programming ...

specification

P(x) {
  ...
}

assume pre(x)
programming ...

```plaintext
formula, input/ output pairs, traces, another program, ...
P(x) {
    ...
}
```
programming with a solver

```plaintext
assume pre(x)
P(x) {
    ...
}
assert post(P(x))
```

translate(…)

SAT/SMT solver
programming with a solver: code checking

Is there a valid input $x$ for which $P(x)$ violates the spec?

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

$\exists x . \text{pre}(x) \land \neg \text{post}(P(x))$

SAT/SMT solver

CBMC [Oxford], Dafny [MSR], Jahob [EPFL], Miniatur / MemSAT [IBM], etc.
programming with a solver: code checking

Is there a valid input $x$ for which $P(x)$ violates the spec?

```
assume pre(x)
P(x) {
  ...
}
assert post(P(x))
```

$\exists x . pre(x) \land \neg post(P(x))$

counterexample $x = 42$

SAT/SMT solver

model

CBMC [Oxford], Dafny [MSR], Jahob [EPFL], Miniatur / MemSAT [IBM], etc.
programming with a solver: localizing faults

Given $x$ and $x'$, what subset of $P$ is responsible for $P(x) \neq x'$?

```plaintext
assume pre(x)

P(x) {
  v = x + 2
  ... }

assert post(P(x))
```

$\text{pre}(x) \land \text{post}(x') \land x' = P(x)$
programming with a solver: localizing faults

Given \( x \) and \( x' \), what subset of \( P \) is responsible for \( P(x) \neq x' \)?

Assume \( \text{pre}(x) \)

\[
P(x) \{ 
\quad v = x + 2
\quad \ldots
\}
\]

Assert \( \text{post}(P(x)) \)

\[
\text{pre}(x) \land \text{post}(x') \land \ x' = P(x)
\]

SAT/SMT solver

MIN CORE / MAXSAT

repair candidates
programming with a solver: angelic execution

Given $x$, choose $v$ at runtime so that $P(x, v)$ satisfies the spec.

\begin{center}
\begin{align*}
\text{assume } & \text{pre}(x) \\
\text{P}(x) \{ & \quad v = \text{choose()} \\
& \quad \ldots \} \\
\text{assert } & \text{post}(\text{P}(x)) \\
\text{\exists v \ . \ } & \text{pre}(x) \land \\
& \text{post(P(x, v))}
\end{align*}
\end{center}
programming with a solver: angelic execution

Given $x$, choose $v$ at runtime so that $P(x, v)$ satisfies the spec.

$$\exists v . \text{pre}(x) \land \text{post}(P(x, v))$$

**assume** $\text{pre}(x)$

$$P(x) \{
\quad v = \text{choose}()
\quad \ldots
\}$$

**assert** $\text{post}(P(x))$

$v = 0, \ldots$

**trace**

**model**

**SAT/SMT solver**
programming with a solver: synthesis

Replace ?? with expression e so that \( P_e(x) \) satisfies the spec on all valid inputs.

\[
\begin{align*}
\text{assume } & \text{pre}(x) \\
\text{P(x)} & \{ \\
\quad v = ?? \\
\quad \ldots \} \\
\text{assert } & \text{post(P(x))}
\end{align*}
\]

\[
\exists e . \forall x . \text{pre(x) } \Rightarrow \text{post(P_e(x))}
\]
Replace ?? with expression e so that $P_e(x)$ satisfies the spec on all valid inputs.

Assume $\text{pre}(x)$

$P(x) \{ 
  v = x - 2 
  \ldots \}

Assert $\text{post}(P(x))$

$\exists e \cdot \forall x \cdot \text{pre}(x) \Rightarrow \text{post}(P_e(x))$

SAT/SMT solver

Expressions

Model

Comfusy [EPFL], Sketch [Berkeley / MIT]
but building solver-aided languages is hard ...

Each new SDSL created by careful custom compilation to constraints, requiring years of training and experience.
a solver-aided framework for building SDSLs

Implement a library or an interpreter for your SDSL, and get a synthesizer, verifier, debugger and angelic oracle for programs in that SDSL.
a tiny solver-aided extension of racket ...

\[
\text{top-level-form} = \text{general-top-level-form} \\
| (\#expression \ expr) \\
| (module \ id \ name-id \\
| \ (#plain-module-begin \\
| \ module-level-form ...)} \\
| (begin \ top-level-form ...) \\
| (begin-for-syntax \ top-level-form ...)
\]

\[
\text{module-level-form} = \text{general-top-level-form} \\
| (\#provide \ raw-provide-spec ...) \\
| (begin-for-syntax \ module-level-form ...)
\]

\[
\text{general-top-level-form} = \text{expr} \\
| (define-values \ (id ...) \ expr) \\
| (define-syntaxes \ (id ...) \ expr) \\
| (\#require \ raw-require-spec ...)
\]

\[
\text{expr} = \text{id} \\
| (\#plain-lambda \ formals \ expr ...+) \\
| (case-lambda \ (formals \ expr ...+) ...) \\
| (if \ expr \ expr \ expr) \\
| (begin \ expr ...+) \\
| (begin0 \ expr \ expr ...) \\
| (let-values \ \{\{id ...) \ expr\ ...\} \ expr ...+) \\
| (letrec-values \ \{\{id ...) \ expr\ ...\} \ expr ...+) \\
| (set! \ id \ expr) \\
| (quote \ datum) \\
| (quote-syntax \ datum) \\
| (with-continuation-mark \ expr \ expr \ expr) \\
| (\#plain-app \ expr ...+) \\
| (\#top . \ id) \\
| (\#variable-reference \ id) \\
| (\#variable-reference \ (#top . \ id)) \\
| (\#variable-reference)
\]

\[
\text{formals} = \ (id ...) \\
| \ (id ...+ . \ id) \\
| \ id
\]
... with a symbolic evaluator and compiler

- debug
- solve
- verify
- synthesize

SDSL + program → Rosette racket → solver

transform, evaluate & compile to constraints

Z3, KODKOD
... with a symbolic evaluator and compiler
... with a symbolic evaluator and compiler
Why a circuit language?

- A teaching aid
- An oracle for testing circuit transformations in SAT-based solvers
Why a circuit language?

- A teaching aid
- An oracle for testing circuit transformations in SAT-based solvers
rosette by example: an SDSL for circuits

Why a circuit language?

- A teaching aid
- An oracle for testing circuit transformations in SAT-based solvers
rosette by example: an SDSL for circuits

Why a circuit language?

- A teaching aid
- An oracle for testing circuit transformations in SAT-based solvers
∀ a, b, c, d. 
impl(a, b, c, d) ≡ spec(a, b, c, d)

Warm up

A classic DSL for testing and verification of circuits.
a sample tcl program

#lang s-exp tcl

(define-circuit (RBC-parity a b c d)
  (! (<> (<> a b) (<> c d))))

(define-circuit (AIG-parity a b c d)
  &&
  (! (&& (! (&& ! (&& a b)) (&& ! a) (! b))))
  (! (&& (! c) (! d)) (! (&& c d))))
(! (&& (! (&& ! a b)) (! (&& ! a) (! b))))
  (&& (! (&& ! c) (! d)) (! (&& c d))))

(verify-circuit AIG-parity RBC-parity)

A circuit is a procedure that works on boolean values.
#lang s-exp tcl

(define-circuit (RBC-parity a b c d) (! (<=> (<=> a b) (<=> c d))))

(define-circuit (AIG-parity a b c d) (&& (! (&& (! (! a) (! b))) (! (&& (! c) (! d)) (! (&& c d)))) (! (&& (! (! a) (! b))) (! (&& (! c) (! d))) (! (&& c d))))))

(verify-circuit AIG-parity RBC-parity)

> (RBC-parity #f #f #t #f)
#t

> (AIG-parity #f #f #t #f)
#t
#lang s-exp tcl

(define-circuit (RBC-parity a b c d)
  (! (⇒ (⇒ a b) (⇒ c d))))

(define-circuit (AIG-parity a b c d)
  (&&
   (! (&& (! (&& a b)) (&& (! a) (! b))))
   (! (&& (&& (! c) (! d)) (! (&& c d))))
   (! (&& (! (&& a b)) (! (&& (! a) (! b))))
      (&& (! (&& (! c) (! d))) (! (&& c d)))))))

(verify-circuit AIG-parity RBC-parity)
a sample tcl program

#lang s-exp tcl

(define-circuit (RBC-parity a b c d) (! (<> (<> a b) (<> c d))))

(define-circuit (AIG-parity a b c d) (&& (! (&& (! (&& a b)) (&& (! a) (! b)))) (! (&& (&& (! c) (! d)) (! (&& c d)))) (! (&& (&& (! a b)) (! (&& (! a) (! b)))) (&& (! (&& (! c) (! d))) (! (&& c d)))))

(verify-circuit AIG-parity RBC-parity)

Verifies equivalence of two n-ary circuit functions.
a shallow embedding of tcl in racket

```racket
#lang racket
(define-syntx-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))
```

```racket
(define-circuit (RBC-parity a b c d)
  (! (==> (==> a b) (==> c d))))
```

```racket
(define-circuit (AIG-parity a b c d)
  (! (==> (==> a b) (==> a c))
      (==> (==> b c) (==> c d))))
```

```racket
(verify-circuit AIG-parity RBC-parity)
```

```racket
(define-circuit (RBC-parity #f #f #t #f)
  #t)
```

```racket
(define-circuit (AIG-parity #f #f #t #f)
  #t)
```

```racket
(verify-circuit AIG-parity RBC-parity)
```

```racket
(define-circuit (RBC-parity #f #f #t #f)
  #t)
```

```racket
(define-circuit (AIG-parity #f #f #t #f)
  #t)
```

```racket
(verify-circuit AIG-parity RBC-parity)
```
a shallow embedding of tcl in racket

```
#lang racket

(define-syntax-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))

(define (! a b) (if a #f #t))
(define (&& ab) (if ab #f))
(define (|| ab) (if a #t b))
(define (<=>) (if ab (! b)))

(define (verify-circuit impl spec)
  (define n (procedure-aryspec))
  (for ([i (expt 2 n)])
    (define bits (for/list ([j n]) (bitwise-bit-set? i j)))
    (unless (eq? (apply impl bits) (apply spec bits))
      (error "failed on" bits)))

(provide ! && || <=>)
(define-circuit verify-circuit)
#%datum #%app #%module-begin #%top-interaction)
```
a shallow embedding of tcl in racket

```
#lang racket

(define-syntax-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))

(define (! a) (if a #f #t))
(define (&& a b) (if a b #f))
(define (|| a b) (if a #t b))
(define (<=> a b) (if a b (! b)))
```

```
(define (RBC-parity a b c d) (! (<=> (<=> a b) (<=> c d))))
```
a shallow embedding of tcl in racket

#lang racket

(define-syntax-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))

(define (! a) (if a #f #t))
(define (&& a b) (if a b #f))
(define (|| a b) (if a #t b))
(define (<=>) (if a b (! b)))

(define (verify-circuit impl spec)
  (define n (procedure-arity spec))
  (for ([i (expt 2 n)])
    (define bits (for/list ([j n]) (bitwise-bit-set? i j)))
    (unless (eq? (apply impl bits) (apply spec bits))
      (error "failed on" bits))))
a shallow embedding of tcl in racket

```
#lang racket

(define-syntax-rule
  (define-circuit (id in ...)) expr)
(define (define (id in ...)) expr)

(define (! a) (if a #f #t))
(define (&& a b) (if a b #f))
(define (!! a b) (if a #t b))
(define (== a b) (if a (! b))

(define (verify-circuit impl spec)
  (define n (procedure-arity spec))
  (for ([i (expt 2 n)])
    (define bits (for/list ([j n]) (bitwise-bit-set? i j)))
    (unless (eq? (apply impl bits) (apply spec bits))
      (error "failed on" bits))))

(provide
  ! && || == define-circuit verify-circuit

#%datum #%app #%module-begin #%top-interaction)
```
verifying circuits with tcl

#lang s-exp tcl

(define-circuit (RBC-parity a b c d)
  (! (<> (<>) a b) (<>) c d)))

(define-circuit (AIG-parity a b c d)
  (&&
   (! (&& (! (&& a b)) (&& (! a) (! b))))
   (! (&& (&& (! c) (! d)) (! (&& c d))))
   (! (&& (&& (! a b)) (! (&& (! a) (! b))))
     (&& (! (&& (! c) (! d))) (! (&& c d))))))

(verify-circuit AIG-parity RBC-parity)

failed on (#f #f #f #f)

> (RBC-parity #f #f #f #f)
#f

> (AIG-parity #f #f #f #f)
#t
verifying circuits with tcl

#lang s-exp tcl

(define-circuit (RBC-parity a b c d) (! (=> (=> a b) (=> c d))))

(define-circuit (AIG-parity a b c d) (&& (! (&& (! (&& a b)) (&& (! a) (! b)))) (! (&& (&& (! c) (! d)) (! (&& c d)))) (! (&& (&& (! a b)) (! (&& (! a) (! b)))) (&& (! (&& (! c) (! d))) (! (&& c d)))))

(verify-circuit AIG-parity RBC-parity)

⚠️ failed on (#f #f #f #f)

> (RBC-parity #f #f #f #f)
#f

> (AIG-parity #f #f #f #f)
#t

Where is the bug?

How to fix it?

Verify circuits with many inputs?
#lang racket

(define-syntax-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))

(define (! a) (if a #f #t))
(define (&& a b) (if a b #f))
(define (|| a b) (if a #t b))
(define (<=>) a b) (if a b (! b)))

(define (verify-circuit impl spec)
  (define n (procedure-arity spec))
  (for ([i (expt 2 n)])
    (define bits (for/list ([j n]) (bitwise-bit-set? i j)))
    (unless (eq? (apply impl bits) (apply spec bits))
      (error "failed on" bits)))

(provide
  ! & & | | <=>
  define-circuit verify-circuit
  #\%datum #\%app #\%module-begin #\%top-interaction)
porting tcl to rosette

```
#lang s-exp rosette

(define-syntax-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))

(define (! a)  (if a #f #t))
(define (&& a b) (if a b #f))
(define (|| a b) (if a #t b))
(define (<==> a b) (if a b (! b)))

(define (verify-circuit impl spec)
  (define n (procedure-arity spec))
  (for ([i (expt 2 n)])
    (define bits (for/list ([j n]) (bitwise-bit-set? i j)))
    (unless (eq? (apply impl bits) (apply spec bits))
      (error "failed on" bits)))

(provide
  ! && || <=> define-circuit verify-circuit

  #%datum #%app #%module-begin #%top-interaction)
```
tcl with solver-aided verification (tcl+)

```scheme
#lang s-exp rosette

(define-syntax-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))

(define (verify-circuit impl spec)
  (define input (symbolic-input spec))
  (evaluate input (verify (correct impl spec input))))

(define (symbolic-input spec)
  (for/list ([i (procedure-arity spec)])
    (define-symbolic* b boolean?)
    b))

(define (correct impl spec input)
  (assert (eq? (apply impl input) (apply spec input))))

(provide
  ! && || <=> define-circuit verify-circuit
  #%datum #%app #%module-begin #%top-interaction)
```
Creates a list of fresh symbolic boolean constants to use as input to the circuits.
tcl with solver-aided verification (tcl+)

```scheme
#lang s-exp rosette

(define-syntax-rule
  (define-circuit (id in ...) expr)
  (define (id in ...) expr))

(define (verify-circuit impl spec)
  (define input (symbolic-input spec))
  (evaluate input (verify (correct impl spec input))))

(define (symbolic-input spec)
  (for/list ([i (procedure-arity spec)])
    (define-symbolic* b boolean?)
    b))

(define (correct impl spec input)
  (assert (eq? (apply impl input) (apply spec input))))

(provide
  ! && || <=> define-circuit verify-circuit
  #%datum #%app #%module-begin #%top-interaction)
```

(verify expr) searches for an interpretation of symbolic constants that causes expr to fail.
verifying circuits with tcl+

#lang s-exp tcl+

(define-circuit (RBC-parity a b c d)
    (! (<=> (<=> a b) (<=> c d)))))

(define-circuit (AIG-parity a b c d)
    (&&
        (! (&& (! (&& a b)) (&& (! a) (! b))))
        (! (&& (&& (! c) (! d)) (! (&& c d)))))
    (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
            (&& (! (&& (! c) (! d))) (! (&& c d)))))))

(verify-circuit AIG-parity RBC-parity)
debugging circuits with tcl+ (desiderata)

#lang s-exp tcl+

(define-circuit (RBC-parity a b c d) (! (==> (==> a b) (==> c d))))

(define/debug (AIG-parity a b c d) (&& (! (&& (! (&& a b)) (&& (! a) (! b)))) (! (&& (&& (! c) (! d)) (! (&& c d)))))))

(debug-circuit AIG-parity RBC-parity '(#f #t #t #t))

Why does the implementation differ from the specification on this input?
extending tcl+ with solver-aided debugging

```lisp
(requires
  rosette/lang/debug
  rosette/lib/tools/render)

(define (debug-circuit impl spec input)
  (render
   (debug [boolean?]
     (correct impl spec input))))

(provide debug-circuit define/debug quote)
```

`(debug [t] expr)` computes a minimal set of expressions of type `t` that are responsible for the failure in `expr`. 
debugging circuits with tcl+ (a minimal core)

```tcl+
#lang s-exp tcl+

(define-circuit (RBC-parity a b c d)
  (! (<=> (<=> a b) (<=> c d))))

(define/debug (AIG-parity a b c d)
  (&&
   (! (&& (! (&& a b)) (&& (! a) (! b))))
   (! (&& (&& (! c) (! d)) (! (&& c d)))))
   (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
   (&& (! (&& (! c) (! d))) (! (&& c d)))))

(debug-circuit AIG-parity RBC-parity '(#f #t #t #t))
```

Non-core (grayed-out) expressions are irrelevant to the failure: there is no way to fix the behavior of AIG-parity on the sample input by editing these expressions.
synthesizing circuits with tcl+ (desiderata)

#lang s-exp tcl+

(define-circuit (RBC-parity a b c d)
 (&&! (== (== a b) (== c d))))

(define-circuit (AIG-parity a b c d)
 (! (&& (&& (! (! a b)) (! (&& (! a) (! b))))
      (&& (! (&& (! c) (! d))) (! (&& c d)))))))

(synthesize-circuit AIG-parity RBC-parity)

A sketch of the desired repair.

Complete the sketch so that the given circuits are equivalent on all inputs.
extending tcl+ with solver-aided synthesis

32 (require rosette/lib/meta/meta)
34 (define (synthesize-circuit impl spec)
35   (define input (symbolic-input spec))
36   (generate-forms
37     (synthesize #:forall input
38       #:guarantee (correct impl spec input)))))
39
40 (define-synthax (Circuit [op1 op2 ...] expr ... #:depth d)
41   #:assert (>= d 0)
42   (choosing op1 identity)
43     (choosing
44       expr ...
45         (choosing op2 ...
46           (Circuit [op1 op2 ...] expr ... #:depth (- d 1))
47           (Circuit [op1 op2 ...] expr ... #:depth (- d 1)))))
49 (provide synthesize-circuit Circuit (for-syntax #%datum))
extending tcl+ with solver-aided synthesis

```
(require rosette/lib/meta/meta)

(define (synthesize-circuit impl spec)
  (define input (symbolic-input spec))
  (generate-forms
   (synthesize #:forall input
                #:guarantee (correct impl spec input)))

(define-synthax (Circuit [op1 op2 ...] expr ... #:depth d)
  #:assert (>= d 0)
  ([choose op1 identity]
   [choose
    expr ...
     ([choose op2 ...]
      (Circuit [op1 op2 ...] expr ... #:depth (- d 1))
      (Circuit [op1 op2 ...] expr ... #:depth (- d 1))))))

(provide synthesize-circuit Circuit (for-syntax #%datum))
```

To construct a circuit, choose a unary operator (or identity) and apply it either to one of the provided terminal expressions or to a circuit constructed using one of the given binary operators.
synthesizing circuits with tcl+ (the repair)

#lang s-exp tcl+

(define-circuit (RBC-parity a b c d)
  (! (<=> (<=> a b) (<=> c d))))

(define-circuit (AIG-parity a b c d)
  (&&
   (! (&& (! (&& a b)) (! (&& (! a) (! b))))
    (! (&& (! (&& c d)) (! (&& (! c) (! d)))))
    (! (&& (&& (! a) (! b))))
    (&& (! (&& (! c) (! d))) (! (&& c d)))))))

(synthesize-circuit AIG-parity RBC-parity)
stacking interpreters to synthesize a compiler

\[ \exists \text{ } \forall \text{ } \text{RBC} \text{ . } \text{RBC} \equiv \text{rax}(\text{RBC}) \]
stacking interpreters to synthesize a compiler

Deep SDSL embedding

$\exists \forall \text{RBC} \cdot \text{RBC} \equiv \text{rax}(\text{RBC})$

Synthesized a rewriter from RBCs to AIGs that works for all RBCs of bounded size.
web, spatial programming, superoptimization
websynth: web scraping by demonstration

Websynth synthesizes a scraping script given a web page and a few examples of desired data, using a declarative SDSL (XPaths).

Implemented by two undergraduate students in a few weeks.
websynth: web scraping by demonstration
partitioning code & data for a low power chip

Instructions/Second vs Power

GreenArrays GA144 Processor

Figure by Per Ljung
partitioning code & data for a low power chip

**Instructions/Second vs Power**

- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core

Figure by Per Ljung
partitioning code & data for a low power chip

**Instructions/Second vs Power**

![Graph showing Instructions/Second vs Power for various processors. The GA144 Processor is marked with a green circle and arrow indicating better performance.]

**GreenArrays GA144 Processor**
- Stack-based 18-bit architecture
- 32 instructions
- 8 x 18 array of asynchronous cores
- No shared resources (cache, memory)
- Limited communication, neighbors only
- < 300 byte memory per core

Manual function partitioning: break functions up into a pipeline with a few operations per core.

Figure by Per Ljung

Drawing by Mangpo Phothilimthana
partitioning code & data for a low power chip

- high-level program
- per-core high-level programs
- per-core optimized machine code

new programming model
new approach using synthesis

Mangpo Phothilimthana
Nishant Totla
partitioning code & data for a low power chip

- high-level program
- per-core high-level programs
- per-core optimized machine code

partitioner

code generator

new programming model

new approach using synthesis

Mangpo Phothilimthana
Nishant Totla
partitioning code & data for a low power chip

one iteration of MD5 hash

optimal code partitioning (synthesized in 5 min)

256-byte mem per core
initial data placement specified
(define-fragment (fast-max x y)
  #:ensures
  (lambda (x y result)
    (= result (if (> x y) x y)))
  #:library
  (bvlib [{bvneg bvge bvand} 1]
    [{bvxor} 2]))

Gulwani et al. PLDI’11

(define (fast-max x y)
  (let* ([t1 (bvge x y)]
         [t2 (bvneg t1)]
         [t3 (bvxor y x)]
         [t4 (bvand t3 t2)]
         [t5 (bvxor y t4)])
    t5))
{emin, bodik}@eecs.berkeley.edu