Growing Solver-Aided Languages with ROSETTE

Emina Torlak Rastislav Bodik
U.C. Berkeley
{emina, bodik}@eecs.berkeley.edu

Abstract

SAT and SMT solvers have automated a spectrum of programming tasks, including program synthesis, code checking, bug localization, program repair, and programming with oracles. In principle, we obtain all these benefits by translating the program (once) to a constraint system understood by the solver. In practice, however, compiling a language to logical formulas is a tricky process, complicated by having to map the solution back to the program level and extend the language with new solver-aided constructs, such as symbolic holes used in synthesis.

This paper introduces ROSETTE, a framework for designing solver-aided languages. ROSETTE is realized as a solver-aided language embedded in Racket, from which it inherits extensive support for meta-programming. Our framework frees designers from having to compile their languages to constraints: new languages, and their solver-aided constructs, are defined by shallow (library-based) or deep (interpreter-based) embedding in ROSETTE itself.

We describe three case studies, by ourselves and others, of using ROSETTE to implement languages and synthesizers for web scraping, spatial programming, and superoptimization of bitvector programs.

Categories and Subject Descriptors D.2.2 [*Software Engineering*]: Design Tools and Techniques; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design; Languages

Keywords Solver-Aided Languages

1. Introduction

A few times in the evolution of programming languages, extra-linguistic advances enabled adoption of new programming constructs: efficient garbage collection led to automatic memory management; transactional hardware gave rise to atomics; and JIT compilation helped popularize dynamically typed languages. SAT and SMT solvers may be on the cusp of enabling another class of language constructs. Reduction to constraint solving has already automated a spectrum of programming tasks—including program checking, program synthesis, and program grading—and more problems previously solved with tricky deterministic algorithms may follow suit. This versatility of solvers encourages us to integrate solver capabilities into programming languages.

We argue that most applications of solvers in programming can be reduced to four elementary solver queries:

- (S) synthesizing a code fragment that implements a desired behavior [23, 34];
- (V) checking that an implementation satisfies a desired property [8, 10, 12, 15, 21, 37, 40];
- (L) localizing code fragments that cause an undesired behavior [22]; and
- (A) runtime nondeterminism, which asks an angelic oracle to divine values that make the execution satisfy a specification [23, 26, 28, 32].

An advanced application may combine several of these queries. For example, a solver-aided tool for program repair [7, 41] might first localize repair candidates (L), then replace a candidate code fragment with an angelically chosen value that makes the program pass a failing test (A), and finally, if the chosen value heuristically appears like a value that a correct program would produce, the repair tool might use it to synthesize a replacement for the candidate code fragment (S).

In principle, these queries can be supported by translating the program (once) to constraints and invoking the solver on these constraints. Depending on the context, the solver searches for a desired value or program, acting as a bidirectional program interpreter (in S and A); falsifies a formula, acting as a verifier (V); or produces an unsatisfiable core, acting as a fault localizer (L).

Existing systems have demonstrated the benefits of using a solver to answer individual queries in the context of a specific language. Extending these benefits to other languages

Onward! 2013, October 29–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-2472-4/13/10/13/10. . . $15.00.
http://dx.doi.org/10.1145/2509578.2509586

requires development of a *symbolic compiler* from the new language to logic constraints. Compiling a language to logical constraints is a tricky process, additionally complicated by having to map the solution back to the program level and having to extend the language with solver-aided constructs, such as symbolic holes [34] that define the space of candidate programs considered by a synthesizer.

This paper describes ROSETTE, a framework for construction of *solver-aided domain-specific languages* (SDSLs). ROSETTE is a small extension of Racket [39] equipped with a symbolic compiler. Because ROSETTE is itself solver-aided, languages embedded into ROSETTE inherit the four queries, which frees language developers from having to write symbolic compilers and formulate low-level queries to a solver.

ROSETTE supports both shallow (library-based) and deep (interpreter-based) embedding of languages. One can even stack languages and synthesize an interpreter for the lower parts of the language stack. Along these lines, we show in Section 2 how ROSETTE synthesizes rewrite rules for a compiler of circuit programs. We believe that this is the first synthesis of rewrites that does not require prior semantic axioms such as proof rules [38].

The key design decision in ROSETTE is to compile only a small a subset of the host language (Racket) to constraints, and to grow this small *symbolic core* with a partial evaluator that simplifies non-core constructs prior to symbolic compilation. This architecture offers several new benefits:

- ROSETTE programs are free to use all Racket features, including its advanced object system and metaprogramming constructs such as macros, as long as these non-core constructs are partially-evaluated away prior to symbolic compilation. In our experiments, non-core constructs never leaked past the partial evaluator. Access to advanced language constructs thus eased construction of new languages without impacting the compilation to logic constraints.
- By dividing work between the partial evaluator and the symbolic compiler, we have enabled a simple implementation while offering a rich solver-aided language. Prior systems either symbolically compiled the entire language, leading to large implementations and/or limited languages [8, 12, 34], or compiled only a tiny core without any support for growing it [23].

While ROSETTE heavily relies on Racket's metaprogramming for both symbolic compilation and partial evaluation, ROSETTE's architecture can be easily realized in other languages too, as evidenced by recent work on embedding solver-aided features into Ruby [27] and Scala [23]. After all, ROSETTE only requires that the operations in the symbolic core can be overridden or otherwise lifted to ROSETTE's custom versions. The operations outside the symbolic core remain unchanged.

Our group has used ROSETTE to implement several new solver-aided systems, including a programming model for ultra-low-power spatial architectures with synthesis-based program partitioning; a declarative SDSL for web scraping, with synthesis support for programming by demonstration; and a superoptimizer for bitvector programs. Two of these systems have been prototyped by senior undergraduate and first-year graduate students—all first-time users of ROSETTE—in just a few weeks. In contrast, the same students needed an entire semester to build prototypes of analogous SDSLs from scratch, without the help of ROSETTE. The key productivity benefits came from not having to build a symbolic compiler.

We present main ideas behind ROSETTE in Section 2, by showing how to grow a stack of tiny SDSLs with support for synthesis, verification, fault localization and angelic execution. We then present ROSETTE's semantics (Section 3), describe some of the systems designed with our framework (Section 4), and discuss related work (Section 5). Section 6 concludes the paper.

2. ROSETTE by Example

In this section, we illustrate how to construct solver-aided DSLs through embedding in a solver-aided host language. We demonstrate both shallow and deep embedding. Our host solver-aided language is ROSETTE, which is itself embedded in a (classical) host language Racket [39].

Shallow embedding is demonstrated with a tiny DSL for specifying boolean circuits (Section 2.1). Here, circuit programs are ordinary Racket functions, and we show that a Racket DSL can be equipped with solver-aided capabilities by a straightforward port to ROSETTE (Section 2.2).

Next, we embed the circuit language deeply, which facilitates development of circuit transformers, *i.e.,* programs that rewrite circuit programs. We show that it is possible to synthesize and debug circuit transformers by asking ROSETTE to reason across three linguistic layers: circuit programs, their transformers, and their interpreters.

We show two styles of deep embedding: a lightweight version, which adopts ROSETTE's symbolic representation of circuits as its abstract syntax tree (Section 2.3); and a full deep embedding, which constructs an explicit abstract syntax tree and symbolically quantifies over a space of trees, giving the transformation language the power to verify correctness of its programs on all circuits up to a bounded size (Section 2.4).

2.1 Getting Started: A Circuit DSL in Racket

Consider the problem of building a tiny circuit language (TCL) for specifying boolean circuits, testing them on input-output pairs, and verifying that two circuits are equivalent on all inputs. TCL may be used in teaching, where it can demonstrate alternative circuits for a given boolean function, or in developing a boolean constraint solver, where it can test that circuit transformations change the structure (representation) of a circuit but not its boolean function.

```
1  #lang s-exp tcl

3  (define-circuit (xor x y)
4    (! (<=> x y)))

6  (define-circuit (RBC-parity a b c d)
7    (xor (<=> a b) (<=> c d)))

9  (define-circuit (AIG-parity a b c d)
10   (&&
11    (! (&& (! (&& (! (&& a b)) (&& (! a) (! b))))
12           (! (&& (&& (! c) (! d)) (! (&& c d))))))
13    (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
14           (&& (! (&& (! c) (! d))) (! (&& c d)))))))

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

Figure 1. A sample program in a tiny circuit language (TCL)

Figure 1 shows a TCL program that verifies equivalence of two circuits, one presumably obtained by transforming the other. TCL is shallowly embedded in Racket and so TCL programs are valid Racket programs. TCL represents circuits as named first-class procedures that operate on boolean values and defines four built-in circuits (`!`, `<=>`, `&&`, and `||`), which are themselves Racket procedures. The language provides constructs for defining new circuits (`define-circuit`) and for verifying that two circuits are equivalent on all inputs (`verify-circuit`). Throughout this paper, we show DSL keywords in blue, and use boldface for keywords in the host language (be it ROSETTE or Racket).

The example program in Figure 1 verifies that a parity-checking circuit, `RBC-parity`, is equivalent to its transformed variant, `AIG-parity`. The original circuit takes the form of a Reduced Boolean Circuit (RBC) [2], and the transformed circuit is an And-Inverter Graph (AIG) [3]. RBCs represent boolean functions using negations and bi-implications. AIGs use only negations and conjunctions. Both representations were designed for use in solver-based tools [1, 6, 20].[1]

Figure 2 shows an implementation of TCL in Racket. Lines 3–4 export (only) the TCL functionality to TCL programs along with a few boilerplate Racket primitives that make TCL a stand-alone language. The `define-circuit` construct is simply rewritten into a procedure definition. This syntactic abstraction is implemented with a Racket macro: when the syntactic *pattern* on line 6 is found in a TCL program, it is rewritten into the *template* on line 7, which happens to be a code fragment in the Racket language. The pattern variables `id`, `in`, and `expr` are substituted in the process. For example, the circuit definition (`define-circuit` `(xor x y)` `(! (<=> x y))`) is rewritten into (**`define`** `(xor x y)` `(! (<=> x y))`). Ellipses in the macro allow the pattern variable `in` to match a variable number of arguments. The `verify-circuit` procedure (lines 9–14) checks the equivalence of two n-ary circuits by applying them to all possible combinations of n bits, and failing if they produce different outputs. The remaining procedures (lines 16–19) define the built-in circuits.

[1] They both also impose additional constraints on circuit structure, which we omit for the purposes of this paper.

```
1  #lang racket

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

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

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

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

Figure 2. A shallow embedding of TCL in Racket

Executing the TCL program in Figure 1 will attempt the verification. Since our `RBC-parity` and `AIG-parity` circuits are not equivalent, the verification fails on line 14 of Figure 2:

```
verification failed on (#f #f #f #f)

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

Because TCL is not solver aided, it misses several desirable features. First, the implementation of `verify-circuit` is inefficient. Based on exhaustive search, it will be slow on circuits with more than a few inputs. Second, TCL provides no automated support for localizing and fixing the bug in the `AIG-parity` circuit that was detected during verification. We add these features in the next subsection by developing TCL$^+$, a tiny circuit language that is solver-aided.

2.2 Getting Symbolic: A Circuit SDSL in ROSETTE

Our first step toward a solver-aided circuit language, TCL$^+$, is to embed TCL in the solver-aided ROSETTE language rather than in Racket. Using ROSETTE's symbolic values and *assertions* about program properties, we can formulate solver-aided *queries*. These queries will accelerate circuit verification as well as automatically locate and fix bugs in TCL$^+$ programs.

***Embedding a language in* ROSETTE** To embed TCL in ROSETTE, we replace the first line in Figure 2, namely the directive **`#lang`** `racket` that embeds TCL in Racket, with the following language declaration:

```
#lang s-exp rosette
```

With this change, our newly created SDSL, TCL$^+$, continues to work exactly as its precursor (although we are not yet exploiting any solver-aided features of ROSETTE). The example program in Figure 1 need not change at all, and its validity check still fails with the same error. Additionally, we will simplify Figure 2 by omitting lines 16–19 because they define procedures already provided by ROSETTE.

Symbolic constants To take advantage of the solver-aided queries that TCL$^+$ inherits from its host, we will need to introduce symbolic values into circuit programs:

```
> (define-symbolic b0 b1 b2 b3 boolean?)
```

This definition creates four symbolic constants of boolean type, and binds them to four Racket variables.

Symbolic constants can be used wherever concrete values of the same type can be used. For example, we can call a circuit procedure on our symbolic constants to obtain another symbolic value—a symbolic expression with those four constants in the leaves:

```
> (RBC-parity b0 b1 b2 b3)
(! (<=> (<=> b1 b0) (<=> b2 b3)))
```

ROSETTE actually provides two kinds of constructs for creating symbolic constants:

```
(define-symbolic id_1 ... id_k expr)
(define-symbolic* id_1 ... id_k expr)
```

The **`define-symbolic`** form creates k fresh symbolic constant of type *expr* and binds the provided program variables to their respective constants every time the form is evaluated (*e.g.,* in the body of a loop). But sometimes, it is desirable to bind a variable to a fresh symbolic value. This is accomplished by the **`define-symbolic*`** form, which creates k streams of fresh constants, binding each variable to the next constant from its stream whenever the form is evaluated. The following example illustrates the semantic difference:

```
  (define (static)
     (define-symbolic b boolean?)
     b)
> (define (dynamic)
     (define-symbolic* n number?)
     n)
> (eq? (static) (static))
#t
> (eq? (dynamic) (dynamic))
(= n$0 n$1)
```

Booleans and numbers (more specifically, finite precision integers) are the only kinds of symbolic constants supported by ROSETTE. But because they are embedded in a programming language, they can be used to create symbolic instances of other data types. We will see an example of this in Section 2.4, where we use primitive symbolic constants to create a symbolic expression that represents all (abstract syntax) trees of bounded depth. (The trick is to use symbolic constants to control what tree is produced by a tree constructor.)

Verification With symbolic values as inputs, and *assertions* that specify desired properties of circuits applied to those inputs, we have all the necessary components to implement the four basic solver-aided queries for TCL$^+$. For example, the following code accelerates verification of circuit programs (V) by querying the solver for an input on which a circuit and its transformation fail to produce the same output:

```
> (define counterexample
    (verify (assert (eq? (RBC-parity b0 b1 b2 b3)
                         (AIG-parity b0 b1 b2 b3)))))
> (evaluate (list b0 b1 b2 b3) counterexample)
'(#t #t #t #f)
> (RBC-parity #t #t #t #f)
#t
> (AIG-parity #t #t #t #f)
#f
```

The (**`verify`** *expr*) query exhibits the usual demonic semantics. It attempts to find a binding from symbolic constants to values that violates at least one of the assertions encountered during the evaluation of *expr*. Bindings are first-class values that can be freely manipulated by ROSETTE programs. We can also interpret any ROSETTE value with respect to a binding using the built-in `evaluate` procedure. In our example, the solver produces a binding that reveals an input (different from the one in Section 2.1) on which the transformed circuit, `AIG-parity`, fails to behave like the original circuit, `RBC-parity`.

Debugging The solver can help localize the cause of this faulty behavior (L) by identifying a maximal set of program expressions that are *irrelevant* to the failure—even if we replaced all such expressions with values provided by an angelic oracle, the resulting program would still violate the same assertion. ROSETTE finds irrelevant expressions by computing the complement set, which we will call a *minimal unsatisfiable core* of the failure. Core expressions are collectively responsible for an assertion failure, in the sense that the failed execution can be repaired by replacing just one core expression (in addition to all irrelevant expressions) with a call to an angelic oracle. In general, there may be many cores for each failure. Still, every core contains at least one buggy expression. In practice, examining one or two cores often leads to the source of the error.

To activate solver-aided debugging (which is off by default due to overhead), we select a few functions as candidates for core extraction by changing their definitions to use the keyword **`define/debug`** instead of **`define`**. In our example, we would change the definition of `AIG-parity` to use **`define/debug`** instead of `define-circuit`, and invoke the debugger as follows:

```
> (define core
    (debug [boolean?]
           (assert (eq? (AIG-parity #t #t #t #f)
                        (RBC-parity #t #t #t #f)))))
> (render core)
(define/debug (AIG-parity a b c d)
  (&&
   (! (&& (! (&& (! (&& a b)) (&& (! a) (! b))))
          (! (&& (&& (! c) (! d)) (! (&& c d))))))
   (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
          (&& (! (&& (! c) (! d))) (! (&& c d)))))))
```

The (**`debug`** [*predicate*] *expr*) query takes as input an expression whose execution leads to an assertion failure, and a predicate that specifies the dynamic type of the expressions

to be considered for inclusion in the core.[2] Cores are first-class values and can be used for further automation (such as program repair), or visualized using the built-in `render` procedure. Given a core, `render` displays all procedures marked with **`define/debug`**, showing the core expressions in red and the irrelevant expressions in gray.

Angelic execution We are going to repair the circuit program with program synthesis, but first, we want to identify suitable repair candidates, *i.e.,* subexpressions that we will replace with a synthesized repair expression. To look for repair candidates, we will heuristically restrict ourselves to the identified minimal unsatisfiable core. (In general, successful program repair may need to involve more than just the identified core, or even the union of all cores, because a program may need to be completely rewritten to implement the desired function in the given language.) The heuristic we will use is to select as the repair candidate the largest subexpression from the core that the synthesizer can handle. This will increase the chance that the repair corrects the program on all inputs (which is the correctness condition we will use in synthesis).

The core that we have obtained in our example suggests several repair candidates for `AIG-parity`. For example, it may be sufficient to fix just the subexpression `(&& a b)`, or we may have to replace the entire first child of the circuit with a new expression. To test the former hypothesis, we ask whether the failure can be removed by replacing the subexpression with an angelically chosen value. If not, repairing this subexpression alone will not be sufficient; we will need to select a larger repair candidate.

To create an oracle, we replace `(&& a b)` with a call to the `dynamic-choose` procedure, which generates fresh symbolic values,[3] and query the solver for a concrete interpretation of those values that saves our buggy execution from failing (A):

```
> (define (dynamic-choose)
    (define-symbolic* v boolean?)
    v)
> (define-circuit (AIG-parity a b c d)
    (&&
     (! (&& (! (&& (! (dynamic-choose)) (&& (! a) (! b))))
            (! (&& (&& (! c) (! d)) (! (&& c d))))))
     (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
            (&& (! (&& (! c) (! d))) (! (&& c d)))))))
> (solve (assert (eq? (AIG-parity #t #t #t #f)
                      (RBC-parity #t #t #t #f))))
solve error: no satisfying execution found
```

The (**`solve`** *expr*) query implements angelic semantics. It returns a binding from symbolic constants to concrete values, if any, that satisfies all assertions encountered during the evaluation of *expr*. We are using the streaming construct **`define-symbolic*`** to create symbolic constants so that the oracle can produce distinct values if it is consulted multiple times during the execution, as is the case in the next example, where the circuit with the oracle is evaluated twice. In this case, the solver is unable to find a satisfying binding, proving that it is not sufficient to fix just the subexpression `(&& a b)`.

To test our second repair hypothesis, we replace the entire first child of `AIG-parity` with a call to `dynamic-choose`. The solver is now able to prevent failures on both failing inputs that we have identified. Therefore, we have found a promising repair candidate:

```
> (define-circuit (AIG-parity a b c d)
    (&&
     (dynamic-choose)
     (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
            (&& (! (&& (! c) (! d))) (! (&& c d)))))))
> (solve
   (begin
     (assert (eq? (AIG-parity #t #t #t #f)
                  (RBC-parity #t #t #t #f)))
     (assert (eq? (AIG-parity #f #f #f #f)
                  (RBC-parity #f #f #f #f)))))
(model
 [dynamic-choose:v$0 #t]
 [dynamic-choose:v$1 #f])
```

The new **`solve`** query yields an angelic binding for the two symbolic constants generated by evaluating `AIG-parity`—and therefore, `dynamic-choose`—on our two counterexample inputs. For `AIG-parity` to work correctly on the first input, the first child expression should produce `#t`. For the second input, it should produce `#f`.

Synthesis With the first child of `AIG-parity` as the repair candidate, we can now synthesize (S) a correct replacement for that child from a syntactic *sketch* [34] of the desired repair. For example, the following sketch specifies that our repair is to be drawn from a grammar of `Circuit` expressions of depth $k \leq 3$, containing only the AIG operators and inputs to `AIG-parity`:

```
> (define-circuit (AIG-parity a b c d)
    (&&
     (Circuit [! &&] a b c d #:depth 3)
     (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
            (&& (! (&& (! c) (! d))) (! (&& c d)))))))
```

Figure 3 (lines 39-46) shows the definition of the `Circuit` grammar. We specify grammars with the help of two constructs: **`choose`**, which selects one of n expressions, and **`define-synthax`** (*i.e.,* "define synthesizable syntax"), which combines expression choices into a (recursive) grammar. The definition corresponds to the usual BNF specification:

$$\begin{array}{lll} \mathit{Circuit} & := & (\mathit{unop}\ (\mathit{expr}\ \ldots \mid (\mathit{binop}\ \mathit{Circuit}\ \mathit{Circuit}))) \\ \mathit{unop} & := & \mathit{op}_1 \mid \texttt{identity} \\ \mathit{binop} & := & \mathit{op}_2 \mid \ldots \mid \mathit{op}_k \end{array}$$

We instantiate this generic grammar by providing a set of circuit operators, a set of terminals, and an upper bound on the depth of circuit expressions drawn from the grammar.

Constructs for specifying grammars, such as **`choose`** and **`define-synthax`**, are provided by a small utility library built on top of ROSETTE, using macros and symbolic values. For example, the **`define-synthax`** form is implemented as a macro-generating macro. It creates a grammar (macro) from a pattern that specifies the syntax of grammar instantiations,

[2] If the predicate is too restrictive—for example, it rejects all values—or if the procedures selected for debugging are not causing the failure, the query will fail with the same assertion error as *expr*.

[3] Recall that the **`define-symbolic*`** form creates a stream of fresh constants for each declared variable.

a mandatory unrolling guard, and a template that specifies the body of the grammar. The unrolling guard helps ROSETTE determine when to stop expanding a given instantiation of the grammar. Our `Circuit` grammar guards the unrolling by decrementing an integer, but the guard can be any expression that evaluates to `#f` after a bounded number of unrollings. During the unrolling process, **`define-synthax`** introduces symbolic (boolean) constants to ensure that the expanded grammar captures all expressions of depth k or less, where k is the maximum number of unrollings allowed by the guard.

Given the sample sketch for repairing `AIG-parity`, we ask the solver to synthesize a repair as follows:

```
> (define model
    (synthesize
     #:forall    (list b0 b1 b2 b3)
     #:guarantee (assert (eq? (AIG-parity b0 b1 b2 b3)
                              (RBC-parity b0 b1 b2 b3)))))
> (generate-forms model)
(define-circuit (AIG-parity a b c d)
  (&&
   (! (&& (&& (! (&& d (! c))) (! (&& (! a) b)))
          (&& (! (&& c (! d))) (! (&& (! b) a)))))
   (! (&& (&& (! (&& a b)) (! (&& (! a) (! b))))
          (&& (! (&& (! c) (! d))) (! (&& c d)))))))
```

The synthesis query takes the form (**`synthesize #:forall`** *input* **`#:guarantee`** *expr*). Symbolic constants that do not appear in the *input* expression are called "holes" [34]. If successful, the query returns a binding from holes to concrete values that satisfies the assertions in *expr* for all possible bindings of the *input* constants. This corresponds to the classic formulation of synthesis [34] as a $\exists \vec{h} \forall \vec{i}. a_1(\vec{h}, \vec{i}) \wedge \ldots \wedge a_n(\vec{h}, \vec{i})$ problem, where $\vec{h}$ denotes the holes, $\vec{i}$ denotes the *input* constants, and $a_j(\vec{h}, \vec{i})$ is an assertion reached during the evaluation of *expr*.

In our example, $\vec{h}$ consists of the symbolic constants introduced by the instantiation of the `Circuit` grammar in the `AIG-parity` sketch. Each unrolling of the grammar introduces one fresh hole, and each (**`choose`** $e_1 \ldots e_n$) expression in the fully unrolled grammar introduces $n - 1$ fresh holes. A binding for these holes encodes a completion of the sketch. We produce a syntactic representation of the completed sketch with the help of the utility function `generate-forms`.

TCL$^+$ Figure 3 combines all of the facilities we have developed so far into a complete implementation of TCL$^+$. The language provides a convenient domain-specific interface for formulating solver-aided queries about circuit programs. It is also compatible with our old TCL implementation—any TCL program can be ported to TCL$^+$ by simply changing its **`#lang`** declaration to `tcl+`.

2.3 Getting Reflective: A Tiny Transformation SDSL

TCL$^+$ enables us to easily verify, debug and repair a circuit function against a reference implementation—for example, we repaired `AIG-parity` so that its behavior matches that of `RBC-parity`. But this does not fully address our original usage scenario, in which `AIG-parity` was the result of applying a circuit transformation procedure to `RBC-parity`. Ideally,

```
1  #lang s-exp rosette

3  (require rosette/lang/debug rosette/lib/tools/render
4           rosette/lib/meta/meta)

6  (provide (all-defined-out) ! && || <=> define/debug
7           #%datum #%app #%module-begin #%top-interaction
8           quote (for-syntax #%datum))

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

13 (define (dynamic-choose)
14   (define-symbolic* v boolean?)
15   v)

17 (define (symbolic-input spec)
18   (for/list ([i (procedure-arity spec)]) (dynamic-choose)))

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

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

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

30 (define (solve-circuit impl spec . inputs)
31   (solve (for ([input inputs]) (correct impl spec input))))

33 (define (synthesize-circuit impl spec)
34   (define input (symbolic-input spec))
35   (generate-forms
36     (synthesize #:forall input
37                 #:guarantee (correct impl spec input))))

39 (define-synthax (Circuit [op1 op2 ...] expr ... #:depth d)
40   #:assert (>= d 0)
41   ([choose op1 identity]
42    [choose
43     expr ...
44     ([choose op2 ...]
45      (Circuit [op1 op2 ...] expr ... #:depth (- d 1))
46      (Circuit [op1 op2 ...] expr ... #:depth (- d 1)))]))
```

Figure 3. TCL$^+$ in ROSETTE

we would like to detect and fix faults in the transformation procedure itself, not in a particular output of that procedure.

***Designing* TTL** Figure 4 shows an example program, implemented in a tiny transformation language (TTL), that demonstrates our original usage scenario. TTL extends TCL$^+$ with constructs for defining and implementing circuit transformation procedures, and with functions for formulating solver-aided queries about these procedures. A circuit transformer takes as input an abstract syntax tree (AST) that represents the body of a circuit procedure, and it produces another AST that represents the body of a functionally equivalent circuit. The `match` form matches an AST against a sequences of clauses. Each clause consists of a pattern whose free variables are bound in the body of the clause, if the match succeeds. The result of the first successful match is returned, and none of the remaining clauses are evaluated. The underscore pattern matches any value.

In our example, the `RAX` procedure takes as input (an AST representation of) an RBC, and transforms it into an AIG by recursive application of three rewrite rules. The first two

rules rewrite the `<=>` and `!` nodes. The last rule (line 16) leaves all other nodes unchanged, ensuring that `RAX` acts as the identity function on ASTs that are not in the RBC form. The `verify-transform` function takes as input a transformer and a circuit procedure, reifies the circuit into an AST, and verifies that applying the transformer to the reified circuit produces (an AST for) a functionally equivalent circuit.

***Implementing* TTL** Figure 5 shows a sample implementation of TTL that extends the TCL$^+$ prototype from Figure 3. The implementation exploits the ability of ROSETTE programs to reflect on the structure of symbolic values at runtime. In particular, applying a ROSETTE function to symbolic inputs produces a symbolic encoding of its output—an AST—that can be examined by ROSETTE code (such as a circuit transformation procedure) with the help of pattern matching:

```
> (RBC-parity b0 b1 b2 b3)
(! (<=> (<=> b1 b0) (<=> b2 b3)))

> (match (RBC-parity b0 b1 b2 b3)
    [(! (<=> (<=> _ _) (<=> _ _))) #t]
    [_ #f])
#t
```

In the case of circuits, ROSETTE's symbolic encoding conveniently reifies a circuit function into an AST that is built out of TTL's (and ROSETTE's) primitive boolean operators: `&&`, `||`, `!` and `<=>`. All user-defined functions are evaluated away.

The sample TTL enables circuit transformers to examine symbolic ASTs by exporting ROSETTE's **`match`** construct, which is syntactic sugar on top of Racket's own **`match`**. The implementation of solver-aided queries is also straightforward. Each query function takes as input a transformer and a circuit procedure; it reifies the circuit by applying it to a list of symbolic values; and it poses a query about the equivalence of the reified circuit and its transformed AST. The `debug-transform` query additionally asserts that the leaves of the ASTs evaluate to their corresponding bits in the provided counterexample input.

***Using* TTL** Executing the program in Figure 4 against our TTL prototype reveals a bug—a concrete input, `'(#f #f #t #f)`, on which the circuit produced by `RAX` differs from `RBC-parity`. After marking `RAX` for debugging with **`define/debug`**, we use this input to localize the fault in the transformer:

```
> (debug-transform RAX RBC-parity '(#f #f #t #f))
(define/debug (RAX ast)
  (match ast
    [(<=> left right)
     (let ([x (RAX left)]
           [y (RAX right)])
       (! (&& (! (&& x y)) (&& (! x) (! y)))))]
    [(! left) (! (RAX left))]
    [_ ast]))
```

Based on the resulting core, we hypothesize that the body of the `let` expression is faulty, replace it with an instantiation of the `Circuit` grammar, and synthesize a fix:

```
1  #lang s-exp ttl

3  (define-circuit (xor x y)
4    (! (<=> x y)))

6  (define-circuit (RBC-parity a b c d)
7    (xor (<=> a b) (<=> c d)))

9  (define-transform (RAX ast)
10   (match ast
11     [(<=> left right)
12      (let ([x (RAX left)]
13            [y (RAX right)])
14        (! (&& (! (&& x y)) (&& (! x) (! y))))))]
15     [(! left) (! (RAX left))]
16     [_ ast]))

18 (verify-transform RAX RBC-parity)
```

Figure 4. A sample program in a tiny transformation language (TTL)

```
1  #lang s-exp rosette

3  (require tcl+ rosette/lang/debug rosette/lib/tools/render
4           rosette/lib/meta/meta rosette/lib/reflect/match)

6  (provide (all-defined-out) (all-from-out tcl+) let
7           match (rename-out [define define-transform]))

9  (define (verify-transform xform circ)
10   (define input (symbolic-input circ))
11   (define ast (apply circ input))
12   (evaluate input (verify (assert (eq? (xform ast) ast)))))

14 (define (debug-transform xform circ bits)
15   (define input (symbolic-input circ))
16   (define ast (apply circ input))
17   (render
18    (debug [boolean?]
19           (begin (assert (eq? (xform ast) ast))
20                  (for ([in input] [bit bits])
21                    (assert (eq? in bit)))))))

23 (define (synthesize-transform xform circ)
24   (define input (symbolic-input circ))
25   (define ast (apply circ input))
26   (generate-forms
27    (synthesize
28     #:forall input
29     #:guarantee (assert (eq? ast (xform ast))))))
```

Figure 5. TTL in ROSETTE

```
> (define-transform (RAX ast)
    (match ast
      [(<=> left right)
       (let ([x (RAX left)]
             [y (RAX right)])
         (Circuit [! &&] x y #:depth 2))]
      [(! left) (! (RAX left))]
      [_ ast]))

> (synthesize-transform RAX RBC-parity)
(define-transform (RAX ast)
  (match ast
    [(<=> left right)
     (let ([x (RAX left)]
           [y (RAX right)])
       (! (&& (! (&& y x)) (! (&& (! x) (! y))))))]
    [(! left) (! (RAX left))]
    [_ ast]))
```

2.4 Getting Deep: A Better Tiny Transformation SDSL

So far, we have only used TTL to formulate queries about correctness of circuit transfomers on specific circuits. For example, the repair we synthesized for `RAX` is guaranteed to produce a correct transformation of `RBC-parity`, but not necessarily of other circuits. What we want instead is a more general guarantee of correctness, for both synthesis and verification queries.

To illustrate, consider a general (bounded) correctness property for TTL transformers: a transformer T is correct if its output $(T\,f)$ is equivalent to f on all n-input circuits of depth k or less. We can express this property easily in TTL by using circuit grammars. For example, the following code instantiates the property for our original `RAX` transformer (Figure 4) and all 4-input RBCs of depth $k \leq 2$:

```
> (define-circuit (RBC a b c d)
    (Circuit [! <=>] a b c d #:depth 2))
> (verify-transform RAX RBC)
verify: no counterexample found
```

But something seems wrong: the solver is unable to find any counterexamples to this more general claim, even though we know that `RAX` is buggy on at least one circuit (`RBC-parity`).

To see why our verification query failed, recall that `verify-circuit` reifies `RBC` by applying it to a list of symbolic inputs (Figure 5). The result of this application is a symbolic AST that encodes all possible RBCs but is not itself an RBC. In particular, the resulting AST contains disjunctions (`||`), which arise from **`choose`** expressions in the `Circuit` grammar that makes up the body of `RBC`. For example, evaluating (**`choose`** `b0 b1`) produces the symbolic value (`||` (`&&` c_i `b0`) (`&&` (`!` c_i) `b1`)), where c_i is a fresh symbolic boolean introduced by **`choose`**. Because of these disjunctions, the `RAX` transformer simply returns `RBC`'s AST unchanged (Figure 4, line 16), and verification fails:

```
> (match (RBC b0 b1 b2 b3)
    [(|| _ ...) #t]
    [_ #f])
#t
> (eq? (RBC b0 b1 b2 b3) (RAX (RBC b0 b1 b2 b3)))
#t
```

TTL$^+$ Deep SDSL embedding is the simplest way to implement TTL so that it supports verification and synthesis queries with strong correctness guarantees. Instead of relying on ROSETTE's symbolic values to represent circuit ASTs, we will define our own circuit data type, and write an interpreter for it. Figure 6 shows the new implementation, called TTL$^+$.

The circuit data type is defined using Racket structures, which are record types with support for subtyping. The TTL$^+$ interpreter recursively traverses a `circuit` tree and assigns meaning to each node. If a node is not an instance of the `circuit` type, the interpreter simply returns it. For example, the leaves of a `circuit` are boolean values, and they are interpreted as themselves.

Note that TTL$^+$, like its predecessor, supports the use of the `Circuit` grammar construct. But in TTL$^+$, the identifiers

```
1  #lang s-exp rosette

3  (require rosette/lib/meta/meta rosette/lib/reflect/match
4           (only-in ttl define-circuit define-transform
5                    Circuit symbolic-input))

7  (provide (rename-out [And &&] [Or ||] [Not !] [Iff <=>])
8           interpret verify-transform synthesize-transform
9           Circuit define-circuit define-transform match let
10          #%datum #%app #%module-begin #%top-interaction
11          (for-syntax #%datum))

13 (struct circuit () #:transparent)
14 (struct And circuit (left right))
15 (struct Or  circuit (left right))
16 (struct Iff circuit (left right))
17 (struct Not circuit (left))

19 (define (interpret ast)
20   (match ast
21     [(And l r) (&& (interpret l) (interpret r))]
22     [(Or l r)  (|| (interpret l) (interpret r))]
23     [(Iff l r) (<=> (interpret l) (interpret r))]
24     [(Not l)   (! (interpret l))]
25     [_         ast]))

27 (define (correct xform ast)
28   (assert (eq? (interpret ast) (interpret (xform ast)))))

30 (define (verify-transform xform circ)
31   (define input (symbolic-input circ))
32   (define ast (apply circ input))
33   (define cex (verify (correct xform ast)))
34   (values (evaluate input cex) (generate-forms cex)))

36 (define (synthesize-transform xform circ)
37   (define ast (apply circ (symbolic-input circ)))
38   (generate-forms
39    (synthesize #:forall (symbolics ast)
40                #:guarantee (correct xform ast))))
```

Figure 6. TTL$^+$ in ROSETTE

for the primitive boolean operators (`!`, `&&`, `||` and `<=>`) are bound to their corresponding circuit constructors (`Not`, `And`, `Or`, and `Iff`). As a result, a TTL$^+$ application of `RBC` to a symbolic input yields a `circuit` structure that represents all RBCs of depth 2 or less. In other words, we obtain a symbolic representation of a rich value using just the symbolic primitives available in ROSETTE (in this case, symbolic booleans introduced by the `Circuit` grammar).

The TTL$^+$ `verify-transform` query, if successful, returns two values: the booleans on which the input circuit differs from the transformed circuit, and a syntactic representation of the input circuit, if any part of its body is drawn from a grammar. For example, the following code applies the new verification query to `RAX` and our sample input circuits:

```
> (verify-transform RAX RBC-parity)
'(#f #t #f #f)
()
> (interpret (RBC-parity #f #t #f #f))
#t
> (interpret (RAX (RBC-parity #f #t #f #f)))
#f
```

```
> (verify-transform RAX RBC)
'(#f #f #f #f)
(define-circuit (RBC a b c d)
  (<=> (! a) (<=> (! a) c)))
> (define-circuit (RBC a b c d)
    (<=> (! a) (<=> (! a) c)))
> (interpret (RBC #f #f #f #f))
#f
> (interpret (RAX (RBC #f #f #f #f)))
#t
```

The `synthesize-transform` query uses the built-in procedure `symbolics` to collect all symbolic constants that appear in the reified representation of its input circuit. Universally quantifying over these constants ensures that the query works correctly on `circuit` structures that represent all RBCs of given size. For example, using the RAX sketch from the previous section, together with the RBC circuit, we can synthesize a completion of this sketch that is correct for all 4-input RBCs of depth 2 or less:

```
> (synthesize-transform RAX RBC)
(define-transform (RAX ast)
  (match ast
    [(<=> left right)
     (let ([x (RAX left)]
           [y (RAX right)])
       (! (&& (! (&& (! y) (! x))) (! (&& x y)))))]
    [(! left) (! (RAX left))]
    [_ ast]))
```

3. Core Solver-Aided Language

In this section, we present the core of the ROSETTE language—R_0—together with rules for its evaluation in the presence of symbolic values. The full language supports additional queries, expressions and data types, including vectors and user-defined algebraic data types. But the semantics of evaluation in the presence of symbolic values can be understood on just R_0. In particular, we provide the R_0 semantics in order to (i) give a model of what a solver-aided host may look like; (ii) explain the benefits of ensuring that such a language is capable of concrete (as well as symbolic) execution; and (iii) explain an alternative approach to finitization, in which the host language does not impose any built-in artifical limits on the length of executions.

3.1 Definitions and Expressions

An abstract grammar for the core language is given in Figure 7. It extends a tiny subset of Racket (core Scheme with mutation [30]) with four forms:

define-symbolic defines a variable and binds it to a fresh symbolic constant that must satisfy the predicate boolean? or number?;

assert specifies that a given expression does not evaluate to the constant #f;

(solve e) computes a binding from symbolic constants in e to concrete values such that all assertions encountered during the evaluation of e are satisfied; and,

```
prog ::= form | (begin form form ...)
form ::= (define x e) | (define-symbolic x t) | e
   e ::= (e e ...) | (set! x e) | (begin e e ...) | (if e e e)
       | (assert e) | (solve e) | (verify e)
       | v | x
   v ::= b | i | void | null | (λ (x ...) e) | false? | t | op | car | cdr | cons
   b ::= #t | #f
   i ::= integer
   t ::= boolean? | number?
  op ::= bop | aop | rop
 bop ::= && | || | !
 rop ::= < | = | >
 aop ::= + | - | * | /
   x ::= identifier
```

Figure 7. Abstract grammar for R_0

(verify e) computes a binding from symbolic constants in e to concrete values such at least one of the assertions encountered during the evaluation of e is violated.

The core language supports all standard list, boolean, arithmetic and bitwise operators, as well as standard predicates for testing equality, falseness, *etc.*—we show only a subset of these for clarity. Note that, as in Racket and Scheme, operators are functions, which are first class values. User-defined functions are supported via λ expressions.

The remaining constructs and expressions in the language are standard Scheme: set! changes the value of a defined variable; define introduces a new variable and binds it to the value of the given expression; begin groups a sequence of expressions (or, at the top level, forms), taking on the value of the last one; and $(e\ e\ldots)$ stands for procedure application, where the first expression must evaluate to a procedure that is applied to the values of the remaining expressions.

3.2 Values

R_0 programs operate on two kinds of values: *concrete* and *symbolic*. Figure 8 extends the grammar of Figure 7, which describes concrete values, with new terms that represent symbolic booleans, symbolic integers, ϕ values and models.

Symbolic booleans and integers are modeled as terms $(T\ e\ldots)$, with T specifying the type of the term. Terms of the form $(T\ x)$, where x is an identifier, represent symbolic constants introduced by define-symbolic; $(t\ op\ v\ v\ldots)$ encodes the result of applying an operator op to the specified values; and (Int (Bool $e\ldots$) $i\ i$) evaluates to one of its integer subterms, depending on the value of its boolean subterm. We introduce the (redundant) productions α, γ and ψ as notational shorthands to refer to only symbolic, only concrete, and all boolean and integer values, respectively.

In addition to primitive symbolic values, evaluation of R_0 programs can also give rise to ϕ values and models. A model μ is a map from symbolic constants to concrete values; models are produced by evaluating solve and verify expressions. A ϕ value is a set of *guard* and value pairs; ϕ values can only result from evaluation of if expressions. The guards are boolean values, and the semantics of evaluation guarantees that at most one of them is true in any model μ produced by solving or verification.

```
  v ::= .... | φ | μ
  b ::= .... | (Bool x) | (Bool bop b b ...) | (Bool rop i i)
  i ::= .... | (Int x) | (Int aop i i ...) | (Int (Bool e ...) i i)
  φ ::= (Φ (b v) ...)
  μ ::= (model mf ...)
 mf ::= ((Bool x) ↦ #t) | ((Bool x) ↦ #f) | ((Int x) ↦ integer)
  T ::= Bool | Int
  ψ ::= α | γ
  α ::= (Bool e ...) | (Int e ...)
  γ ::= #t | #f | integer
```

Figure 8. A grammar of R_0 values (including the literal values v from Figure 7)

```
  P ::= (σ π ϰ F)
  σ ::= (sf ...)
 sf ::= (x ↦ v)
  π ::= b
  ϰ ::= (b ...)
  F ::= (begin F form ...) | (define x E) | (define-symbolic x E) | E
  E ::= [] | (v ... E e ...) | (set! x E)
      | (begin E e ...) | (if E e e)
      | (assert E)
err ::= (error)
```

Figure 9. R_0 evaluation contexts (including the definitions form Figure 8)

3.3 Semantics

We define the operational semantics of R_0 by extending the reduction semantics of Scheme [30]. We have implemented and tested this semantics in PLT Redex [14]. Figure 10 shows key reduction rules for R_0, generated from our Redex model,[4] and Figure 9 shows the grammar of evaluation contexts in which these rules are applied. Recall that, in reduction semantics [13], an evaluation context is a term with a "hole," denoted by $[]$, which designates the subterm to be evaluated. We write $E[e] \longrightarrow E[e']$ to indicate that the subterm e is replaced by e' in the evaluation context E.

Reduction rules in Figure 10 operate on program states. The error state is represented by the tuple (error) and cannot be reduced any further. A valid state is represented by a tuple $(\sigma\ \pi\ \kappa\ F)$, where F is the form (with a hole) to be evaluated; σ is the program store, which maps variables to values; π is the current path condition; and κ is the constraint store. The path condition is a boolean value encoding branch decisions taken to reach the current point in the evaluation, and the constraint store contains the boolean values that have been asserted so far.

Evaluating solver-aided forms. Rules Define-1 and Define-2 use define to bind a variable with the given identifier to a fresh symbolic constant with the same identifier. Rules Assert-0 and Assert-1 enforce the usual meaning of assertions for concrete values. Assert-2 is more interesting. It updates the constraint store with a formula stating that the current path condition implies the asserted value—that is, the assertion must hold if the execution reaches the given point. We use $([\![op]\!]\ v \ldots)$ to denote construction of symbolic terms; for example, $([\![!]\!]\pi)$ constructs the value $(\mathsf{Bool}\ !\ \pi)$.

The rule for Solve-0 uses the meta-function $\mathcal{R}$, which denotes the transitive closure of the reduction relation, to fully evaluate the expression e in the current state, yielding a new state that is processed by Solve-1. The latter simply passes the formulas in κ_1, which include the original assertions from κ, to a SAT function. This function calls an underlying solver and returns a model that satisfies all the assertions in κ_1 (or raises an error if no such model exists). The Verify rules are identical, except that κ_1 assertions are treated as post-conditions to be falsified, while the original κ assertions are treated as pre-conditions.

Evaluating base forms. Given an if expression with a symbolic condition, rule If-3 fully evaluates both branches under suitably amended path conditions. If both branches produce valid states, rule If-4 uses the merge meta-function $\oplus_\alpha$ to merge the resulting program stores and values. The merge function generates ϕ values to merge terms of different types—for example, if v_0 is a boolean and v_1 is an integer, the result of their merge is the term $(\Phi\ (\alpha\ v_0)\ ((\mathsf{Bool}\ !\ \alpha)\ v_1))$. If either of the branches results in an error, that branch is abandoned by asserting the negation of its guard, and evaluation proceeds with a merge of the pre-state and the remaining branch.

Application rules App-γ and App-ψ handle the application of operators to non-ϕ values. If the operator op is applied to purely concrete values, we produce the standard meaning of this application. If it is applied to at least one abstract value, we call the term constructor on the arguments (which may yield an error). The last rule, App-ϕ, handles ϕ values. In the case shown in the figure, the range of ϕ includes exactly one value in the domain of the given operator. Evaluation proceeds by asserting the guard of that value to be true, and supplying the value to the operator. Essentially, we use assertions to implement dynamic type checks in the symbolic domain. Other cases are handled similarly.

3.4 Discussion

The R_0 evaluation rules maintain two important properties—fully concrete programs behave exactly as they would in Racket (Scheme), and our symbolic execution is both sound and complete [9]. In particular, at each step of the evaluation, the symbolic state, as given by σ, π and κ, encodes *only* and *all* concrete states (if any) that could be reached via some fully concrete execution. When no symbolic values are used in the program, this is precisely the single state reachable via a particular concrete execution.

R_0 ensures that fully concrete programs behave like Racket code in order to enable incremental development of SDSLs. In our experience, a natural way to develop an SDSL is to start with a Racket prototype of the language, test it on concrete programs and values, and then gradually add

[4] Omitted rules are similar to those in core Scheme and in Figure 10.

$$P[(\text{define-symbolic } x \text{ boolean?})] \longrightarrow P[(\text{define } x \ (\text{Bool } x))] \quad \text{[Define-1]}$$

$$P[(\text{define-symbolic } x \text{ number?})] \longrightarrow P[(\text{define } x \ (\text{Int } x))] \quad \text{[Define-2]}$$

$$P[(\text{assert } \#\text{f})] \longrightarrow (\text{error}) \quad \text{[Assert-0]}$$

$$P[(\text{assert } \#\text{t})] \longrightarrow P[\text{void}] \quad \text{[Assert-1]}$$

$$(\sigma\ \pi \varkappa\ F[(\text{assert } \alpha)]) \longrightarrow (\sigma\ \pi\ ([\![|\!|]\!]([\![!]\!]\pi)\ \alpha) :: \varkappa\ F[\text{void}]) \quad \text{[Assert-2]}$$
$$\text{where } \alpha \in \text{Bool}$$

$$(\sigma\ \pi \varkappa\ F[(\text{solve } e)]) \longrightarrow (\sigma\ \pi \varkappa\ F[(\text{solve } \mathscr{R}[\![(\sigma\ \pi \varkappa\ e)]\!])]) \quad \text{[Solve-0]}$$

$$(\sigma\ \pi \varkappa\ F[(\text{solve } (\sigma_1\ \pi_1 \varkappa_1\ v))]) \longrightarrow (\sigma_1\ \pi_1 \varkappa_1\ F[SAT[\![\varkappa_1]\!]]) \quad \text{[Solve-1]}$$

$$(\sigma\ \pi \varkappa\ F[(\text{verify } e)]) \longrightarrow (\sigma\ \pi \varkappa\ F[(\text{verify } \mathscr{R}[\![(\sigma\ \pi \varkappa\ e)]\!])]) \quad \text{[Verify-0]}$$

$$(\sigma\ \pi \varkappa\ F[(\text{verify } (\sigma_1\ \pi\ (b\ \ldots)\ v))]) \longrightarrow (\sigma_1\ \pi \varkappa\ F[SAT[\![([\![|\!|]\!]([\![!]\!]b)\ \ldots) :: \varkappa]\!]]) \quad \text{[Verify-1]}$$

$$(\sigma\ \pi \varkappa\ F[(\text{if } \alpha\ e_0\ e_1)]) \longrightarrow (\sigma\ \pi \varkappa\ F[(\text{if } \alpha\ \mathscr{R}[\![(\sigma\ ([\![\&\&]\!]\pi\ \alpha) \varkappa\ e_0)]\!]\ \mathscr{R}[\![(\sigma\ ([\![\&\&]\!]\pi\ ([\![!]\!]\alpha)) \varkappa\ e_1)]\!])]) \quad \text{[If-3]}$$

$$(\sigma\ \pi \varkappa\ F[(\text{if } \alpha\ (\sigma_0\ \pi_0 \varkappa_0\ v_0)\ (\sigma_1\ \pi_1 \varkappa_1\ v_1))]) \longrightarrow (\sigma_0 \bigoplus_\alpha \sigma_1\ \pi \varkappa_1 :: \varkappa_0 :: \varkappa\ v_0 \bigoplus_\alpha v_1) \quad \text{[If-4]}$$

$$(\sigma\ \pi \varkappa\ F[(\text{if } \alpha\ err_0\ (\sigma_1\ \pi_1 \varkappa_1\ v_1))]) \longrightarrow (\sigma \bigoplus_\alpha \sigma_1\ \pi \varkappa_1 :: \varkappa\ (\text{begin } (\text{assert } (!\ \alpha))\ v_1)) \quad \text{[If-5]}$$

$$P[(op\ \gamma\ \ldots)] \longrightarrow P[[\![op\ \gamma\ \ldots)]\!]] \quad \text{[App-}\gamma\text{]}$$

$$P[(op\ \psi_0\ \ldots\ \alpha\ \psi_1\ \ldots)] \longrightarrow P[([\![\text{op}]\!]\psi_0\ \ldots\ \alpha\ \psi_1\ \ldots)] \quad \text{[App-}\psi\text{]}$$

$$P[(op\ v_0\ \ldots\ \phi\ v_1\ \ldots)] \longrightarrow P[(op\ v_0\ \ldots\ (\text{begin } (\text{assert } b)\ v)\ v_1\ \ldots)] \quad \text{[App-}\phi\text{]}$$
$$\text{where } (\Phi\ (b\ v)) = \phi \downarrow dom[\![op]\!]$$

Figure 10. R_0 reduction rules as an extension of Figure 9

the desired solver-aided functionality. This is how two of the case studies presented in Section 4 were developed.

The R_0 semantics also captures an important design decision behind ROSETTE: it admits only encodings of finite executions to formulas. This can be seen from rules for Solve and Verify. But unlike other symbolic execution systems that are based on bounded reasoning (*e.g.,* [8, 11, 12, 16, 34, 36]), we do not artificially finitize executions by, *e.g.,* unrolling recursions a finite number of times for two reasons.

First, many uses of loops and recursion just iterate over either a concrete instance or a symbolic encoding of a data structure, such as a list, a vector, or a **`struct`**. For example, our TTL$^+$ interpreter (Figure 6) uses recursion to traverse an instance (or a symbolic encoding) of a `circuit` **`struct`**. ROSETTE executes all such loops and recursion precisely—that is, they are executed exactly as many times as needed to fully traverse the given structure. As a result, most loops and recursion in ROSETTE programs are self-finitizing with respect to program state, enabling, for example, synthesis of programs that traverse large heap structures, such as a **`struct`** representation of a 2000-node HTML tree (Section 4.1).

Second, in cases where explicit finitization is required, it is easy to implement in ROSETTE with the help of macros. We therefore leave the control over how this is performed to users, allowing them to define finitization behaviors that are best suited to their problem. For example, Figure 11 shows a macro that implements a finitized **`while`** loop construct, which we used in a toy SDSL version of the Brainfudge language [43]. The loop can execute at most three times:

```
> (define (upto10 start)
    (define x start)
    (while (< x 10)
           (set! x (+ x 1))))
> (define-symbolic i number?)
> (evaluate i (solve (upto10 i)))
8
> (solve (begin (assert (< i 7))
                (upto10 i)))
solve error: no satisfying execution found
```

```
1  (define-syntax-rule (while test body ...)
2    (local [(define (loop bound)
3              (if (<= bound 0)
4                  (assert (not test))
5                  (when test
6                    body ...
7                    (loop (- bound 1)))))]
8      (loop 3)))
```

Figure 11. A macro for a finitized **`while`** loop

```
1  ; A DOM node consists of a tag name, a list of attributes,
2  ; and a list of content nodes or strings.
3  (struct DOM (tagname attributes content) #:transparent)

5  ; Returns true if the provided list of strings represents
6  ; a ZPath that connects the given source and sink
7  ; elements in an HTML tree (e.g., the root of the tree
8  ; and an example input).
9  (define (zpath? zpath source sink)
10   (or (and (equal? source sink)
11            (andmap (curry equal? "") zpath))
12       (and (DOM? source)
13            (not (null? zpath))
14            (equal? (car zpath) (DOM-tagname source))
15            (ormap (lambda (child)
16                     (zpath? (cdr zpath) child sink))
17                   (DOM-content source)))))
```

Figure 12. The WebSynth ZPath interpreter and DOM

4. Case Studies

In this section, we present three solver-aided systems that have been developed with ROSETTE, including a declarative DSL for web scraping; a spatial programming model for ultra low-power architectures; and a superoptimizer for bitvector programs. Two of these systems have been developed by undergraduates and first-year graduate students with no prior experience with using ROSETTE. We observe that ROSETTE's embedding in a fully-featured programming language, as well as its lightweight approach to symbolic reasoning, make it possible to quickly prototype solver-aided systems that are both immediately usable and sufficiently scalable.

```
1  #lang s-exp websynth

3  (define dom
4    (DOM "[document]" '()
5         `(,(DOM "html" '()
6                 `(,(DOM "body" '()
7                         `(,(DOM "ul" '()
8                                 `(,(DOM "li[0]" '()
9                                         `("Hello"))
10                                ,(DOM "li[1]" '()
11                                      `("World")))))))))

13 (define-symbolic z0 z1 z2 z3 z4 string?)
14 (define zp (list z0 z1 z2 z3 z4))
15 (evaluate zp (solve (assert (zpath? zp dom "World"))))
```

(a) Synthesizing a ZPath to scrape the string "World" from the HTML page shown in (b)

```
<html>
<body>
   <ul>
     <li>Hello</li>
     <li>World</li>
   </ul>
</body>
</html>
```

(b) A tiny HTML page

```
'("[document]"
  "html"
  "body"
  "ul"
  "li[1]")
```

(c) The output of the program in (a)

Figure 13. A sample WebSynth program

4.1 WebSynth

WebSynth is a solver-aided system for example-based web scraping. The problem of retrieving data from HTML files is surprisingly difficult in practice—it is usually solved by writing custom scripts or regular expressions, which must be manually revised whenever the target page changes. The problem is compounded when many slightly different scripts have to be maintained in order to scrape data from related, but separately coded and maintained web sites (*e.g.,* to collate meeting schedules for organizations with many independent chapters, such as Alcoholics Anonymous).

WebSynth takes a more robust, solver-aided approach to the scraping problem. Given an HTML file and one or more representative examples of data that should be retrieved, the system synthesizes a single *ZPath expression* that can be used to retrieve all of the example data. ZPaths are expressions in the ZPath language, which is a declarative DSL based on XPath [42]. Expressions in the language specify how to traverse the HTML tree to retrieve the desired data, and the retrieval is performed by the ZPath interpreter. For example, the ZPath "/html/body/div/" selects all top-level DIV elements in an HTML tree. ZPath nodes can also include indices that enable selection of a particular child of a node in the path.

Because ZPaths are generated automatically by the solver, it is easy to maintain them—if the tree structure of the target page changes, the synthesizer is re-executed to produce a new script. In fact, during the development of WebSynth, one of its benchmark pages [19] was restructured, and the system was able to generate another scraper for it in seconds.

The WebSynth system was developed by two undergraduate students in just a few weeks. They first implemented their ZPath interpreter and a DOM data structure for representing HTML trees in Racket. Figure 12 shows an excerpt from this implementation. ZPaths are modeled as lists of string tokens (such as "html" or "div[0]"), and a DOM node is a **struct** with fields for storing the node's HTML tag and children. The interpreter simply checks that a given list of strings forms a ZPath between two elements in the HTML tree—for example, the root of the tree and a string to be scraped. This initial prototype enabled the students to test the system by providing concrete ZPaths to the interpreter and checking that they retrieved the desired data, *e.g.,* the top 100 song names from Apple's iTunes Charts [19].

The next step was to turn the ZPath interpreter into a synthesizer using ROSETTE. This involved changing the implementation language from Racket to ROSETTE; making ZPaths symbolic by letting each element of a ZPath list be a fresh symbolic value;[5] and asserting that the example data is retrieved by running the interpreter on the symbolic ZPath. Figure 13 shows a sample HTML page; the ZPath program generated for this page and the example input "World"; and the result of executing the program in ROSETTE.

The system handles multiple input examples by extending the one-input case as follows. For each input example i, we create a symbolic ZPath zp_i that scrapes just i (see lines 13-14 in Figure 13a). Then, we create a symbolic $mask$—expressed as a nested list of symbolic booleans—that can be applied to the individual zp_i's to compute a generalized ZPath that scrapes all the given data. For example, to scrape both "Hello" and "World" from the page in Figure 13, the system replaces the last three lines of our ZPath program with the following code (as well as the code that creates the zp_i's and the $mask$):

```
(define model
 (solve (begin (assert (zpath? zp0 dom "World"))
               (assert (zpath? zp1 dom "Hello"))
               (assert (generalizes? mask zp0 zp1)))))
(generalize (evaluate mask model)
            (evaluate zp0 model) (evaluate zp1 model))
```

WebSynth can synthesize ZPaths for real websites, with DOMs that consist of thousands of nodes, in a few seconds. Table 1 shows the performance data we obtained by applying WebSynth to three example web pages: iTunes Top 100 Songs [19]; IMDb Top 250 Movies [18]; and AlAnon AR Meetings [4]. The second and third column display the size of each page, given as the number of DOM nodes and the depth of the DOM tree. We scraped each page four times, varying the number of input examples. The corresponding rows in the table show the total running time and just the solving time for each set of examples. The solving time includes symbolic execution, compilation to formulas, and the solver's running time. All example sets for a given page correspond to the same generalized ZPath. The IMDb ZPath extracts the titles of the 250 movies listed on the page; the iTunes ZPath extracts the title and the artist for each song; and the AlAnon ZPath extracts the group name, address and city for each

[5] ROSETTE allows use of symbolic strings that are treated as atomic values—*i.e.,* they can be compared for equality but are otherwise uninterpreted.

Page	Nodes	Depth	Examples	Total (sec)	Solve (sec)
iTunes	1104	10	2	14.0	0.4
			4	14.5	0.7
			8	15.5	1.3
			16	17.5	2.3
IMDb	2152	20	2	15.1	0.5
			4	15.6	0.7
			8	16.7	1.0
			16	18.7	1.6
AlAnon	2002	22	2	15.6	.9
			4	17.0	1.6
			8	20.0	3.1
			16	26.2	5.0

Table 1. Performance of WebSynth on 3 sample web pages

listed meeting. We performed all experiments on an Intel Core 2 Duo 2.13 GHz processor with 4 GB of memory.

This case study shows the advantage of ROSETTE's approach to symbolic evaluation of loops and recursion, which is enabled by aggressive partial evaluation. Rather than imposing artificial bounds on the length of executions, we allow the structure of the data to guide the execution. In the case of WebSynth, the concrete structure of the DOM tree precisely determines how many times the body of the interpreter in Figure 12 is executed. As a result, the students were able to turn their concrete interpreter of ZPaths into a scalable synthesizer simply by making the ZPaths symbolic, while keeping the DOM concrete. No special handling of the recursion in the interpreter was required, nor did the students have to resort to developing a tricky deterministic algorithm for synthesizing generalized ZPaths that work for multiple, potentially ambiguous input examples.[6]

4.2 A Partitioner for a Spatial Programming Model

A project in our group focuses on developing a synthesis-aided programming model for GA144, a many-core architecture composed of 144 tiny ultra-low-power processors. Because each core has only 300 words of memory and each word is only 18-bits wide, a program must be partitioned in an unusually fine-grained fashion, including bit-slicing of arithmetic operations—*i.e.,* splitting 32-bit integer operations into two 18-bit operations placed on neighboring cores. Additionally, the cores use a stack-based architecture with a small subset of Forth as its machine language. There are no compilers for either code partitioning or for the generation of optimal stack-based code. To illustrate the magnitude of the programming challenge, we note that an MD5 hash computation is partitioned across 10 cores and the machine code sometimes exploits tricks such as intentionally overflowing an 8-word stack.

We are developing a programming model that (i) partitions a single-core program; (ii) routes the communication across the fabric of cores; and (iii) uses superoptimization to generate optimal single-core code. The machine-code superoptimizer is similar to that of Section 4.3. Here, we describe our experience with using ROSETTE to develop the top part of the compilation pipeline.

To support partitioning, the programming model maps each logical program location (a variable or an array element) onto one of the cores. Integers larger than a machine word are represented as tuples of logical locations, each of which may be mapped onto a different core. Additionally, each operation (such as '+') is mapped onto the core that will execute it.

The sample code below shows the surface syntax of a program in our imperative spatial programming language. An integer is first defined as a pair of machine integers. A 64-element array of these pairs is then defined and distributed across cores 106 and 6 in such a way that the more significant words of these 64 integers are on core 106, while the lower words are all on core 6. Next, the function `sumrotate` is distributed across cores 105 and 5, which means that its return value will be computed at these two cores. The keyword `@here` at the first addition indicates that the addition will execute on the same cores as its containing function—cores 105 and 5. Note that this addition decomposes into two additions of machine words. The second addition is not assigned to specific cores, which means that the synthesizer is free to assign the two component additions onto suitable cores of its choice. In the extreme, the programmer can choose to assign cores to no operations or data, and the synthesizer will map all of them to cores that minimize communication cost of passing data to operations, while ensuring that the data fit into the memory capacity of each core.

```
1  typedef pair<int,int> myInt;

3  vector<myInt>@{[0:64]=(106,6)} k[64];

5  myInt@(105,5) sumrotate(myInt@(104,4) buffer, ...) {
6    myInt@here sum = buffer +@here k[i] + message[g];
7    ...
8  }
```

The core assigned to a program location or an operation can be thought of as their type. The corresponding non-traditional type system computes an abstract communication cost for each inter-core communication that must be performed to evaluate the program. A program type checks if its communication cost is below a certain threshold and when the total amount of locations assigned to a core does not exceed the memory capacity of the core.

The problem of partitioning a program can be thought of as optimal type inference, *i.e.,* assigning cores to program locations and operations such that the total abstract cost is minimized, subject to the capacity constraints of the cores.

To implement a synthesizer for this problem, a student first implemented a classic type checker for the partitioning type system. The type checker, designed as an abstract interpreter, traverses the program AST and computes the abstract communication cost given a provided assignment of

[6] An example string is ambiguous when it appears in several different DOM nodes, and can therefore be scraped by several different ZPaths.

Program	Unknowns	Asserts	LOC	Time (sec)
array	3	77	6	1
for/array	12	607	9	7
for/array/tuple	12	607	9	17
function	12	435	17	2
matrix multiply	8	1079	18	17
md5	14	10940	116	1197
md5	8	6440	116	335

Table 2. Performance of the code partitioning synthesizer for the GA144 imperative spatial programming language

cores to locations and operations. It also computes the number of logical locations assigned to each core. A communication cost is charged whenever an argument of an operation lives on a different core than the operation. In the end, the type checker verifies that the total cost is below the threshold and the data fit into each core. This type checker was written in pure Racket and debugged on programs where all data and operations were assigned to cores by the programmer, *i.e.,* there was no core assignment left to synthesize.

Next, to turn this type checker into a type inferencer (and thus into a partitioning synthesizer), the student did nothing more than to replace the cores previously assigned by the programmer with ROSETTE symbolics, which made these core assignments unknown. Given the same type checker, the query (`solve` (`type-check program`)) was then used to compute an assignment of cores that meets the capacity constraints, while keeping the abstract cost below a given threshold. To obtain optimal partitioning, it sufficed to iteratively tighten the bound, performing binary search until a better assignment of cores could not be found.

We have successfully used this synthesizer to partition the MD5 checksum computation across 10 cores, obtaining optimal partitionings that differed as we modified the memory capacity of cores. Table 2 displays the synthesizer's running time on the MD5 function and five other benchmarks. We show, for each benchmark, the number of unknown placements of variables and operations; the number of assertions emitted by the type checker; the lines of code in the high-level SDSL program; and the total time taken to synthesize the optimal solution. The most challenging program, requiring placement of 14 program elements, was partitioned in about 20 minutes. We consider this an acceptable compilation time considering that a human is likely to take much longer to perform the task.

This case study pointed out the advantage of ROSETTE's introduction of partial evaluation into symbolic compilation. The access to non-symbolic parts of Racket greatly simplified the development of the synthesizer. Specifically, the AST of the spatial programming model was developed with the Racket object system and the type checker used a Visitor pattern that relied on inheritance. Compiling objects and class hierarchies to constraints is a nontrivial effort and, in fact, neither our symbolic core nor the Sketch synthesizer [34] support objects and classes. Unable to compile classes to constraints, the student would have to rewrite the type checker to operate on a simpler data structure, *e.g.,* a list-based AST. Additionally, the object-based AST would have to be converted to the list-based tree. Luckily, with ROSETTE, neither the AST nor the type checker had to be modified. The checker's traversal of the object-based AST was partially evaluated away (because the partial evaluation was performed on a given program that was a runtime constant to the partial evaluator), leaving to the symbolic compiler only a residual program that calculated and checked the memory capacities and communication costs as a function of the symbolic cores. Thanks to the partial evaluator, the type checker code did not need to be modified at all and it was converted to an inferences automatically.

```
1  (configure [bitwidth 32])

3  (define-fragment (fast-max x y)
4    #:ensures  (lambda (x y result) (= result (max x y)))
5    #:library  (bvlib [{bvneg bvge bvand} 1] [{bvxor} 2]))
```

Figure 14. Defining a bit vector program that computes the maximum of two integers

4.3 A Superoptimizer for Bitvector Programs

We expect that most tools developed with ROSETTE will be able to use its solver-aided facilities—such as **`solve`** and **`synthesize`**—as a black box. But occasionally, specialized solving facilities may be needed for performance reasons.

We have used ROSETTE to implement such a facility, in the form of a specialized synthesis algorithm [17] for superoptimization of bitvector programs. The synthesizer takes as input a reference program and a bag (multi set) of low-level bitwise instructions, each of which can occur (at most) once in the synthesized program. Given these inputs, it finds a permutation of the instructions—and a way to wire the output of one to the input of another—so that the final program is functionally equivalent to the reference implementation. The search for the program is cleverly reduced to a constraint solving problem, so that the resulting encoding is quadratic in the size of the input multi set. A compact encoding of this kind cannot be produced by general-purpose synthesis tools, such as Sketch [34], without modifying their hardwired synthesis algorithms.

We were able to prototype a synthesizer that generates this encoding with fewer than 600 hundred lines of ROSETTE code. The synthesizer is implemented entirely as a user-level library. Its functionality is exposed to clients via a simple macro, which enables succinct specification of the bag of instructions and the reference function. The bit vector implementation is synthesized at macro-expansion time, and becomes available to client code at runtime. It can be used immediately, as well as printed and saved for future use.

Figure 14 shows an example use of the bit vector synthesizer. The `define-fragment` macro invokes the synthesizer

(at expansion time) with the specified correctness checker and library of instructions. In our example, `define-fragment` is used to create a branch-free bit vector program for finding the maximum of two 32-bit integers. The synthesizer produces an implementation in 6 seconds. The implementation is bound to the variable `fast-max`, and it can be used like any other Racket function. The `define-fragment` macro also introduces the identifier `fast-max-stx`, which is bound to a syntactic representation of the synthesized code:

```
> (fast-max 49392 848291)
848291

> fast-max-stx
(lambda (x y)
  (let* ([t11 (bvge y x)]
         [t12 (bvneg t11)]
         [t13 (bvxor y x)]
         [t14 (bvand t13 t12)]
         [t15 (bvxor x t14)])
    t15))
```

Our synthesizer achieves comparable performance to that reported for the Brahma tool on a set of 25 benchmarks [17]. Table 3 shows the size of each benchmark, given as the number of lines of code, and the time taken to synthesize it. Unlike Brahma, our implementation times out after an hour on four benchmarks, which we believe is due to the use of a different solver. Nonetheless, this is considerably better than the results reported for other general-purpose synthesizers [17]. We are currently working on connecting ROSETTE to different backends—a project made relatively easy by the fact that ROSETTE compiles programs to simple formulas over bitvectors and booleans.

Benchmark	LOC	Time (sec)
P1	2	2.7
P2	2	2.3
P3	2	1.9
P4	2	2.0
P5	2	2.0
P6	2	1.9
P7	3	2.3
P8	3	3.2
P9	3	3.5
P10	3	2.5
P11	3	2.3
P12	3	2.4
P13	4	3.6
P14	4	2.5
P15	4	2.6
P16	4	3.8
P17	4	3.6
P18	6	5.1
P19	6	11.3
P20	7	684.2
P21	10	483.0
P22	8	timeout
P23	10	timeout
P24	12	timeout
P25	16	timeout

Table 3. Performance of ROSETTE's superoptimizer on 25 bitvector benchmarks from [17]

5. Related Work

ROSETTE builds on a rich body of prior work on solver-aided languages [5, 23–25, 27, 31, 34, 35] and symbolic evaluation [8, 10, 12, 15, 21, 37, 40]. Its language is most closely related to Sketch [34], Kaplan [23], and Rubicon [27]. All three are domain-specific languages: Sketch is a small, hand-crafted C-like language; Kaplan is a high level functional language embedded in Scala; and Rubicon is an SDSL embedded in Ruby. ROSETTE shares some features of these languages, but it differs from each in two key aspects that enable its use as a host solver-aided language. In particular, ROSETTE supports metaprogramming (inherited from Racket), which enables easy language embedding; and it also supports all basic solver-aided queries, which enables automation of a variety of constructs in a guest SDSL.

Like Kaplan, ROSETTE is an embedded language that exposes symbolic values as first-class constructs. A distinguishing feature of Kaplan is its native support for many different types of symbolic values, including sets and maps, which are not supported natively in ROSETTE. Instead, ROSETTE programs build richer symbolic values using a small core language and a few primitive data types. But because ROSETTE tracks only a few values symbolically, it can afford to provide full symbolic support for its core language, which, like Sketch and Rubicon, includes state mutation and reasoning about both branches of a conditional that depends on a symbolic value. Neither of these are supported in Kaplan.

Unlike Sketch, ROSETTE provides a rich set of metaprogramming facilities, inherited from Racket. The access to these facilities, as well as the access to first-class symbolic values, is particularly important for development of new language constructs and facilities. This is difficult to do in a stand-alone language like Sketch without modifying its compiler. In contrast, we have been able to enrich ROSETTE with user-level libraries that implement new kinds of synthesis algorithms [17] and constructs for specifying rich holes.

Rubicon's approach to symbolic evaluation implements a solver-aided semantics that is similar to the semantics of R_0 (Section 3). In particular, both R_0 and Rubicon behave like their host languages (Racket and Ruby, respectively) on fully concrete programs. But Rubicon's symbolic extension to its host language is specialized for verification queries, while ROSETTE handles other queries as well.

ROSETTE's symbolic evaluation is related to that performed in other bounded verification tools such as [8, 10, 12, 15, 21, 37, 40], and its angelic execution facilities are related to prior work on angelic program repair [28, 29, 32, 33] and declarative execution [26]. Unlike these tools, however, ROSETTE does not reason symbolically about complex features of a large language, relying instead on aggressive partial evaluation. This enhances both its flexibility and scalability,

and we have found it to be a powerful way to obtain many of the benefits of solver-aided programming without as much engineering effort.

6. Conclusion

Solver-aided languages help write programs that invoke runtime oracles, contain holes to be completed by a synthesizer, and include checks of correctness. They also help localize and repair bugs. We describe ROSETTE, an extension to Racket that includes a symbolic compiler for translating solver-aided programs into logical constraints. ROSETTE also maps the result of constraint solving back to the program, which the program can use to update program state, produce an expression, or generate other constraints. The novel advantage of ROSETTE is that it provides a relatively small symbolic core but extends the power of its language by preceding symbolic compilation with partial evaluation, which allows ROSETTE program to use any Racket constructs not compilable to constraints as long as these constructs are partially evaluated away. This architecture allowed us to construct synthesizers without changing existing interpreters and type checkers, by just making their inputs symbolic.

Acknowledgments

This research was supported in part by awards from the Department of Energy (DE–SC0005136 and DE–FOA-0000619), National Science Foundation (NSF CCF–0916351 and NSF CCF–1139138), Samsung Electronics, UC Discovery, and Intel and Microsoft Awards to the Berkeley Universal Parallel Computing Research Center. We would also like to thank Mangpo Phothilimthana for providing the performance data on her code partitioning synthesizer, and Brandon King and Omar Rehmane for their WebSynth source code and benchmarks.

References

[1] A. Aavani. Translating pseudo-boolean constraints into CNF. In *Theory and Application of Satisfiability Testing (SAT)*, 2011.

[2] P. A. Abdulla, P. Bjesse, and N. Eén. Symbolic reachability analysis based on SAT-solvers. In *Tools and Algorithms for Construction and Analysis of Systems (TACAS)*, 2000.

[3] AIGER. `fmv.jku.at/aiger/`.

[4] Al-Anon AR Meetings. `www.ar.al-anon.alateen.org/alanonmeetings.htm`.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure and Interoperable Smart Devices (CASSIS)*, 2004.

[6] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification. `www.eecs.berkeley.edu/~alanmi/abc/`.

[7] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *Intl. Conf. on Software Engineering (ICSE)*, 2011.

[8] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004.

[9] G. Dennis. *A relational framework for bounded program verification*. PhD thesis, Massachusetts Institute of Technology, 2009.

[10] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2006.

[11] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2006.

[12] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *Foundations of Software Engineering (FSE)*, 2007.

[13] M. Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[14] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. The MIT Press, 1st edition, 2009. ISBN 0262062755, 9780262062756.

[15] J. Galeotti, N. Rosner, C. Pombo, and M. Frias. Analysis of invariants for efficient bounded verification. In *Intl. Symp. on Software Testing and Analysis (ISSTA)*, 2010.

[16] J. P. Galeotti. *Software Verification Using Alloy*. PhD thesis, University of Buenos Aires, 2010.

[17] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Programming Language Design and Implementation (PLDI)*, 2011.

[18] IMDb Top 250 movies. `www.imdb.com/chart/top`.

[19] iTunes Top 100 Songs. `www.apple.com/itunes/charts/songs/`.

[20] P. Jackson and D. Sheridan. Clause form conversions for boolean circuits. In *Theory and Application of Satisfiability Testing (SAT)*, 2005.

[21] JavaPathFinder. `babelfish.arc.nasa.gov/trac/jpf/`.

[22] M. Jose and R. Majumdar. Bug-Assist: assisting fault localization in ansi-c programs. In *Computer Aided Verification (CAV) verification*, 2011.

[23] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *Principles of Programming Languages (POPL)*, 2012.

[24] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, 2010.

[25] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2010.

[26] A. Milicevic. Executable specifications for Java programs. Master's thesis, Massachusetts Institute of Technology, 2010.

[27] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *Foundations of Software Engineering (FSE)*, 2012.

[28] R. Nokhbeh Zaeem and S. Khurshid. Contract-based data structure repair using Alloy. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2010.

[29] R. Nokhbeh Zaeem, D. Gopinath, S. Khurshid, and K. S. McKinley. History-aware data structure repair using SAT. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2012.

[30] J. D. Ramsdell. An operational semantics for Scheme. *SIGPLAN Lisp Pointers*, V(2):6–10, 1992.

[31] H. Samimi and K. Rajan. Specification-based sketching with Sketch#. In *Workshop on Formal Techniques for Java-Like Programs (FTfJP)*, 2011.

[32] H. Samimi, E. D. Aung, and T. Millstein. Falling back on executable specifications. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2010.

[33] J. H. Siddiqui and S. Khurshid. Symbolic execution of Alloy models. In *Intl. Conf. on Formal Methods and Software Engineering (ICFEM)*, 2011.

[34] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2006.

[35] P. Suter, A. S. Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *Static Analysis Symp. (SAS)*, 2011.

[36] M. Taghdiri. *Automating Modular Program Verification by Refining Specifications*. PhD thesis, Massachusetts Institute of Technology, 2007.

[37] M. Taghdiri. *Automating Modular Program Verification by Refining Specifications*. PhD thesis, Massachusetts Institute of Technology, 2007.

[38] R. Tate, M. Stepp, and S. Lerner. Generating compiler optimizations from proofs. In *Principles of Programming Languages (POPL)*, 2010.

[39] The Racket Programming Language. `racket-lang.org`.

[40] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: checking axiomatic specifications of memory models. In *Programming Language Design and Implementation (PLDI)*, 2010.

[41] W. Weimer, S. Forrest, C. L. Goues, and T. Nguyen. Automatic program repair with evolutionary computation. *Commun. ACM*, 53(5):109–116, 2010.

[42] XPath. XML Path Language. `www.w3.org/TR/xpath/`.

[43] D. Yoo. Fudging up Racket. `hashcollision.org/brainfudge/`.