A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages

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

Abstract

Solver-aided domain-specific languages (SDSLs) are an emerging class of computer-aided programming systems. They ease the construction of programs by using satisfiability solvers to automate tasks such as verification, debugging, synthesis, and non-deterministic execution. But reducing programming tasks to satisfiability problems involves translating programs to logical constraints, which is an engineering challenge even for domain-specific languages.

We have previously shown that translation to constraints can be avoided if SDSLs are implemented by (traditional) embedding into a host language that is itself solver-aided. This paper describes how to implement a symbolic virtual machine (SVM) for such a host language. Our symbolic virtual machine is lightweight because it compiles to constraints only a small subset of the host's constructs, while allowing SDSL designers to use the entire language, including constructs for DSL embedding. This lightweight compilation employs a novel symbolic execution technique with two key properties: it produces compact encodings, and it enables concrete evaluation to strip away host constructs that are outside the subset compilable to constraints. Our symbolic virtual machine architecture is at the heart of ROSETTE, a solver-aided language that is host to several new SDSLs.

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, Symbolic Virtual Machine

1. Introduction

Satisfiability solvers are the workhorse of modern formal methods. At least four classes of tools reduce programming problems to satisfiability queries: verification [11], synthesis [37], angelic (non-deterministic) execution [10], and fault localization [20]. The key component of all such tools is a *symbolic compiler* that translates a program to logical constraints.

Building a symbolic compiler is often the most difficult aspect of creating solver-aided tools, especially for general-purpose languages. Only a few symbolic compilers exist for C [11, 41] and Java [15, 16], for example, and all took years to develop. Moreover, programs in these languages are typically large, and even highly optimized symbolic compilers reduce them to large, hard-to-solve encodings. As a result, the applicability of solver-aided tools to general-purpose programming remains limited.

We believe that domain-specific languages (DSLs) are more suitable for wider adoption of solver-aided tools, for two reasons. First, DSLs are already part of everyday programming, in the form of frameworks and libraries. Second, DSL programs are smaller, resulting in smaller logical encodings, which can be further optimized by exploiting domain invariants. But building a symbolic compiler to generate these encodings remains a formidable task, even for a DSL (see, *e.g.,* [26]).

In prior work [39], we showed how to implement DSLs that are equipped with solver-aided tools (*i.e.,* verification, debugging, angelic execution, and synthesis), without constructing new symbolic compilers. The designer of such a *solver-aided DSL* (SDSL) simply defines the semantics of his language by writing an interpreter or a library in a *solver-aided host language*. The host language exposes solver-aided tools as first-class constructs, which SDSLs designers use to implement solver-aided facilities for their languages. The host's symbolic compiler then reduces the SDSL implementation and a program in that SDSL to constraints.

In this paper, we focus on the challenge of implementing a solver-aided host language, the crux of which is building its symbolic compiler. Since host languages offer metaprogramming constructs for DSL construction, such as syntactic macros, which have not yet been successfully compiled to constraints, volunteering to symbolically compile a host language may seem overly altruistic: while creators of guest SDSLs are saved from writing symbolic compilers, we are faced with compiling a complex general-purpose language. Using ROSETTE [39], our solver-aided extension to Racket [32], this paper shows that it is possible to host SDSLs without building a complex symbolic compiler.

We avoid the difficulties of classic symbolic compilation through a lightweight design that compiles to formulas only a small *lifted* subset of ROSETTE, while allowing SDSL designers to use the entire language. We call this compiler a *symbolic virtual machine* (SVM), because it virtualizes access to the underlying satisfiability solver. Its key contribution is a new way to combine symbolic execution [23] and bounded model checking [11] to achieve two competing goals: (1) efficient compilation of lifted constructs, and (2) concrete evaluation of unlifed constructs.

Symbolic execution and bounded model checking are classic techniques for compiling (loop-free) programs to constraints. Given a program with symbolic inputs, they both produce a formula that encodes the program's semantics in terms of those inputs. But symbolic execution maximizes opportunities for concrete evaluation at the cost of exponential compilation, while bounded model checking

PLDI '14, June 9–11, 2014, Edinburgh, United Kingdom.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2784-8/14/06…$15.00.
http://dx.doi.org/10.1145/2594291.2594340

prioritizes efficient compilation at the cost of minimized opportunities for concrete evaluation. The difference between the two approaches boils down to how they perform *state merging*.

Symbolic execution encodes each path through a program separately, so program states (bindings of variables to values) that correspond to different paths are never merged. This encoding process is exponential, because the number of paths in a program is generally exponential in its size. But path-based encoding also maximizes opportunities for concrete evaluation. Because many values in the program state remain concrete along a given path, all constructs that consume these values can be evaluated concretely. We say that such constructs are stripped away, since they do not need to be compiled to constraints.

Bounded model checking merges program states obtained by encoding the branches of each conditional, which results in polynomial compilation. The price, however, is the loss of opportunities for concrete evaluation. When two concrete values from different branches are merged, the result is a symbolic value. After a few merges, most values in the program state become symbolic, and all constructs that consume them must be encoded into the final formula.

Our SVM approach employs a new *type-driven state merging* strategy. Like bounded model checking, the SVM merges states at each control-flow join, enabling efficient compilation. But our merging function is different. To expose (more) opportunities for concrete evaluation, it produces a symbolic value only when merging two values of a primitive type (integer or boolean). Merging values of other types results in either a concrete value or a *symbolic union* of (guarded) concrete values. For example, two lists of the same length are merged element-wise to produce a concrete list, while lists of different lengths are merged into a symbolic union. If such a union flows into a lifted procedure that operates on lists, the SVM evaluates the procedure concretely on the individual members of the union, and then combines the results into a single (concrete or symbolic) value. As a result, the SVM does not need to encode complex lifted constructs, such as list operations, in the solver's input language—they are simply evaluated with the Racket interpreter, under the SVM's guidance.

The output of SVM evaluation is a formula comprised of all symbolic boolean values that flow into an assertion. This formula is passed to the underlying solver. Because the SVM maintains the invariant that symbolic values of primitive types, such as boolean or integer, are expressed exclusively in terms of other primitive values, the solver receives encodings that are free of symbolic unions.

The SVM lifts only a subset of ROSETTE's constructs to operate on symbolic values. This subset is rich enough, and our merging strategy produces enough concrete values, that symbolic values rarely flow to unlifted constructs. If they do, however, the SVM offers an easy mechanism, which we call *symbolic reflection*, to extend symbolic evaluation to those unlifted constructs. With a few lines of code, and without modifying the SVM, SDSL designers can obtain symbolic encodings of complex operations (such as regular expression matching) that are, in general, poorly supported by symbolic compilers.

Contributions This paper makes the following contributions:

- We develop a lightweight symbolic virtual machine (SVM) that compiles to constraints only a small lifted subset of a solver-aided host language, while exposing rich metaprogramming facilities to SDSL designers. This compilation approach is enabled by a new state-merging technique that relies on symbolic unions. It is both efficient (*i.e.,* polynomial in the size of finitized input programs) and conducive to concrete evaluation of unlifted constructs.

```
(define m (automaton init
            [init : (c → more)]
            [more : (a → more)
                    (d → more)
                    (r → end)]
            [end : ]))
```

Figure 1. An automaton for the language $c(ad)^*r$ [27]

```
(define-syntax automaton
  (syntax-rules (: →)
    [(_ init-state [state : (label → target) ...] ...)
     (letrec ([state
               (lambda (stream)
                 (cond
                   [(empty? stream) true]
                   [else
                    (case (first stream)
                      [(label) (target (rest stream))] ...
                      [else false])]))] ...)
       init-state)]))
```

Figure 2. A macro for executable automata [27]

- We introduce symbolic reflection, which enables SDSL designers to extend symbolic evaluation to unlifted operations, without modifying the SVM or re-implementing (modeling) those operations in terms of the lifted constructs.
- We evaluate the SVM on programs from three different SDSLs, including new languages for OpenCL programming and specifying executable semantics of secure stack machines. We believe that these languages are more sophisticated, and more easily developed, than existing SDSLs.

Outline The rest of the paper is organized as follows. We first present the ROSETTE language (Section 2), and then review the background on symbolic execution and bounded model checking (Section 3). The SVM is presented next (Section 4), followed by a description of our case studies (Section 5). We conclude with a discussion of related work (Section 6).

2. A Solver-Aided Host Language

This section illustrates key features of a sample solver-aided host language, ROSETTE. The ROSETTE language extends Racket [32], a modern descendent of Scheme that includes Scheme's powerful metaprogramming facilities [24]. We start with a brief review of these facilities, borrowing the example of a simple declarative language [27] for specifying executable finite state automata. We then show how to make our sample language solver-aided by embedding it in ROSETTE. The resulting SDSL will allow us to invert, debug, verify and synthesize automata programs.

2.1 Metaprogramming with Macros

Suppose that we want to build a declarative language for implementing finite state automata. A program in this language specifies an automaton at a high level, listing only its states and labeled transitions. Figure 1 shows an example of such a high-level description for an automaton m that recognizes the language $c(ad)^*r$ of Lisp-style identifiers (car, cdr, $caar$, and so on). From this description, we would like to obtain an executable implementation—a function that takes as input a word (a list of symbols) and outputs true or false, depending on whether the automaton accepts the word or not.

There are many ways to implement such a language in Racket. As a starting point, we will reuse the basic implementation from Krishnamurthi's educational pearl [27], which is reproduced in Figure 2. The implementation consists of a macro that pattern-matches an automaton specification, such as the one for m (Figure 1), and expands it into a set of mutually recursive functions. We use blue to distinguish DSL keywords (*e.g.,* `automaton`) from Racket and ROSETTE keywords, which are shown in boldface (*e.g.,* **`lambda`**) and gray boldface (*e.g.,* **`solve`**), respectively.

Racket macros use ellipses to indicate pattern repetition: the syntactic pattern before an ellipsis is repeated zero or more times. Pattern variables matched in the head of a rule (line 3) are available for use in its body. Our macro uses pattern variables and ellipses to create one function for each state of the automaton (lines 4-11). A state function takes as input a list of symbols and transitions to the next state based on the value of the first symbol—or it returns **`false`** if the first symbol is not a valid input in that state. The output of the generated recursive let-expression is the function that implements the initial state of the automaton (line 12). For example, the definition in Figure 1 binds the identifier `m` to a function that implements the initial state of the $c(ad)^*r$ automaton.

Since ROSETTE is embedded in Racket, Figures 1 and 2 form a valid ROSETTE program. ROSETTE programs can be executed, just like Racket programs. We can test our implementation of the `automaton` macro by applying m to a few concrete inputs:

```
> (m '(c a d a d d r))
true
> (m '(c a d a d d r r))
false
```

2.2 Symbolic Values, Assertions, and Solver-Aided Queries

ROSETTE adds to Racket a collection of solver-aided facilities, which enable programmers to conveniently access a powerful constraint solver that can answer interesting questions about program behavior. These facilities are based on three simple concepts: *assertions*, *symbolic values* and *queries*. We use assertions to express desired program behaviors and symbolic values to formulate queries about these behaviors.

Symbolic Values ROSETTE provides two constructs for creating symbolic constants and binding them to Racket variables:

```
(define-symbolic id expr)
(define-symbolic* id expr)
```

The **`define-symbolic`** form creates a single fresh symbolic constant of type *`expr`*, and binds the identifier *`id`* to that constant every time the form is evaluated. The **`define-symbolic*`** form, in contrast, creates a stream of fresh constants, binding *`id`* to the next constant from its stream whenever the form is evaluated. The following example illustrates the difference, and it also shows that symbolic values can be used just like concrete values of the same type. We can store them in data structures and pass them to functions to obtain an output value—either concrete or symbolic:

```
> (define (static)                    > (eq? (static) (static))
    (define-symbolic x boolean?)      true
    x)

> (define (dynamic)                   > (eq? (dynamic) (dynamic))
    (define-symbolic* y number?)      (= y$0 y$1)
    y)
```

The **`define-symbolic`**`[*]` form can only create symbolic constants of type `boolean?` and `number?`. We build all other (finite) symbolic values from these primitives. For example, the following functions create symbolic words that can be used as inputs to our `automaton` programs:

```
(define (word k alphabet)             ; Draws a word of length k
  (for/list ([i k])                   ; from the given alphabet.
    (define-symbolic* idx number?)
    (list-ref alphabet idx)))

(define (word* k alphabet)            ; Draws a word of length
  (define-symbolic* n number?)        ; 0 <= n <= k from the
  (take (word k alphabet) n))         ; given alphabet.
```

Angelic Execution Given a way to create symbolic words, we can now run the `automaton` program m "in reverse," searching for a word of length up to 4 that is accepted by m:

```
> (define w (word* 4 '(c a d r)))
> (define model (solve (assert (m w))))
> (evaluate w model)
'()
> (m '())
true
```

The `(`**`solve`** *`expr`*`)` query implements angelic semantics. It asks the solver for a concrete interpretation of symbolic constants that will cause the evaluation of *`expr`* to terminate without assertion failures. The resulting interpretation, if any, is a first-class value that can be freely manipulated by ROSETTE programs. Our example uses the built-in `evaluate` procedure to obtain the solver's interpretation of the symbolic word w, revealing a bug: the automaton m accepts the word `'()`, which is not in the language $c(ad)^*r$.

Debugging To help debug m, we can ask the solver for a minimal set of expressions in m's implementation that are collectively responsible for its failure to reject the empty word:

```
> (define core (debug [boolean?] (assert (not (m '())))))
> (render core)
(define-syntax automaton
  (syntax-rules (: →)
    [(_ init-state [state : (label → target) ...] ...)
     (letrec ([state
               (lambda (stream)
                 (cond
                   [(empty? stream) true]
                   [else
                    (case (first stream)
                      [(label) (target (rest stream))] ...
                      [else false])]))] ...)
       init-state)]))
```

The `(`**`debug`** `[`*`predicate`*`]` *`expr`*`)` query takes as input an expression whose execution leads to an assertion failure, and a dynamic type predicate specifying which executed expressions should be treated as potentially faulty by the solver. That is, the predicate expresses the hypothesis that the failure is caused by an expression of the given type. Expressions that produce values violating the predicate are assumed to be correct.

The output of a **`debug`** query is a minimal set of program expressions, called a *minimal unsatisfiable core*, that form an irreducible cause of the failure. Expressions outside of the core are *irrelevant* to the failure—even if we replace all of them with values chosen by an angelic oracle, the resulting program will still violate the same assertion. But if we also replace at least one core expression with an angelically chosen value, the resulting program will terminate successfully. In general, a failing expression may have many such cores, but since every core contains a buggy expression, examining one or two cores often leads to the source of the error.

Like interpretations, cores are first-class values. In our example, we simply visualize the core using the utility procedure `render`. The visualization reveals that the sample core consists of the **`cond`** and **`true`** expressions in the implementation of the `automaton` macro. We could change the value produced by either of these expressions in order to satisfy the assertion `(`**`assert`** `(not (m '())))`. In this case, an easy fix is to replace **`true`** with an expression that distinguishes accepting states from non-accepting ones. For example, if we define all (and only) states with no outgoing transitions as accepting, we can repair the `automaton` macro by replacing **`true`** in Figure 2 with the expression `(empty? '(label ...))`.

Verification Having fixed the `automaton` macro, we may want to verify that m correctly implements the language $c(ad)^*r$ for all words of bounded length. The following code snippet shows how to do so by checking m against a golden implementation—Racket's own regular expression matcher:

```
(define M (automaton init
            [init : (c → (? s1 s2))]
            [s1   : (a → (? s1 s2 end reject))
                    (d → (? s1 s2 end reject))
                    (r → (? s1 s2 end reject))]
            [s2   : (a → (? s1 s2 end reject))
                    (d → (? s1 s2 end reject))
                    (r → (? s1 s2 end reject))]
            [end  : ]))
```

Figure 3. An automaton sketch for $c(ad)^+r$

```
(require (rename-in rosette/lib/meta/meta [choose ?]))
(define reject (lambda (stream) false))
```

Figure 4. Sketching constructs for executable automata

```
; Returns a string encoding of the given list of symbols.
; For example, (word->string '(c a r)) returns "car".
(define (word->string w)
  (apply string-append (map symbol->string w)))

; Returns true iff the regular expression regex matches
; the string encoding of the word w.
(define (spec regex w)
  (regexp-match? regex (word->string w)))

> (define w (word* 4 '(c a d r)))
> (verify (assert (eq? (spec #px"^c[ad]*r$" w)  (m w))))
verify: no counterexample found
```

The `(verify expr)` query is the demonic complement of `(solve expr)`; it queries the solver for an interpretation of symbolic values that will cause the evaluation of `expr` to fail. If the query succeeds, the resulting interpretation is called a *counterexample*. In our case, the oracle fails to find a counterexample, and we can be sure that m is correct for all words of length 4 or less. To gain more confidence in m's correctness, we can repeat this query with larger bounds on the length of words, until the solver no longer produces answers in a reasonable amount of time.

Synthesis The `automaton` macro provides a high level interface for specifying automata programs, but it still requires the details of the specification to be filled in manually. To mechanize the construction of automata, we will extend our SDSL with two new keywords, `?` and `reject`, allowing programmers to *sketch* [37] an outline of the desired automaton, which will then be completed by the solver.

Figure 3 shows a sample automaton sketch for the language $c(ad)^+r$. The sketch specifies the states of the automaton and outlines the possible transitions between the states. For example, the state `s1` may accept the label `a` by transitioning to itself, `s2`, or `end`. Alternatively, it may reject the label by transitioning to the special `reject` state, which rejects all words.

Figure 4 shows an implementation of our new sketching constructs. The keyword `reject` is bound to a procedure that always returns false, thus implementing a state that rejects all words. The keyword `?` is bound to a sketching construct, `choose`, imported from a ROSETTE library.[1]

The `(choose expr ..+)` form is a convenience macro for specifying the space of expressions that may be used to complete a sketch. Given n expressions, the macro uses `define-symbolic` to create $n-1$ symbolic boolean values, which are then used to select one of the supplied expressions. For example, `(choose expr1 expr2)` expands into the following code:

```
(local [(define-symbolic tmp boolean?)]
  (cond [tmp expr1]
        [else expr2]))
```

[1] Sketching constructs such as `choose` are implemented in ROSETTE itself using macros, and include advanced constructs for specifying recursive grammars found in other synthesis-enabled languages (*e.g.,* Sketch [37]).

The use of `define-symbolic` ensures that an instance of `choose` picks the same expression every time it is evaluated.

Given the sketch M and a regular expression for $c(ad)^+r$, we can now use the solver to complete the sketch so that the resulting automaton is correct for all words of bounded length:

```
> (define w (word* 4 '(c a d r)))
> (define model
    (synthesize [w]
     (assert (eq? (spec #px"^c[ad]+r$" w) (M w)))))
> (generate-forms model)
(define M
  (automaton init
   [init : (c → s1)]
   [s1   : (a → s2) (d → s2) (r → reject)]
   [s2   : (a → s2) (d → s2) (r → end)]
   [end  : ]))
```

The `(synthesize [input] expr)` query uses the `input` form to specify a set of distinguished symbolic values, which are treated as inputs to the expression `expr`. The result, if any, is an interpretation for the remaining symbolic values that guarantees successful evaluation of `expr` for all interpretations of the `input`. The `generate-forms` utility procedure takes this interpretation and produces a syntactic representation of the completed sketch.

2.3 Symbolic Reflection

SDSL designers can usually treat symbolic values in the same way they treat concrete values of the same type. As demonstrated in our `word` implementation, for example, the `list-ref` procedure works as expected when called with a symbolic index. We say that such procedures are *lifted*.

Because Racket is a rich, evolving language with many libraries, ROSETTE cannot practically lift all of its features. Instead, ROSETTE lifts a small set of core features, while also providing a mechanism for lifting additional Racket constructs from within ROSETTE programs. We call this mechanism *symbolic reflection*.

The key idea behind symbolic reflection is simple. ROSETTE represents a symbolic value of an unlifted data-type as a union of concrete components of that type (see Section 4). Symbolic reflection enables a ROSETTE program to disassemble a symbolic union into its concrete components, apply an unlifted Racket construct to each component, and then reassemble the results into a single symbolic or concrete value. This allows SDSL designers to write a few lines of code and obtain symbolic evaluation of unlifted constructs.

For example, we have used symbolic reflection in the previous section to lift Racket's regular expression matcher to work on symbolic strings:

```
(define (regexp-match? regex str)
  (for/all ([v str])
    (racket/regexp-match? regex v)))
```

The lifted `regexp-match?` function uses the identifier `racket/regexp-match?` to refer to Racket's own regular expression matcher that only works on concrete values. The `for/all` reflection macro applies `racket/regexp-match?` to each concrete string component of `str`, assembling the results of these individual matches into a single concrete or symbolic boolean value.

In contrast to the symbolic reflection approach, using regular expressions (or other advanced language features) in a standard solver-aided tool (*e.g.,* [11]) requires one of two heavyweight implementation strategies. First, the SDSL designer could modify the tool's translator to constraints to include support for regular expressions, which is complicated by the need for a specialized solver that can reason about strings (*e.g.,* [35]). Second, he could re-implement (or "model") regular expression facilities in terms of the simpler constructs supported by the tool. Both of these approaches involve writing hundreds of lines of tricky code.

3. Design Space of Precise Symbolic Encodings

There are many ways to compile solver-aided queries to logical constraints (*e.g.,* [3, 8–11, 15, 16, 22, 31, 37, 41]). Most existing approaches employ an encoding strategy that is based either on symbolic execution [23] or on bounded model checking [11]. In this section, we review these two standard approaches and illustrate why a new approach is needed for encoding solver-aided host languages.

3.1 Basic Design Decisions

The design of a symbolic encoding technique involves several basic design decisions [28], including whether the technique is static or dynamic; whether and how it merges symbolic states from different paths; and how it handles loops and recursion. Symbolic execution engines (such as [8–10, 22]) are at the dynamic end of the spectrum: they execute the target program path-by-path, performing no state merging and using heuristics to determine how many times to execute individual loops. Bounded model checkers (such as [11, 15, 16, 18, 31, 37, 40]) and extended static checkers (such as [3, 41]) form the static end of the spectrum: they finitize the representation of the target program by statically unrolling loops up to a given bound, and then encode it by merging states from different paths at each control-flow join.

We illustrate both approaches on the sample program in Figure 5a. The program is implemented in a Python-like language with symbolic constants (line 9), assertions (line 11) and solver-aided queries (line 8). The procedure `revPos` takes as input an immutable list of integers and reverses it, keeping only the positive elements. The **`solve`** query searches for a two-element list `xs` on which `revPos` produces a list of the same length.

3.2 Symbolic Execution

Symbolic execution answers queries by exploring paths in the program's *execution tree* [23]. In the case of a **`solve`** query, the tree is searched for a path that terminates without any assertion failures. Figure 5b shows the execution tree for our example query. The nodes represent program states, and the edges represent transitions between states. Each transition is labeled with a branch condition, expressed in terms of symbolic inputs, that must be true for the transition to take place. The conjunction of edge labels along a given path is called a *path condition*, and a path is feasible only if its path condition and any reachable assertions (also expressed in terms of symbolic inputs) are satisfiable. If a path becomes infeasible, the execution backtracks and tries a new path.

Feasibility of paths is tested by compiling the path condition and assertions to a constraint system, whose satisfiability is checked with an off-the-shelf solver. Because program state remains largely concrete along a given execution path, many tricky-to-encode operations, such as the list manipulations in our example, are evaluated concretely, which eliminates the need to compile them to constraints. This simplifies implementation of symbolic execution engines and lessens the burden on the underlying solver.

But path-based encodings have a well-known disadvantage: since the number of paths in a (loop-free) program is generally exponential in its size, a symbolic execution engine may need to make exponentially many calls to the underlying solver. In our example, an engine may have to explore $O(2^n)$ failing paths for an input of size n before it finds the single path that leads to a successful evaluation of the target assertion. Many heuristics have been developed to manage this problem (see [7] for an overview), but they are only applicable to solver-aided queries that can be answered by finding a single (failing or successful) path through the program—namely, verification and angelic execution. Synthesis and debugging queries require reasoning about all paths simultaneously, which, in the case of symbolic execution, would be encoded as an exponentially-sized disjunction of path constraints. As a result, symbolic execution is limited in the kind of solver-aided queries that it can efficiently answer.

3.3 Bounded Model Checking

Techniques based on bounded model checking, in contrast, do not suffer from path explosion, enabling both synthesis [37] and debugging [20] queries. Given a target program, a bounded model checker first transforms it by inlining all function calls, unrolling all loops by a fixed amount, renaming program variables so that each one is assigned exactly once, and representing control flow merges explicitly with (guarded) ϕ expressions (see, *e.g.,* [11, 16, 41] for details). The result is an acyclic program in Static Single Assignment form. Figure 5c shows the transformed code for our example.

To compile a finitized program to constraints, a bounded model checker encodes the value of each defined variable in a suitable theory and then uses these values in the translation of the target assertions. This process is complicated by the presence of the state-merging ϕ expressions. Once two concrete values from different branches are *logically merged* with a ϕ expression, as in the definition of `ps1`, their representation becomes an opaque symbolic value and all operations that consume that value must also be translated to symbolic values and constraints. Unlike symbolic execution engines, bounded model checkers tend to evaluate very few operations concretely and must therefore be able to symbolically compile all constructs in their target programming language. This also has the effect of offloading all reasoning about program semantics to the solver. As a result, encodings produced by bounded model checkers, while compact in size, are harder to solve than path constaints.

In our example, a bounded model checker will need to know how to encode list values and operations on the list data type (in order to represent the value of, *e.g.,* `ps2`). Producing such an encoding is not straightforward even when the underlying solver supports the theory of lists as Z3 [13] does, for example. Since the list length operation is not included in the theory, it must be either axiomatized (using expensive universal quantification) or finitized and encoded by the bounded model checker like any other procedure.

3.4 Encoding Solver-Aided Host Languages

As illustrated in Section 2, hosting SDSLs requires advanced language features (such as macros and first-class procedures); a rich set of libraries and datatypes; and support for a variety of solver-aided queries and programming styles. Neither symbolic execution nor bounded model checking are well suited for compiling such a host language to constraints. Building a bounded model checker for a host language would involve a heroic engineering effort, resulting in a heavyweight symbolic compiler that performs program finitization, advanced static analyses (see, *e.g.,* [16, 40]), and logical encoding of complex language features. Our first attempt at compiling the ROSETTE language was based on bounded model checking, and we abandonded it after three months of laborious implementation work, when the system became too complicated. Symbolic execution offers an attractive alternative to heavyweight compilation by increasing opportunities for concrete evaluation of hard-to-encode constructs. But the price of this simplicity is a loss of versatility, since path-based encoding is only practical for answering verification and angelic execution queries. A lightweight compiler for a host language therefore needs a way to combine symbolic execution with a state merging strategy that enables concrete evaluation.

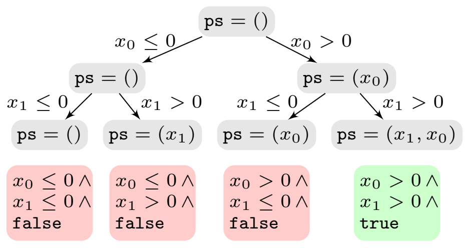
4. A Lightweight Symbolic Virtual Machine

In this section, we present a new technique for precise symbolic encoding of solver-aided host languages. Our solution is to combine key elements of symbolic execution and bounded model checking in a *symbolic virtual machine with type-driven state merging* (SVM).

```
1  def revPos(xs):
2    ps = ()
3    for x in xs:
4      if x > 0:
5        ps = cons(x, ps)
6    return ps

8  solve:
9    xs = (x0, x1)
10   ps = revPos(xs)
11   assert len(ps) == len(xs)
```

(a) Sample program



ps = ()
$x_0 \leq 0$
$x_0 > 0$
ps = ()
ps = (x_0)
$x_1 \leq 0$
$x_1 > 0$
$x_1 \leq 0$
$x_1 > 0$
ps = ()
ps = (x_1)
ps = (x_0)
ps = (x_1, x_0)
$x_0 \leq 0 \wedge x_1 \leq 0 \wedge$ false
$x_0 \leq 0 \wedge x_1 > 0 \wedge$ false
$x_0 > 0 \wedge x_1 \leq 0 \wedge$ false
$x_0 > 0 \wedge x_1 > 0 \wedge$ true

(b) Symbolic execution tree for (a)

```
solve:
  xs = (x0, x1)
  ps0 = ()
  for0 = xs != ()
  x0 = φ(for0, car(xs), 0)
  ps1 = φ(for0 and x0 > 0, cons(x0, ps0), ps0)
  for1 = for0 and cdr(xs) != ()
  x1 = φ(for1, car(cdr(xs)), 0)
  ps2 = φ(for1 and x1 > 0, cons(x1, ps1), ps1)
  for2 = for1 and cdr(cdr(xs)) != ()
  assert ! for2
  assert len(ps2) == len(xs)
```

(c) Acyclic finitized representation of (a)

Figure 5. Logical encoding of a sample solver-aided query (a), using symbolic execution (b) and bounded model checking (c).

Like a symbolic execution engine, an SVM executes the target program. But the SVM execution graph is not a tree—it is a DAG, in which states from different paths are merged at every control join point. Values of the same primitive type (*e.g.,* boolean or integer) are merged logically, as in bounded model checking, resulting in an opaque symbolic value. All other values are merged structurally, based on their type, resulting in concrete values or in transparent *symbolic unions* that can be unpacked for concrete evaluation.

We first illustrate our approach on the example program from the previous section, showing how careful construction of symbolic unions increases opportunities for concrete evaluation (compared to bounded model checking), while ensuring efficient evaluation (compared symbolic execution). We then present SVM evaluation and merging rules on a small but expressive solver-aided language. The section concludes with brief correctness and efficiency arguments, as well as a discussion of practical consequences of our design.

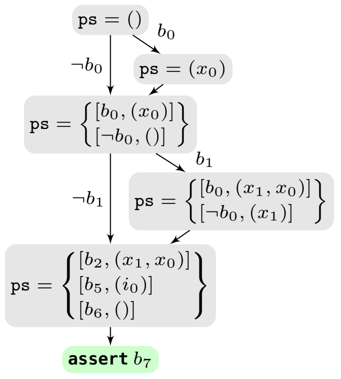
4.1 Example

Figure 6 shows the SVM execution DAG for the sample program in Figure 5a. The labels of the form b_k and i_k to refer to the opaque symbolic booleans and integers created during symbolic evaluation. The definitions of these values are shown next to the graph. We use ϕ to denote the logical merge (*i.e.,* if-then-else) operator and $\oplus$ to denote bitwise disjunction of two integer values.[2]

The execution starts by initializing the variable ps to the empty list. It proceeds by evaluating both branches of the condition on line 4 independently: the true-branch updates the variable ps to the list (x_0), and the empty false-branch leaves its value unchanged. The resulting states are then merged, binding the variable ps to the symbolic union $\{[b_0, (x_0)]\,[\neg b_0, ()]\}$. As its textual representation suggests, a symbolic union is a set of guarded values, in which the guards are, by construction, disjoint (*i.e.,* at most one of them is true in every concrete interpretation).

In the second iteration of the loop, the true-branch (guarded by b_1) updates the value of ps by unpacking the symbolic union $\{[b_0, (x_0)]\,[\neg b_0, ()]\}$ and concretely executing the cons operation on each of the union's components. The states from both branches are once again merged, ensuring that all list values of the same length are collapsed into a single list by performing (type-driven) merging of their elements. Because list elements are primitive values (integers), they are merged logically (see, *e.g.,* i_0).

After the loop exits, the assertion on line 11 is evaluated bottom-up, by computing and comparing the lengths of the lists ps and xs. The length of ps is obtained by applying the len operation to the individual components of ps, and combining the resulting guarded concrete values into a single symbolic value, i_1. The symbolic boolean value b_7 encodes the result of comparing i_1 and 2 (the



ps = ()
b_0
$\neg b_0$
ps = (x_0)
ps = $\left\{ \begin{matrix} [b_0, (x_0)] \\ [\neg b_0, ()] \end{matrix} \right\}$
b_1
$\neg b_1$
ps = $\left\{ \begin{matrix} [b_0, (x_1, x_0)] \\ [\neg b_0, (x_1)] \end{matrix} \right\}$
ps = $\left\{ \begin{matrix} [b_2, (x_1, x_0)] \\ [b_5, (i_0)] \\ [b_6, ()] \end{matrix} \right\}$
assert b_7

$$
\begin{aligned}
b_0 &= x_0 > 0 \\
b_1 &= x_1 > 0 \\
b_2 &= b_0 \wedge b_1 \\
b_3 &= \neg b_0 \wedge b_1 \\
b_4 &= b_0 \wedge \neg b_1 \\
b_5 &= b_3 \vee b_4 \\
b_6 &= \neg b_0 \wedge \neg b_1 \\
i_0 &= \phi(b_1, x_1, x_0) \\
i_1 &= \phi(b_2, 2, 0) \oplus \\
&\quad\ \phi(b_5, 1, 0) \oplus \\
&\quad\ \phi(b_6, 0, 0) \\
b_7 &= (i_1 = 2)
\end{aligned}
$$

Figure 6. SVM encoding for the program in Figure 5a

length of xs) for equality. This value, together with the primitive values on the right of Figure 6, comprise the final encoding of the program, which is free of lists and unions.

The final encoding is a polynomially-sized formula in the theory of bit vectors. Our merging strategy enables all list operations to be evaluated concretely, and it also ensures that there is no state explosion during evaluation. The cardinality of the symbolic union representing the state of the variable ps grows polynomially rather than exponentially with the size of the execution DAG, which is, in this case, determined by the length of the input list xs. If we were to execute our sample program on a list of n symbolic values, the symbolic union representing the final state of ps would contain $n+1$ guarded lists. Because lists of the same length are merged element-wise, filtering a list of n symbolic values results in a symbolic union with $n+1$ merged lists of length 0 through n. All primitive symbolic values (list guards and elements) created during these merges are also polynomially-sized, since primitive values are merged logically, as in bounded model checking.

4.2 A core solver-aided language

We describe the SVM evaluation process on H_L, a small solver-aided host language shown in Figure 7. The language extends core Scheme with mutation [33] to include symbolic values, assertions and the **solve** query. We omit other queries for brevity, since their evaluation is analogous to that of the **solve** query.

Like Scheme, H_L supports first-class procedures, procedure application, conditional execution and mutation expressions. It also includes a set of built-in procedures for operating on its five core[3] data types: booleans, finite precision integers, immutable

[2] We assume that all integers and integer operations are modeled using finite-precision bitvectors. Theory of integers can also be used if the underlying solver supports it, with a suitable adjustment to the SVM evaluation rules.

[3] In Scheme [33], procedures cons, car and cdr operate on *pairs*, and lists are represented as **null** terminated pairs. We omit the pair data type to simplify the presentation, and restrict pair operations to work only on lists.

$$
\begin{array}{llcl}
\text{expressions} & e & ::= & l \mid x \mid (\textbf{lambda}\,(x)\,e) \mid (e\,e\ldots) \mid (\textbf{if}\,e\,e\,e) \mid \\
 & & & (\textbf{set!}\,x\,e) \mid (\textbf{assert}\,e) \mid (\textbf{solve}\,e) \\
 & l & ::= & \textbf{true} \mid \textbf{false} \mid \text{integer literal} \mid \textbf{null} \\
 & x & ::= & \text{identifier} \mid \texttt{equal?} \mid \texttt{union?} \mid \texttt{number?} \mid \texttt{boolean?} \mid \\
 & & & \texttt{procedure?} \mid \texttt{list?} \mid \texttt{cons} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{length} \mid \\
 & & & \texttt{=} \mid \texttt{<} \mid \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{\textbackslash} \mid \ldots \\
\text{definitions} & d & ::= & (\textbf{define}\,x\,e) \mid (\textbf{define-symbolic}\,x\,e) \\
\text{forms} & f & ::= & d \mid e \\
\text{programs} & p & := & f\ldots
\end{array}
$$

Figure 7. The H_L language

lists, procedure objects, and symbolic unions. Figure 7 shows (the identifiers for) a few sample built-in procedures.

Integers and booleans can be either concrete values, symbolic constants (introduced by `define-symbolic`), or symbolic expressions (obtained, *e.g.,* by applying primitive procedures to symbolic constants). Lists are concrete, immutable, finite sequences of H_L values. Procedures are created using `lambda` expressions in the usual way. Symbolic unions are sets of guarded values, which are pairs of the form $[b, v] \in \mathcal{B} \times (\mathcal{V} \setminus \mathcal{U})$, where $\mathcal{B}$ and $\mathcal{U}$ stand for all boolean and union values, and $\mathcal{V}$ stands for all H_L values. The guards in each union are disjoint with respect to all concrete interpretations.

Thanks to the inclusion of lists, first-class procedures and mutation, H_L forms the core of a rich solver-aided language that supports multiple programming styles. For example, lists and higher order functions facilitate a functional programming style, while closures and mutation enable construction of mutable storage—*e.g.,* vectors and objects—for use in imperative and object-oriented programming. We therefore show how to build an SVM for this core language, but note that the approach presented here can be naturally extended with rules for handling richer features and data types. Our prototype SVM implements direct evaluation and merging rules for (im)mutable vectors and user-defined record types, although both could be represented and evaluated in terms of the core features.

4.3 SVM Evaluation and Merging Rules

Figure 8 shows a representative subset of the SVM evaluation rules for H_L. A rule $\langle f, \sigma, \pi, \alpha\rangle \to \langle v, \sigma', \pi', \alpha'\rangle$ says that a succesful execution of the form f in the program state $\langle \sigma, \pi, \alpha\rangle$ results in the value v and the state $\langle \sigma', \pi', \alpha'\rangle$. A program state consists of the program store σ, the path condition π, and the assertion store α. The program store maps program identifiers to values and procedure pointers to procedure objects, as described below; the path condition is a boolean value encoding the branch decisions taken to reach the current state; and the assertion store is the set of boolean values (*i.e.,* constraints) that have been asserted so far. We use $\langle f, \sigma, \pi, \alpha\rangle \to \bot$ to indicate that the evaluation of f in the given state leads to a failure. Meta-variables v, w, and u denote any H_L value, while b, i, l, and pp stand for booleans, integers, lists, and procedures, respectively.

The evaluation process starts with the Top rule, which populates the program store with bindings from built-in procedure identifiers (such as `+`) to procedure pointers, which are in turn bound to the corresponding procedure objects (such as $+$). Procedure pointers, rather than the objects themselves, represent H_L procedure values, and a new pointer is created with each evaluation of a `lambda` expression (see the Proc rule).

Rules Def1 and Pl1 demonstrate the basic mechanisms for handling primitive symbolic values (booleans and integers). Def1 uses the constant constructor $\llparenthesis x : t \rrparenthesis$ to create a new symbolic constant x of type t. Pl1 uses the expression constructor $\llparenthesis i_1 + i_2 \rrparenthesis$ to create a value that represents the sum of i_1 and i_2. The expression constructor evaluates its arguments, creating a symbolic expression whenever either of the arguments is symbolic. Symbolic expressions are represented as DAGs that share common subexpressions.

$$
\textsc{Top}\;\dfrac{\begin{array}{c}\sigma_0 = \{[\texttt{+} \mapsto pp_+]\,[pp_+ \mapsto +]\ldots\} \quad \pi = true \quad \alpha_0 = \{\} \\ \langle f_i, \sigma_{i-1}, \pi, \alpha_{i-1}\rangle \to \langle v_i, \sigma_i, \pi, \alpha_i\rangle \text{ where } f_i \in \{f_1, \ldots, f_n\}\end{array}}{\langle f_1 \ldots f_n, \sigma_0, \pi, \alpha_0\rangle \to \langle v_n, \sigma_n, \pi, \alpha_n\rangle}
$$

$$
\textsc{Proc}\;\dfrac{\lambda = (\textbf{lambda}\,(x)\,e) \quad pp \notin dom(\sigma)}{\langle \lambda, \sigma, \pi, \alpha\rangle \to \langle pp, \sigma[pp \mapsto \lambda], \pi, \alpha\rangle}
$$

$$
\textsc{Def1}\;\dfrac{\begin{array}{c}\langle e, \sigma, \pi, \alpha\rangle \to \langle v, \sigma_0, \pi, \alpha_0\rangle \quad u = \llparenthesis x : \sigma_0(v)\rrparenthesis \\ \sigma_0(v) \in \{boolean?, number?\} \quad \forall_{y \in dom(\sigma_0)} \sigma_0(y) \neq u\end{array}}{\langle (\textbf{define-symbolic}\,x\,e), \sigma, \pi, \alpha\rangle \to \langle void, \sigma_0[x \mapsto u], \pi, \alpha_0\rangle}
$$

$$
\textsc{Pl1}\;\dfrac{\begin{array}{c}\langle e_0, \sigma, \pi, \alpha\rangle \to \langle pp_+, \sigma_0, \pi, \alpha_0\rangle \\ \langle e_1, \sigma_0, \pi, \alpha_0\rangle \to \langle i_1, \sigma_1, \pi, \alpha_1\rangle \;\; \langle e_2, \sigma_1, \pi, \alpha_1\rangle \to \langle i_2, \sigma_2, \pi, \alpha_2\rangle\end{array}}{\langle (e_0\; e_1\; e_2), \sigma, \pi, \alpha\rangle \to \langle \llparenthesis i_1 + i_2\rrparenthesis, \sigma_2, \pi, \alpha_2\rangle}
$$

$$
\textsc{If1}\;\dfrac{\begin{array}{c}\langle e_0, \sigma, \pi, \alpha\rangle \to \langle v_0, \sigma_0, \pi, \alpha_0\rangle \quad b = isTrue(v_0) \quad isSymbolic(b) \\ \langle e_1, \sigma_0, \pi_1, \alpha_0\rangle \to \langle v_1, \sigma_1, \pi_1, \alpha_1\rangle \;\; \pi_1 = \llparenthesis \pi \wedge b\rrparenthesis \;\; isTrue(\pi_1) \neq false \\ \langle e_2, \sigma_0, \pi_2, \alpha_0\rangle \to \langle v_2, \sigma_2, \pi_2, \alpha_2\rangle \;\; \pi_2 = \llparenthesis \pi \wedge \neg b\rrparenthesis \;\; isTrue(\pi_2) \neq false \\ \sigma_3 = \{x \mapsto \mu(b, \sigma_1(x), \sigma_2(x)) \mid x \in dom(\sigma_0), isIdentifier(x)\} \cup \\ \{pp \mapsto \sigma_0(pp) \mid pp \in dom(\sigma_0), isProcedure(pp)\} \cup \\ \{y \mapsto \sigma_1(y) \mid y \in dom(\sigma_1) \setminus dom(\sigma_0)\} \cup \\ \{y \mapsto \sigma_2(y) \mid y \in dom(\sigma_2) \setminus dom(\sigma_0)\}\end{array}}{\langle (\textbf{if}\; e_0\; e_1\; e_2), \sigma, \pi, \alpha\rangle \to \langle \mu(b, v_1, v_2), \sigma_3, \pi, \alpha_1 \cup \alpha_2\rangle}
$$

$$
\textsc{Co1}\;\dfrac{\begin{array}{c}\langle e_0, \sigma, \pi, \alpha\rangle \to \langle pp_{cons}, \sigma_0, \pi, \alpha_0\rangle \\ \langle e_1, \sigma_0, \pi, \alpha_0\rangle \to \langle v_1, \sigma_1, \pi, \alpha_1\rangle \;\; \langle e_2, \sigma_1, \pi, \alpha_1\rangle \to \langle v_2, \sigma_2, \pi, \alpha_2\rangle \\ isUnion(v_2) \quad u = \{[b, cons(v_1, l)] \mid [b, l] \in v_2, isList(l)\} \\ |u| > 1 \quad b_u = \llparenthesis \pi \Rightarrow \bigvee_{[b,l] \in u} b\rrparenthesis\end{array}}{\langle (e_0\; e_1\; e_2), \sigma, \pi, \alpha\rangle \to \langle u, \sigma_2, \pi, \alpha_2 \cup \{b_u\}\rangle}
$$

$$
\textsc{Ap1}\;\dfrac{\begin{array}{c}\langle e_0, \sigma, \pi, \alpha\rangle \to \langle pp, \sigma_0, \pi, \alpha_0\rangle \quad \sigma_0(pp) = (\textbf{lambda}\,(x)\,e) \\ \langle e_1, \sigma_0, \pi, \alpha_0\rangle \to \langle v_1, \sigma_1, \pi, \alpha_1\rangle \\ \langle e[x := y], \sigma_1[y \mapsto v_1], \pi, \alpha_1\rangle \to \langle v, \sigma_2, \pi, \alpha_2\rangle \quad \text{fresh } y\end{array}}{\langle (e_0\; e_1), \sigma, \pi, \alpha\rangle \to \langle v, \sigma_2, \pi, \alpha_2\rangle}
$$

$$
\textsc{Ap2}\;\dfrac{\begin{array}{c}\langle e_0, \sigma, \pi, \alpha\rangle \to \langle v_0, \sigma_0, \pi, \alpha_0\rangle \quad \langle e_1, \sigma_0, \pi, \alpha_0\rangle \to \langle v_1, \sigma_1, \pi, \alpha_1\rangle \\ v_0 = \{[b_0, pp_0]\,[b_1, pp_1]\} \quad b_1 = \llparenthesis \neg b_0\rrparenthesis \\ \sigma_2 = \sigma_1[x_0 \mapsto b_0, x_1 \mapsto pp_0, x_2 \mapsto pp_1, x_3 \mapsto v_1] \\ \text{fresh } x_0, x_1, x_2, x_3 \\ \langle (\textbf{if}\; x_0\; (x_1\; x_3)\; (x_2\; x_3)), \sigma_2, \pi, \alpha_1\rangle \to \langle v, \sigma_3, \pi, \alpha_2\rangle\end{array}}{\langle (e_0\; e_1), \sigma, \pi, \alpha\rangle \to \langle v, \sigma_3, \pi, \alpha_2\rangle}
$$

$$
\textsc{As1}\;\dfrac{\langle e, \sigma, \pi, \alpha\rangle \to \langle v, \sigma_1, \pi, \alpha_1\rangle \quad isTrue(v) = false}{\langle (\textbf{assert}\; e), \sigma, \pi, \alpha\rangle \to \bot}
$$

$$
\textsc{As2}\;\dfrac{\langle e, \sigma, \pi, \alpha\rangle \to \langle v_0, \sigma_1, \pi, \alpha_1\rangle \quad b = isTrue(v_0) \quad b \neq false}{\langle (\textbf{assert}\; e), \sigma, \pi, \alpha\rangle \to \langle void, \sigma, \pi, \alpha \cup \{\llparenthesis \pi \Rightarrow b\rrparenthesis\}\rangle}
$$

$$
\textsc{SQ1}\;\dfrac{\langle e, \sigma, \pi, \alpha\rangle \to \langle v, \sigma_0, \pi, \alpha_0\rangle \quad \exists \mathcal{M} \vdash \llparenthesis \textstyle\bigwedge_{a \in \alpha_0} a\rrparenthesis}{\langle (\textbf{solve}\; e), \sigma, \pi, \alpha\rangle \to \langle modelToList(\mathcal{M}), \sigma_0, \pi, \alpha\rangle}
$$

$$
isTrue(v) := \begin{cases} v & \text{if } isBool(v) \\ \llparenthesis \bigvee_{[b_i, v_i] \in v} b_i \wedge (isBool(v_i) \Rightarrow v_i)\rrparenthesis & \text{if } v = \{[b_i, v_i]\ldots\} \\ true & \text{otherwise} \end{cases}
$$

$$
\llparenthesis i_1 + i_2\rrparenthesis \;:= \begin{cases} i_1 + i_2 & \text{if } i_1, i_2 \in Z \\ (\mathsf{int}\; +\; i_1\; i_2) & \text{otherwise} \end{cases}
$$

$$
\llparenthesis x : boolean?\rrparenthesis := (\mathsf{bool}\; x) \qquad \llparenthesis x : number?\rrparenthesis := (\mathsf{int}\; x)
$$

Figure 8. A subset of the SVM evaluation rules for H_L

Rule If1 illustrates type-driven state merging. The rule employs the meta-function μ, defined in Figure 9, to merge states from alternative execution paths. The key idea is to partition H_L values into classes, and to merge values in the same class without creating symbolic unions. Unions are used only to merge values from different classes. The boolean and integer types, for example, form two of these classes; their members are merged logically using ϕ expressions. If other primitive types were added to the language, they would also be merged logically. Two lists of length k are merged element-wise. In general, we use such structural merging for non-primitive immutable values, with the details of the merging process specific to their type. Pointer values, such as procedure pointers, are merged only if they represent the same location in the program store,

$$
\begin{array}{lll}
u \approx_{Prim} v & := & isBool(v) \wedge isBool(u) \vee isInt(v) \wedge isInt(u) \\
u \approx_{List} v & := & isList(v) \wedge isList(u) \wedge len(v) = len(u) \\
u \approx_{Ptr} v & := & u = v \\
u \approx v & := & (u \approx_{Ptr} v) \vee (u \approx_{Prim} v) \vee (u \approx_{List} v) \\
b \circ u & := & \bigcup_{[b_i, u_i] \in u} \{[\llparenthesis b \wedge b_i \rrparenthesis, u_i]\} \\
\mu(b, u, v) & := &
\end{array}
$$

$$
\begin{cases}
u & \text{if } b = true \\
v & \text{if } b = false \\
u & \text{if } u = v \\
\llparenthesis \phi(b, u, v) \rrparenthesis & \text{if } u \approx_{Prim} v \\
(w_0, \ldots, w_n) & \text{if } u \approx_{List} v \text{ and } w_i = \mu(b, u[i], v[i]) \text{ for } 0 \leq i \leq len(u) \\
\mu(\llparenthesis \neg b \rrparenthesis, v, u) & \text{if } \neg isUnion(u) \text{ and } isUnion(v) \\
u' \cup v' & \text{if } isUnion(u), \neg isUnion(v), \nexists [b_i, u_i] \in u.\, u_i \approx v, \\
 & \quad u' = b \circ u, \text{ and } v' = \{[\llparenthesis \neg b \rrparenthesis, v]\} \\
u' \cup v' & \text{if } isUnion(u), \neg isUnion(v), [b_i, u_i] \in u, u_i \approx v, \\
 & \quad u' = b \circ (u \setminus \{[b_i, u_i]\}) \text{ and} \\
 & \quad v' = \{[\llparenthesis b \Rightarrow b_i \rrparenthesis, \mu(b, u_i, v)]\} \\
w \cup u' \cup v' & \text{if } u = \{[b_i, u_i], \ldots\}, v = \{[b_j, v_j], \ldots\}, \\
 & \quad w = \{[\llparenthesis b \wedge b_i \vee \neg b \wedge b_j \rrparenthesis, \mu(b, u_i, v_j)] \mid u_i \approx v_j\}, \\
 & \quad u' = b \circ \{[b_i, u_i] \mid \nexists v_j . u_i \approx v_j\}), \text{and} \\
 & \quad v' = \llparenthesis \neg b \rrparenthesis \circ \{[b_j, v_j] \mid \nexists u_i . u_i \approx v_j\}) \\
\{[b, u]\, [\llparenthesis \neg b \rrparenthesis, v]\} & \text{otherwise}
\end{cases}
$$

Figure 9. Merging function for H_L values

implementing sound tracking of aliasing relationships. Unions are merged member-wise, so that the resulting union contains at most one member from each class of values.

We include one sample rule, CO1, that shows how built-in procedures handle union arguments. The primitive *cons* procedure adds a value to the beginning of a concrete list. Given a value v_1 and a union v_2 containing multiple lists, the rule produces a new union by applying *cons* to v_1 and every list in v_2. To ensure soundness, the rule also extends the assertion store with a constraint stating that at least one list guard must be true if the execution takes the path π. Since union guards are disjoint by construction, this forces v_2 (and the output u) to evaluate to exactly one list in every concrete interpretation that satisfies π. In other words, rules such as CO1 emit assertions to ensure that unions flowing into built-in procedures have the right dynamic types.

Rule AP2 illustrates the evaluation of an application expression in which the first (procedure) subexpression evaluates to a symbolic union. The rule combines the guarded results and effects of applying all procedures in the union. This is analogous to how existing bounded model checkers for object oriented languages (*e.g.,* [16]) handle dynamic dispatch.

Evaluation of other H_L rules, except for SQ1, is standard. Rule SQ1 ensures that a (**solve** e) query succeeds only if there is an interpretation of symbolic constants (*i.e.,* a model) satisfying all assertions in the state obtained by evaluating the expression e, which includes the assertions collected before the evaluation of **solve**. That is, the **solve** query searches for a feasible path through the entire program up to and including e. Other queries are evaluated similarly—by searching for a model or a minimal unsatisfiable core of a logical formula that combines the constraints in the assertion store. For example, the verification query searches for a model of the formula $\llparenthesis \bigvee_{b \in \alpha} \neg b \rrparenthesis$. These formulas are easy to compile to input languages of modern solvers, since all constraints in the assertion store are expressed solely in terms of symbolic primitives.

4.4 Correctness of SVM Evaluation

It can be shown by structural induction on H_L that the SVM evaluation rules are correct in the following sense [14]: the program state produced by each evaluation step represents *all* and *only* those concrete states that could be reached via some fully concrete execution, using the semantics of core Scheme with mutation [33]. An important point to note is that this is only possible because H_L excludes Scheme's `eq?` and `eqv?` operators, which, unlike `equal?`, allow Scheme programs to distinguish between two concrete immutable objects that represent the same value. If these operators were allowed in H_L, then our strategy for merging lists would be unsound, and they would have to be treated as pointers.

To see that the rules correctly encode the H_L subset of Scheme, first note that they reduce to Scheme rules in the absence of symbolic values (*e.g.,* PL1 and AP1). Next, observe that the SVM behaves just like a bounded model checker on programs that require only values of the same primitive type to be merged. Finally, recall that the merging function μ soundly combines non-primitive values (in particular, by precisely tracking aliasing relationships); that the rules for applying built-in procedures (*e.g.,* CO1) enforce the relevant (dynamic) type constraints; and that the generic rules for procedure application (*e.g.,* AP2) work correctly when the target of a call may be one of several procedures.

4.5 Efficiency of SVM Evaluation

Like bounded model checking, and unlike symbolic execution, the SVM evaluation is efficient in that it is free of path or state explosion.[4] In particular, all values generated during SVM evaluation, including symbolic unions, are polynomial in the size of the (finitized) input program. We show this for unions, and note that similar reasoning can be applied to derive polynomial bounds on the size of symbolic expressions and concrete values.

Precise symbolic encoding techniques operate, implicitly or explicitly, on finite acyclic programs, obtained from general programs by unwinding loops finitely many times and inlining calls to user-defined (non-primitive) procedures. Symbolic execution and the SVM perform this finitization implicitly, during evaluation (see, *e.g.,* rules AP1 and AP2). In bounded model checking, the finitization step is explicit [16]. The SVM generates unions that are at worst linear in the size of such finitized programs.

Without loss of generality, suppose that the SVM is applied to an already finitized H_L program $P = (f_1 \ldots f_n)$. Let $|P|$ be the number of forms that comprise the syntactic representation of P; that is, $|P| = |f_1| + \ldots + |f_n|$, where $|x| = 1$ for an identifier x, $|(e_1 \ldots e_k)| = 1 + |e_1| + \ldots + |e_k|$ for an application expression, and so on. Because P is finitized, it is free of loops, recursion and non-primitive procedure calls. As a result, the only forms in P that may produce a union value are **if** expressions (see IF1) and primitive procedure applications (see, *e.g.,* CO1). These unions are linear in $|P|$, as shown next.

An **if** expression creates unions by applying the merging function μ to values from different branches. Recall that μ maintains the following invariant on the structure of unions it creates: a union contains at most one boolean, at most one integer, at most one list of length k, and any number of distinct procedure pointers. A procedure pointer may appear in a union created by μ only if the evaluation of some form in P generates such a pointer. The same is true for a list of length k. To show that unions produced by μ are linear in $|P|$, we therefore need to establish that the SVM evaluation of P creates at most $O(|P|)$ distinct procedure pointers and at most $O(|P|)$ differently-sized lists.

Since P is finite, the SVM evaluates every **lambda** expression in P at most once. As a result, evaluating P produces at most $|P|$ distinct procedure pointers. Similarly, P contains at most $|P|$ applications of the `cons` primitive procedure, which grows the size of a concrete input list by 1. Since the application of `cons` is the only way to construct lists in H_L, P cannot construct a list that is longer than $|P|$. Hence, P can create at most $|P|$ differently-sized lists.

[4] Of course, state explosion may still happen in the solver.

Having shown that **if** expressions produce unions of size $O(|P|)$, we can now show that the same bound holds for unions produced by primitive procedure applications. Recall that all members of a union are themselves either symbolic constants, or symbolic expressions, or concrete values. When a primitive H_L procedure (*e.g.,* `cons`) is applied to such a value, it yields another symbolic or concrete value, but never a union. Hence, when a primitive procedure is applied to all members of a union of size $O(|P|)$, as in the CO1 rule, the size of the resulting union is also $O(|P|)$.

4.6 Loops and Recursion

Although the SVM encodes only finite executions of programs to formulas (see SQ1), our evaluation rules do not artifically finitize executions by explicitly bounding the depth of recursion. Instead, the SVM assumes that its input programs are *self-finitizing*, and it fails to terminate on those that are not. This design decision is intentional, for two reasons.

First, many SDSLs implementations and programs use loops and recursion only to iterate over data structures (*e.g.,* lists), and all such code is self-finitizing. Since the shape of data structures remains concrete during SVM evaluation, looping or recursive traversals over these structures are automatically unwound as many times as needed. This is true for our running example (Fig. 5a), as well as the code in Section 2.

Second, the SVM is designed for use within a solver-aided host language, which provides facilities for defining bounded looping constructs, if these are needed by an SDSL. In our prototype language, for example, bounded looping constructs are easy to implement with the help of macros [39]. We therefore leave the control of finitization to SDSL designers, allowing them to define finitization behaviors that are best suited to their application domain.

4.7 Symbolic Reflection

In Section 2, we introduced symbolic reflection as a mechanism for extending symbolic evaluation to advanced language features, such as regular expressions, that are hard to support in classic solver-aided tools. The SVM supports symbolic reflection by exposing symbolic unions and primitives as first-class values in the host language. Our host language provides several useful operations on unions, such as the **for/all** lifting macro from Section 2.3 (which is implemented much like the CO1 rule). ROSETTE code can also obtain the cardinality of a union, which is useful for controling the SVM's finitization behavior. For example, if a procedure asserts that the size of a union returned by a recursive call does not exceed a given concrete threshold, the SVM will stop unwinding the recursion when that threshold is exceeded.

5. Case Studies

In this section, we evaluate the effectiveness of our prototype SVM on a collection of benchmarks from three different SDSLs: an imperative language for solver-aided development of OpenCL kernels; a declarative SDSL for web scraping; and a functional SDSL for specifying executable semantics of secure information-flow control (IFC) mechanisms. We show that our type-driven state merging effectively controls state explosion on a range of applications, and that the SVM produces good encodings for these applications, as measured by constraint solving time. The benchmarks are described first, followed by presentation and discussion of results.

5.1 Benchmarks

SYNTHCL SYNTHCL is a new imperative SDSL for development, verification and synthesis of OpenCL [21] code. OpenCL is a programming model that supports both task and data (SIMD) parallelism. SYNTHCL focuses on the latter. A typical OpenCL (and SYNTHCL) program consists of two parts: a *host* procedure, which is executed on the CPU, and a *kernel* procedure, many of instances of which are executed concurrently on the available parallel hardware (such as a GPU). The host procedure manages the launching of kernels that comprise the parallel computation.

The SYNTHCL language is designed to support stepwise refinement of a sequential reference implementation into a vectorized data-parallel implementation, from which a code generator can produce a pure OpenCL program. As such, it supports key OpenCL primitive data types (booleans, integers and floats); their corresponding vector and pointer data types; and common operators on these types (arithmetic, bitwise, and logical). It also provides a model of the OpenCL API, and an abstract model of the OpenCL runtime. The SYNTHCL runtime distinguishes host memory from (global) device memory, with implicit and conservative handling of memory barriers. In particular, the runtime emits assertions to ensure that no two kernel instances ever perform a conflicting memory access. These assertions are checked by the SYNTHCL verifier and enforced by the synthesizer, along with any explicit assertions added by programmers to their SYNTHCL code.

We used SYNTHCL to develop new fast vectorized versions of three standard, manually optimized OpenCL benchmarks [1]: Matrix Multiplication (MM), Sobel Filter (SF), and Fast Walsh Transform (FWT). These benchmarks are representative of low-level imperative programs that use the OpenCL framework. The MM benchmark computes the dot product of two matrices with dimensions $n \times p$ and $p \times m$, both represented as one-dimensional arrays of floats. SF performs edge detection on a $w \times h$ image, represented as a $w \times h \times 4$ array of integers (*i.e.,* an integer encodes one of the four color components of a pixel). FWT computes the Walsh transform [2] of an array of 2^k numbers.

Each of these standard benchmarks is distributed with a sequential reference implementation, which we took as a starting point for development. The development process involved two main phases, broken up into several refinement steps. In the first phase, the reference implementation was transformed into a data-parallel implementation operating on scalar data types (*e.g.,* `int`). In the second phase, the scalar parallel implementation was refined to operate on vector data types (*e.g.,* `int4`), taking advantage of SIMD operations. We used SYNTHCL **verify** and **synthesize** queries to ensure that every refinement step resulted in a correct program. This refinement process resulted in a total of 12 SYNTHCL implementations of the three programs. We use these implementations and their queries as benchmarks in our evaluation.

Table 1 describes the benchmarks in more detail. Each row of the table shows a set of benchmark identifiers, together with the query finitization bounds for those benchmarks. A benchmark identifier includes the program name, the refinement step, and a query descriptor (v for verification and s for synthesis). Query bounds are expressed as bounds on the length of the input arrays (of symbolic numbers) accepted by the benchmark programs. We use 32-bit numbers for verification queries, and 8-bit numbers for synthesis queries. Input bounds are the only source of finitization in our queries; all loops in the benchmark programs are self-finitizing (unwound precisely) with respect to the input bounds.

For example, the first row of Table 1 describes the bounds for two verification queries, $\mathrm{MM1}_v$ and $\mathrm{MM2}_v$. These queries verify the correctness of the first and second refinement of MM with respect to all possible pairs of input matrices comprised of 32-bit numbers, with their dimensions drawn from the set $\{4, 8, 12, 16\}$. There are 2^{409600} such pairs of matrices, where $409600 = 32 * \sum_{n,p,m \in \{4,8,12,16\}} n * p + p * m$.

WEBSYNTH WEBSYNTH [39] is a small declarative SDSL for example-based web scraping. Given an HTML tree and a few representative examples of the data to be scraped, WEBSYNTH

benchmark	bounds on input length
MM1$_v$, MM2$_v$	$\{\langle n*p, p*m\rangle \mid n,p,m \in \{4,8,12,16\}\}$
MM2$_s$	$\{\langle n*p, p*m\rangle \mid n,p,m \in \{8\}\}$
SF1$_v$, SF2$_v$, SF3$_v$, SF4$_v$, SF5$_v$	$\{w*h*4 \mid w,h \in \{1,2,\ldots,9\}\}$
SF6$_v$, SF7$_v$	$\{w*h*4 \mid w,h \in \{3,4,\ldots,9\}\}$
SF3$_s$	$\{w*h*4 \mid w,h \in \{1,2,3,4\}\}$
SF7$_s$	$\{w*h*4 \mid w,h \in \{4\}\}$
FWT1$_v$, FWT2$_v$	$\{2^k \mid k \in \{0,1,2,3,4,5,6\}\}$
FWT1$_s$, FWT2$_s$	$\{2^k \mid k \in \{3\}\}$

Table 1. Query bounds for SYNTHCL benchmarks.

benchmark	# of tree nodes	tree depth	# of XPath tokens
iTunes$_s$	1104	10	150
IMDb$_s$	2152	20	359
AlAnon$_s$	2002	22	161

Table 2. Query bounds for WEBSYNTH benchmarks.

benchmark	# of instructions	max sequence length
B1$_v$, B2$_v$	7	3
B3$_v$	7	5
B4$_v$	7	7
J1$_v$	8	6
J2$_v$	8	4
CR1$_v$	9	7
CR2$_v$, CR3$_v$	9	8
CR4$_v$	9	10

Table 3. Query bounds for IFCL benchmarks.

synthesizes an XPath expression that retrieves the desired data. The synthesizer works by checking that every example datum is reached when a recursive XPath interpreter traverses the input tree according to a symbolic XPath, represented as a list of symbolic constants. The depth of the input tree provides a natural upper bound on the length of the XPath to be synthesized and on the unwinding of recursive traversals of the input tree. In particular, the WEBSYNTH interpreter is self-finitizing with respect to the structure of the input tree. The set of tokens from which XPath elements are drawn is also computed from the structure of the tree.

We include three WEBSYNTH programs as evaluation benchmarks; these synthesize XPaths for scraping data from three real websites [39], using four input examples for each. Table 2 shows the name of each website; the number of nodes in its HTML tree; the depth of the tree; and the number of tokens from which XPath elements are drawn. Conceptually, a synthesis query searches a space of t^d XPath candidates, where d is the depth of the tree and t is the number of XPath tokens. For example, the iTunes$_s$ search space consists of $150^{10} \approx 2^{72}$ possible XPath candidates.

IFCL IFCL is a new functional SDSL for specifying and verifying executable semantics of abstract stack-and-pointer machines that track dynamic information flow, in order to enforce security properties. An IFCL program is an implementation of a set of instructions that define such a stack machine. Inituitively, an IFCL machine is "secure" if it does not allow secret inputs to influence publicly observable outputs. The verification problem therefore involves finding two indistinguishable sequences of machine instructions, which produce distinguishable final states when executed.

We used IFCL to implement the ten (versions of the) machine semantics described in [19], and to confirm that they are buggy with respect to the desired security property, also known as end-to-end non-interference [19]. We use these semantics implementations and the corresponding verification queries as our benchmarks.

Table 3 shows the verification bounds for all benchmarks, in terms of the size of each benchmark's instruction set, as well as the upper bound on the length of instruction sequences explored by the verifier. Given an upper bound of k, and a machine semantics with n instructions, the verifier searches for two sequences of up to k instructions drawn from that semantics, which violate the security property in at most k steps. We use a 5-bit representation of numbers for all queries, and the machine memory is limited to 2 cells (as in [19]). For each benchmark, we pick the upper bound k to be the length of the known counterexample [19] for that benchmark, resulting in a space of $2*n^k$ candidate instruction sequences.[5] The bound k, and the limit on the machine's memory, are the only source of finitization in IFCL programs. The execution of the symbolic instruction sequences is self-finitizing with respect to k.

5.2 Results

Table 4 shows the results obtained by executing all three sets of benchmarks with the ROSETTE SVM. The evaluation was performed using the Z3 [13] solver (version 4.3.1) on a 2.13 GHz Intel Core 2 Duo processor with 4 GB of memory. For each benchmark, we show the number of control-flow joins encountered during SVM evaluation; the total number of symbolic unions that were created; the sum of their cardinalities; the maximum cardinality; the SVM execution time (in seconds); and the Z3 solving time (in seconds). Execution times are averaged over three runs and rounded to the nearest second.

Figure 10 presents an alternative set of results obtained by executing one benchmark program, B1$_v$, with different query bounds. We varied the upper bound on the instruction sequence length from 1 to 15, inclusive. The resulting sums of cardinalities are plotted against the number of control-flow joins encountered during each execution. The data is fitted to the slow-growing quadratic curve $y = 0.00003122x^2 + 1.2253x - 494.2$, with $R^2 = 0.9993$.

5.3 Discussion

Since the number of paths is expected to be roughly exponential in the number of joins, the results in Table 4 and Figure 10 confirm the effectiveness of our state merging strategy. None of the benchmarks leads to an exponentially large representation of the symbolic state, despite the exponential number of possible paths in each. In particular, the sum of union cardinalities is polynomial in the size of each (finitized) benchmark, as measured by the number of control-flow joins.

Table 4 reveals a couple of different SVM execution patterns. Many of the benchmarks were evaluated without the use of symbolic unions even though they operate on complex data types (such as mutable arrays in the case of SYNTHCL and a recursive data type representing HTML trees in the case of WEBSYNTH). The reason is simple—the operations on these complex data types were all evaluated concretely, and the only values that needed merging were the primitives comprising their contents (for SYNTHCL) or the primitives computed by traversing their structure (for WEBSYNTH).

Unions were most heavily used in the encoding of SYNTHCL synthesis queries (*e.g.,* MM2$_s$) and IFCL verification queries (*e.g.,* CR4$_v$). In the case of IFCL, for example, the instruction at each program position is unknown, so executing one step of the machine requires the SVM to merge all "next" machine states (represented as immutable records) that could result from executing any of the n possible machine instructions. Executing a sequence of symbolic instructions also involves merging lists of different lengths, since we use lists to represent the machine's stack, which grows and shrinks during execution.

For all but two benchmarks, solving times are less than 20 seconds, demonstrating that the SVM produces good (easy-to-solve) encodings. Our IFCL verifier is not as fast as the specialized random-testing tool from [19]. But unlike this tool, IFCL explores the (same) bounded input space exhaustively, occasionally finding

[5] The solver's search space is larger than this, because it also has to discover the values of arguments for some of the instructions.

		joins	count	sum	max	SVM (sec)	Z3 (sec)
SynthCL	MM1$_v$	2,233,155	0	0	0	59	0
	MM2$_v$	1,586,947	0	0	0	64	1
	MM2$_s$	18,781	3,705	15,915	65	136	69
	SF1$_v$	576,636	0	0	0	28	0
	SF2$_v$	463,740	0	0	0	27	0
	SF3$_v$	465,868	0	0	0	28	0
	SF4$_v$	426,988	0	0	0	25	0
	SF5$_v$	385,980	0	0	0	23	0
	SF6$_v$	707,318	0	0	0	28	0
	SF7$_v$	553,948	0	0	0	23	0
	SF3$_s$	20,716	0	0	0	2	18
	SF7$_s$	4,258	48	3,120	65	9	6
	FWT1$_v$	21,644	0	0	0	1	0
	FWT2$_v$	16,914	0	0	0	0	0
	FWT1$_s$	3,419	279	894	9	3	3
	FWT2$_s$	945	50	142	9	0	0
WebSynth	iTunes$_s$	76,438	0	0	0	0	0
	IMDb$_s$	85,413	0	0	0	0	0
	AlAn$_s$	279,543	0	0	0	0	0
IFCL	B1$_v$	1,097	328	767	7	0	0
	B2$_v$	1,097	328	767	7	0	0
	B3$_v$	2,273	992	2,237	7	3	0
	B4$_v$	3,785	1,992	4,459	8	10	1
	J1$_v$	3,353	1,644	3,712	8	7	0
	J2$_v$	1,737	698	1,608	8	2	0
	CR1$_v$	6,099	3,488	7,770	9	27	5
	CR2$_v$	7,511	4,636	10,366	9	49	19
	CR3$_v$	7,669	4,636	10,366	9	48	8
	CR4$_v$	17,759	11,982	35,768	11	290	119

Table 4. SVM performance on all benchmarks with respect to the number of control flow join points, as measured by the total number of symbolic unions created during evaluation, the sum of their sizes, SVM evalution time, and Z3 solving time.

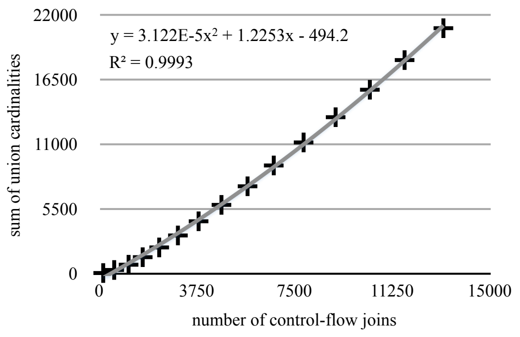


Figure 10. SVM performance on B1$_v$ verification queries, with respect to a range of verification bounds (1 to 15, inclusive). The graph plots the sum of sizes of symbolic unions against the number of control-flow joins encountered during evaluation of each query.

shorter counterexamples. We also implemented the IFCL language and its bounded verifier in just a few days, without having to develop custom search algorithms or heuristics.

6. Related Work

We discussed the related work on standard symbolic execution and bounded model checking techniques in Section 3, where we illustrated the need for a new approach to compile solver-aided host languages to constraints. In this section, we discuss other related techniques for symbolic compilation and state merging.

Symbolic Compilation of Low-Level Languages Intermediate Verification Languages (IVLs) [15, 17, 30] are low-level representations, designed to simplify development of verifiers for general-purpose languages, such as Spec#, Java or Ruby. They are equipped with basic programming and specification constructs, as well as symbolic compilers that generate efficient logic encodings of program verification queries. Developers of high-level verifiers benefit from these compilers by translating their source languages to IVLs, rather than lowering them directly to constraints. This eases some of the engineering burden of building a verifier, but compilation to an IVL still requires significant effort and expertise, especially when there is a fundamental mismatch between the verifier's source language and the target IVL (*e.g.,* a dynamically typed source language and a statically typed IVL) [36]. IVL compilers are more limited than the SVM in the kinds of queries they can encode, but their encodings support full functional verification in the presence of unbounded loops and domains [17, 30].

Symbolic Compilation of High-Level Languages Dafny [29] and Spec# [5] are stand-alone, imperative, object-oriented languages with solver-aided verification facilities. Both languages provide a rich set of constructs for specifying contracts and invariants. These specifications are verified at compile-time by translation to the Boogie [30] IVL, whose symbolic compiler generates verification queries (via a weakest precondition computation [4]) in the input language of the Z3 [13] solver. Dafny and Spec# both support full functional correctness verification. Neither, however, supports other solver-aided queries or meta-programming.

Sketch [37] is a stand-alone, Fortran-like imperative language with solver-aided synthesis facilities. Like the SVM, the Sketch symbolic compiler is complete only for finite programs. But the Sketch language is not designed for hosting SDSLs, and its compiler is specialized for synthesis queries.

Leon [6] and Kaplan [25] are high-level languages embedded in Scala. Leon supports verification and deductive synthesis queries for programs in PureScala, a purely functional, Turing-complete subset of Scala. The Leon compiler uses Z3 to discharge queries involving unbounded domains and recursion, which cannot be handled by our approach. Kaplan reuses Leon's symbolic compiler to implement a restricted form of angelic execution for PureScala. Unlike the SVM, however, the Leon compiler does not support queries about imperative programs with arbitrary mutation.

Rubicon [31] is an SDSL embedded in Ruby. It extends Ruby with first-class symbolic values, which are used to specify bounded verification queries about programs written in the Rails web programming framework. The Rubicon symbolic compiler is implemented similarly to ours, by selectively lifting Ruby's own interpreter to operate on symbolic values. The two approaches differ in their symbolic merging strategies: Rubicon performs logical merging of heap-allocated values, while we perform type-driven merging of these values. Rubicon does not support other solver-aided queries.

State Merging Strategies Some symbolic execution engines (see, *e.g.,* [28]) employ heuristic state merging strategies as a means of limiting path explosion. These engines work by selecting (either statically or dynamically) a subset of states for logical merging, and exploring the remaining states path-by-path. Unlike type-driven merging, they are optimized for verification of low-level code, preventing path explosion for some but not all programs.

Abstract interpretation [12] statically computes a set of abstract facts about a program's behavior, in order to verify that the program satisfies a given property (*e.g.,* the sum of two variables is zero). It works by merging abstract states at join points, which may result in a loss of precision and cause the verifier to report false positives. Improving the precision of abstract interpretation typically entails expanding [38] or modifying [34] the abstract domain so that the property of interest is tracked (more) directly. Our work is similar in that a property of the propagated value (*e.g.,* the length of a list) affects how the symbolic values are merged. We differ in that the SVM never loses information at merge points, because it tracks concrete semantics and never performs abstraction. How we perform merges thus affects only the size of the resulting symbolic representation. To reduce the representation size, the SVM algebraically modifies merged expressions, distributing merge operators across lists and other composite (immutable) values.

7. Conclusion

Solver-aided domain-specific languages (SDSLs) rely on satisfiability solvers to automate program verification, debugging, synthesis and non-deterministic (angelic) execution. Providing this automation requires translating programs to constraints, which is time-consuming and hard even for domain-specific languages.

Our prior work [39] introduced a new way to build SDSLs that avoids translation to constraints: SDSL designers implement their languages simply by writing an interpreter or a library in a solver-aided host language. In this paper, we showed how to implement such a host language, by building a lightweight symbolic virtual machine (SVM) that can efficiently translate SDSL implementations and programs to constraints.

The SVM translates to constraints only a small lifted subset of the host language, although SDSL designers are free to use the entire language. Our approach relies on a novel state merging strategy to eliminate path explosion during symbolic execution, and it relies on concrete evaluation to strip away unlifed constructs. Unlike classic solver-aided tools, the SVM is also easily extensible with the help of symbolic reflection. This new mechanism allows SDSL designers to write a few lines of code and extend the set of lifted operations, without modifying the SVM. We showed that our approach effectively translates a variety of SDSLs and programming styles, generating easy-to-solve constraints in a simple logic.

Acknowledgments

We thank Jeffrey Foster, Shaz Qadeer, and the anonymous reviewers for their comments on this paper. The research presented here was supported in part by awards from the Department of Energy (DOE DE–SC0005136 and DOE FOA–0000619) and the National Science Foundation (NSF CCF–0916351, NSF CCF–1139138, and NSF CCF–1337415), as well as gifts from Intel, Nokia, and Samsung.

References

[1] AMD. Samples & demos. http://developer.amd.com/tools-and-sdks/heterogeneous-computing/amd-accelerated-parallel-processing-app-sdk/samples-demos/, 2013.

[2] J. Arndt. *Matters Computational: Ideas, Algorithms, Source Code*. Springer, 2011.

[3] D. Babic and A. J. Hu. Calysto: scalable and precise extended static checking. In *ICSE*, 2008.

[4] M. Barnett and K. R. M. Leino. Weakest-precondition of unstructured programs. In *PASTE*, 2005.

[5] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, 2004.

[6] R. W. Blanc, E. Kneuss, V. Kuncak, and P. Suter. On Verification by Translation to Recursive Functions. Technical Report 186233, EPFL, 2013.

[7] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, Feb. 2013.

[8] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[9] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[10] S. Chandra, E. Torlak, S. Barman, and R. Bodik. Angelic debugging. In *ICSE*, 2011.

[11] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *TACAS*, 2004.

[12] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[13] L. De Moura and N. Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

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

[15] G. Dennis, F. S.-H. Chang, and D. Jackson. Modular verification of code with SAT. In *ISSTA*, 2006.

[16] J. Dolby, M. Vaziri, and F. Tip. Finding bugs efficiently with a SAT solver. In *FSE*, 2007.

[17] J. C. Filliâtre and C. Marché. The Why/Krakatoa/Caduceus platform for deductive program verification. In *CAV*, 2007.

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

[19] C. Hritcu, J. Hughes, B. C. Pierce, A. Spector-Zabusky, D. Vytiniotis, A. Azevedo de Amorim, and L. Lampropoulos. Testing noninterference, quickly. In *ICFP*, 2013.

[20] M. Jose and R. Majumdar. Bug-Assist: assisting fault localization in ANSI-C programs. In *CAV*, 2011.

[21] *The OpenCL Specification, Version 1.2*. Khronos OpenCL Working Group, November 2012.

[22] S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.

[23] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[24] E. Kohlbecker, D. P. Friedman, M. Felleisen, and B. Duba. Hygienic macro expansion. In *LFP*, 1986.

[25] A. S. Köksal, V. Kuncak, and P. Suter. Constraints as control. In *POPL*, 2012.

[26] A. S. Köksal, Y. Pu, S. Srivastava, R. Bodík, J. Fisher, and N. Piterman. Synthesis of biological models from mutation experiments. In *POPL*, 2013.

[27] S. Krishnamurthi. Educational pearl: Automata via macros. *J. Funct. Program.*, 16(3):253–267, 2006.

[28] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *PLDI*, 2012.

[29] K. R. M. Leino. Dafny: an automatic program verifier for functional correctness. In *LPAR*, 2010.

[30] K. R. M. Leino and P. Rümmer. A polymorphic intermediate verification language: Design and logical encoding. In *TACAS*, 2010.

[31] J. P. Near and D. Jackson. Rubicon: bounded verification of web applications. In *FSE*, 2012.

[32] Racket. The Racket programming language. racket-lang.org.

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

[34] M. Sagiv, T. Reps, and S. Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *TAPSOFT*, 1996.

[35] P. Saxena, D. Akhawe, S. Hanna, F. Mao, S. McCamant, and D. Song. A symbolic execution framework for JavaScript. In *SP*, 2010.

[36] L. Segal and P. Chalin. A comparison of intermediate verification languages: Boogie and Sireum/Pilar. In *VSTTE*, 2012.

[37] A. Solar-Lezama, L. Tancau, R. Bodik, V. Saraswat, and S. A. Seshia. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.

[38] B. Steffen. Property-oriented expansion. In *SAS*, 1996.

[39] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

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

[41] Y. Xie and A. Aiken. Saturn: A scalable framework for error detection using boolean satisfiability. *ACM Trans. Program. Lang. Syst.*, 2007.