Functional SMT solving with Z3 and racket

Conference Paper · May 2013

DOI: 10.1109/FormaliSE.2013.6612272

2 authors, including:

Amey Karkare
Indian Institute of Technology Kanpur
44 PUBLICATIONS **372** CITATIONS

SEE PROFILE

Functional SMT Solving with Z3 and Racket

Siddharth Agarwal*†
sid0@fb.com
*Facebook Inc,
Menlo Park, CA, USA

Amey Karkare†
karkare@cse.iitk.ac.in
†Department of Computer Science & Engineering
Indian Institute of Technology Kanpur, India

***Abstract*—Satisfiability Modulo Theories (SMT) solvers are powerful tools that can quickly solve complex constraints involving Booleans, integers, first-order logic predicates, lists, and other data types. They have a vast number of potential applications, from constraint solving to program analysis and verification. However, they are so complex to use that their power is inaccessible to all but experts in the field. We present an attempt to make using SMT solvers simpler by integrating the Z3 solver into a host language, Racket. The system defines a programmer's interface in Racket that makes it easy to harness the power of Z3 to discover solutions to logical constraints. The interface, although in Racket, retains the structure and brevity of the SMT-LIB format. This system is expected to be useful for a wide variety of applications, from simple constraint solving to writing tools for debugging, verification, and automatic test generation for functional programs.**

I. Introduction

The Boolean satisfiability or SAT problem asks: *Given a Boolean formula with a set of variables in it, is there a way to assign each variable a value such that the formula becomes true?* The SAT problem is one of the cornerstones of computer science, with enormous theoretical and practical implications. Indeed, it was the very first problem to be proved *NP-complete* [1]. Yet, interest in efficiently solving so-called "natural" or "real-world" instances of SAT has remained. This is at least partly because a large number of practical problems are also NP-complete and can be *reduced* to SAT [2].

Typically, program analysis tools that used SAT solvers would have to find a way to translate variables found in programs to Boolean ones. For example, a 32-bit integer could be encoded as a set of 32 Boolean variables[1]. It was soon realized that pushing this step into the SAT solver would help. Since users would still be asking whether formulas were satisfiable, except with variables from more complex domains or *theories*, this approach was dubbed Satisfiability Modulo Theories (SMT). Several popular SMT solvers have been developed : Z3 [4], Yices [5] and CVC4 [6] to name a few.

These solvers let programmers specify constraints over Booleans, integers, pure functions and other types, and either come up with assignments that satisfy these constraints, or, if possible, a proof that the constraints aren't satisfiable. Over the last few years, SMT solvers using DPLL(T) [7] and other frameworks have come into their own and can solve a wide variety of problems using efficient heuristics. Problems they can attack range from simple puzzles like Sudoku and n-queens, to planning and scheduling, program analysis [8], whitebox fuzz testing [9] and bounded model checking [10]. Yet SMT solvers are only used by a small number of experts. It isn't hard to see why: the standard way for programs to interact with SMT solvers like Z3 [4], Yices [5] and CVC3 [11] is via powerful but relatively arcane C APIs that require the users to know the particular solver's internals. For example, here is a C program that asks Z3 whether the simple proposition $p \wedge \neg p$ is satisfiable.

```
Z3_config cfg = Z3_mk_config();
Z3_context ctx = Z3_mk_context(cfg);
Z3_del_config(cfg);
Z3_sort bool_srt = Z3_mk_bool_sort(ctx);

Z3_symbol sym_p = Z3_mk_int_symbol(ctx, 0);

Z3_ast p = Z3_mk_const(ctx, sym_p, bool_srt)
    ;
Z3_ast not_p = Z3_mk_not(ctx, p);

Z3_ast args[2] = {p, not_p};
Z3_ast conjecture = Z3_mk_and(ctx, 2, args);
Z3_assert_cnstr(ctx, conjecture);

Z3_lbool sat = Z3_check(ctx);
Z3_del_context(ctx);
return sat;
```

Simultaneously, most SMT solvers also feature interaction via the standard input language SMT-LIB [12]. SMT-LIB is *significantly* easier to use in isolation. The same program in SMT-LIB would look something like

```
; Declare a Boolean variable
(declare-fun p () Bool)
; Try to find a contradiction
(assert (and p (not p)))
(check-sat)
; Prints "unsat", meaning "unsatisfiable"
```

However, the SMT-LIB interfaces are generally hard to use directly from programs and often not as full-featured or extensible as corresponding C APIs[2]. Importantly, it is difficult to write programs that *interact* with the solver in some way, for example by adding assertions based on generated models. This makes it difficult to build new abstractions to enhance functionality.

[1]It is also possible to represent integers as Boolean variables via predicate abstraction [3], which is more efficient but potentially loses information.

[2]Z3, for instance, supports plugging in external theories via the C API, but not via the textual SMT-LIB interface.

978-1-4673-6292-4/13 © 2013 IEEE

To overcome these difficulties, we decided to implement an SMT-LIB-like interface to Z3 in a way that allowed for the same power as the C interface while appearing naturally integrated into a host language. Since SMT-LIB is *s-expression*-based, for the host language a Lisp dialect was a natural choice. We chose Racket [13] for our implementation, `z3.rkt`, because of its extensive facilities for implementing new languages [14], not just for the interface to the solver, but also for the resulting tools that the solver would make possible.

Using this system, the program above becomes almost as brief as the SMT-LIB version.

```
(smt:with-context
 (smt:new-context)
 (smt:declare-fun p () Bool)
 (smt:assert (and/s p (not/s p)))
 (smt:check-sat))
```

It is important to note that we are neither increasing the power of the Z3 SMT solver, nor adding any new features to it. We are providing a new interface in Racket so that the solver can be used from within the Racket language with much ease. This itself is an interesting, challenging and useful task as the rest of the paper demonstrates.

II. Interactive SMT Solving

To demonstrate the value in integrating a language with an SMT solver, we turn our attention to a pair of classic logical puzzles.

A. Sudoku

We first turn our attention to a problem that demonstrates how the interaction of a language with an SMT solver is useful. A Sudoku puzzle asks the player to complete a partially pre-filled 9×9 grid with the numbers 1 through 9 such that no row, column, or 3×3 box has two instances of a number. This is a classic constraint satisfaction problem, and any constraint solver can handle it with ease.

A Racket program using `z3.rkt` to solve Sudoku would look like the following:

```
(define (solve-sudoku grid)
  (smt:with-context
   (smt:new-context)
   ; Declare a scalar datatype (finite domain
   ; type) with 9 entries
   (smt:declare-datatypes ()
     ((Sudoku S1 S2 S3 S4 S5 S6 S7 S8 S9)))
   ; Represent the grid as an array from
   ; integers to this type
   (smt:declare-fun sudoku-grid ()
    (Array Int Sudoku))
   ; Assert the standard grid rules
   ;  (row, column, box)
   (add-sudoku-grid-rules)
   ; Add pre-filled entries
   (add-grid grid)
   (define sat (smt:check-sat))
   ; 'sat means we found a solution,
   ; 'unsat means we didn't
   (if (eq? sat 'sat)
       ; Retrieve the values from the model
       (for/list ([x (in-range 0 81)])
         (smt:eval (select/s sudoku-grid x)))
       #f)))
```

Here we omit a couple of function definitions: `add-sudoku-grid-rules` asserts the standard Sudoku grid rules, and `add-grid` reads a partially filled grid in a particular format and creates assertions based on it. We note that the function `(select/s arr x)` retrieves the value at `x` from the array `arr`, and that this can be used to add constraints on the array (for instance, `(smt:assert (=/s (select/s arr x) y))`). We also note that if a set of constraints is satisfiable, Z3 can generate a *model* showing this; values can be extracted out of this model using the `smt:eval` command.

However, simply finding a solution isn't enough for a good Sudoku solver: it must also verify that there aren't any other solutions. The usual way to do that for a constraint solver is by retrieving a generated model, adding assertions such that this model cannot be generated again, and then asking the solver whether the system of assertions is still satisfiable. If it is, a second solution exists and the puzzle is considered invalid.

In such situations, the interactivity offered by `z3.rkt` becomes useful: it lets the programmer add dynamically discovered constraints on the fly. The last part of the program then becomes

```
..
if (eq? sat 'sat)
   ; Make sure no other solution exists
   (let ([result-grid
     (for/list ([x (in-range 0 81)])
       (smt:eval (select/s sudoku-grid x)))])
     ; Assert that we want a new solution
     ; by asserting (not <current solution>)
     (smt:assert
      (not/s
        (apply and/s
          (for/list
            ([(x i) (in-indexed result-grid)])
            (=/s (select/s sudoku-grid i) x)))
              ))
     (if (eq? (smt:check-sat) 'sat)
         #f ; Multiple solutions
         result-grid))
   #f)))
```

This part can even be abstracted out into a function that returns a lazily-generated sequence of satisfying assignments for any given set of constraints.

B. Number Mind

The deductive game Bulls and Cows, commercialized as Master Mind [15], is popular all around the world. The rules may vary slightly, but their essence stays the same: Two players play the game. One player (we'll call her Alice) thinks of a 4-digit number, and the other (Bob) tries to find it. Bob guesses a number, and Alice tells him how many digits he

has correct and in the correct position (*bulls*) and how many he has correct but in the wrong position (*cows*). Through repeated guessing Bob tries to arrive at the answer.

The game is deceptively simple: while even the standard 4-digit variant is challenging for humans, the general problem for n digits is NP-complete [16]. As such, it becomes an interesting problem for constraint solvers.

For simplicity, we tackle a variant of the game: Number Mind [17], where Alice only tells Bob how many digits are correct and in the correct place (bulls). The user is Alice and the computer Bob, which means that the game is *interactive*. An API to solve Number Mind would have

(a) a way to tell the computer how many digits the number has
(b) a way for the computer to guess a number
(c) a way for the user to tell the computer how many digits it got correct in the last guess.

The constraint solver would have an important role in not just (a) and (c) but also (b), since we would like the computer to make "reasonable" guesses and not just wild ones. We do this by never guessing a number that would be impossible because of the answers already given.

Our system makes all three tasks simple. The following code shows the three functions, each corresponding to one of the tasks above.

```
; (a) Create variables for each digit
(define (make-variables num-digits)
  (define vars (smt:make-fun/list num-digits
      () Int))
  ; Every variable is between 0 and 9
  (for ([var vars]) (smt:assert (and/s (>=/s
     var 0) (<=/s var 9))))
  vars)

; (b) Guess a number. Returns the guess as a
; list of digits, or #f meaning no number can
; satisfy all the constraints.
(define (get-new-guess vars)
  (define sat (smt:check-sat))
  (if (eq? sat 'sat)
      ; Get a guess from the SMT solver
      (map smt:eval vars)
      #f))

; (c) How many digits the computer got correct
; If a digit is correct then we assign it the
; value 1, otherwise 0. We sum up the values
; and assert that that's equal to the number
; of correct digits.

(define (add-guess vars guess correct-digits)
  (define correct-lhs
    (apply +/s
           (for/list ([x guess]
                      [var vars])
             (ite/s (=/s var x)
                    1      ; Correct guess
                    0))))  ; Wrong guess
  (smt:assert (=/s correct-lhs correct-digits)
      ))
```

As a demonstration of `z3.rkt`, we have written a small web application around the code [18].

III. Design and Implementation

`z3.rkt` is currently implemented as a few hundred lines of Racket code that interface with the Z3 engine via the provided library. Since the system is still a work in progress, some of these details might change in the future.

A. The Z3 Wrapper

We use Racket's foreign interface [19] to map the Z3 library's C functions into Racket. The high-level interface communicates with Z3 by calling the Racket functions defined by the wrapper. While it is possible to use the Z3 wrapper directly, we highly recommend using the high-level interface instead.

B. Built-in Functions

Z3 comes with a number of built-in functions that operate on Booleans, numbers, and more complex values. We expose these functions directly but add a `/s` suffix to their usual names in the SMT-LIB standard, because most SMT-LIB names are already defined as functions by Racket and we want to avoid colliding with them.

C. The Core Commands

This is a small set of Racket macros and functions layered on top of the Z3 wrapper. As noted in Section I, the aim here is to hide the complexities of the C wrapper and stay as close to SMT-LIB version 2 commands [12] as possible. We prefix commands with `smt:` to avoid collisions with Racket functions.

D. Deriving Abstractions

Since the full power of Racket is available to us, we can define abstractions that allow users to simplify their code. For example, SMT-LIB allows users to define macros via the `define-fun` command.

```
(define-fun max ((a Int) (b Int)) Int
  (ite (> a b) a b)) ; ite is if-then-else
...
(assert (= (max 4 7) 7))
```

However, Z3's C API exposes no such command. Our first attempt to implement this facility for `z3.rkt` was to define a Racket function to do the same thing:

```
(define (smt-max a b)
  (ite/s (>/s a b) a b))
...
(smt:assert (=/s (smt-max 4 7) 7))
```

This works for smaller macros like `max`, but in our experience this sort of naïve substitution can result in final expressions for deeply nested functions becoming too large for Z3 to handle. Consider a function `(f a b)` that uses `(smt-max a b)` m times. This expression will be repeated

m times in the expression for `f`. Now consider a function `g` that uses the value of `(f 0 1)` n times. The expression for `g` will contain that of `(f 0 1)` n times, and consequently that of `(smt-max 0 1)` mn times. As the nesting increases, we see an exponential blowup in the size of the final expression[3].

We note, however, that any macro can also be written as a universally quantified formula. For example, `max` can be rewritten in the following way.

```
(declare-fun max (Int Int) Int)
(assert (forall ((a Int) (b Int))
                (= (max a b)
                   (ite (> a b) a b))))
```

Using this technique, we finally solved the problem of defining macros in our interface by providing a Racket macro, `smt:define-fun`, that has the same syntax as the SMT-LIB command and that outputs the equivalent universally quantified formula. Our solution is efficient owing to the fact that Z3 has a *macro finder* component that identifies and eliminates universal quantifiers that are macros in disguise.

The definition of `smt:define-fun` is as follows:

```
(define-syntax smt:define-fun
  (syntax-rules ()
    [(_ id () type body) ; Plain identifier
     (begin ; don't need a forall
       (smt:declare-fun id () type)
       (smt:assert (=/s id body)))]
    [(_ id ((argname argtype) ...) type body)
     (begin
       (smt:declare-fun id (argtype ...) type)
       (smt:assert
         (forall/s ((argname argtype) ...)
                   (=/s (id argname ...) body)
                        )))]))
```

We use Scheme's `syntax-rules` macro system [20] to its fullest extent. `syntax-rules` accepts pairs of input and output patterns and goes with the output pattern for the first input that can be matched, somewhat like the `cond` construct found in many Lisps. We handle two separate cases: (a) we're defining a plain identifier, in which case we have no need for the `forall`, and (b) we're defining a macro as above, in which case we do. The `...` as part of the macro definition is a special form recognized by `syntax-rules`: wherever it sees them in the output pattern, it substitutes for them a list of whatever was present in the input pattern. For example,

```
(smt:define-fun foo ((x Int) (y Bool)) Int
                    (+/s x (ite/s y 20 0)))
```

expands to

```
(smt:declare-fun foo (Int Bool) Int)
(smt:assert (forall/s ((x Int) (y Bool))
                      (=/s (foo x y) (+/s x (
                         ite/s y 20 0)))))
```

[3]We could merge common parts of expressions to reduce the number of AST nodes generated. In our experiments, this proved to be quite effective, yet still significantly slower than the solution we finally adopted.

E. Reusing Racket Abstractions

It is important that any new interface work well with existing abstractions. For example, consider users who want to write web applications in Racket using Z3 on the server. Racket's web server libraries [21] let them use delimited continuations whenever they need to pause computation on the server and wait for a client response, so our interface ought to be continuation-safe.

The `smt:with-context` macro uses another of Racket's abstractions, dynamic binding via *parameters*, to ensure that the Z3 context remains valid for the macro's *dynamic extent* [22]. In particular, for the case of a web server, the context is garbage collected only once either the computation is finished, or it times out waiting for a client response. By surrounding just the first part of a computation with `smt:with-context`, the user can insert arbitrary calls to Z3 at any point during the computation without losing context. The details of state preservation are hidden from the user, and the resulting code tends to be quite elegant.

F. Porting Existing SMT-LIB Code

One of our explicit goals is to enable existing SMT-LIB version 2 code to be ported with a small number of systematic changes. Table I lists the minimal set of changes that needs to be made to port existing SMT-LIB code to `z3.rkt`. We expect many SMT-LIB programs to become shorter as authors use Racket features wherever appropriate.

TABLE I. DIFFERENCES BETWEEN SMT-LIB AND `z3.rkt`

SMT-LIB code	`z3.rkt` code
Options: `(set-option :foo true)`	Keyword arguments: `(smt:new-context #:foo #t)`
Logics: `(set-logic QF_UF)`	The `#:logic` keyword: `(smt:new-context #:logic "QF_UF")`
Commands: `declare-fun`, `assert`, ...	Prefixed with `smt:`
Functions: `and`, `or`, `+`, `distinct` ...	Suffixed with `/s`
Boolean literals: `true` and `false`	`#t` and `#f`

IV. OUR EXPERIENCE

We see two broad classes of applications for interfaces like `z3.rkt`: enabling formal methods researchers to be more productive in their research, and making SMT solving more accessible to programmers in general. We have already looked at the second in Section II. In this section we explore a few ideas we've had and lessons we've learned while looking at the first. These ideas, though elementary, form a starting point for potential research in the area of formal verification.

A. Quantified Formulas are Hard

Z3 and other SMT solvers support lists and other recursive types. Z3 provides only basic support for lists: `insert` (cons), `head`, and `tail`. Further, Z3's macros are substitutions and do not support recursion. This makes it challenging to define functions that operate over the entirety of an arbitrary-length list.

Our first thought was to use a universal quantifier, as in Section III-D. Here is an example of a function that calculates the length of an integer list:

```
(declare-fun len ((List Int)) Int)
(assert (forall ((xs (List Int)))
  (ite (= xs nil)
       (= (len xs) 0)
       (= (len xs) (+ 1 (len (tail xs)))))))
```

There is a drawback with this approach: solving quantified assertions in Z3 requires model-based quantifier instantiation (MBQI) [23]. MBQI, while powerful, can also be very slow. In our experience, it is very easy to write a quantified formula that Z3's MBQI engine fails to solve in reasonable time[4].

So avoiding quantified formulas altogether seems like a good idea, but how do we do that? The easiest way is to unroll and bound the recursion to a desired depth [26]. One way to do this is to define macros `len-0`, `len-1`, `len-2`, ...`len-N`, where each `len-k` returns the length of the list if it is less than or equal to `k`, and `k` otherwise.

```
(define-fun len-0 ((xs (List Int)))
  0)

(define-fun len-1 ((xs (List Int)))
  (ite (= xs nil)
       0
       (+ 1 (len-0 (tail xs)))))

(define-fun len-2 ((xs (List Int)))
  (ite (= xs nil)
       0
       (+ 1 (len-1 (tail xs)))))
...
```

Our system makes defining a series of macros like this very easy.

```
(define (make-length n)
 (smt:define-fun len ((xs (List Int))) Int
   (if (zero? n)
       0
       (ite/s (=/s xs nil/s)
              0
              (let ([sublen (make-length (sub1
                   n))])
                (+/s 1 (sublen (tail/s xs)))))
                   ))
 len)
```

`(make-length 5)` returns an SMT function that works for lists of up to length 5, and returns 5 for anything bigger than that. Note how freely the Racket `if` and `let` forms are mixed into the SMT body. These constructs are evaluated at definition time, meaning that this definition reduces to the series of macros defined above up to length `n`. (At this point, the observant reader might have noticed that `smt:define-fun` expands to a universal quantifier as well. However, Z3 doesn't need MBQI to solve universally quantified assertions equivalent to `define-fun` macros. The initial example given for `len` is not one such assertion, since it refers to itself and `define-fun` macros aren't permitted to.)

It is easy to define other bounded recursive functions along the same lines that reverse lists, concatenate them, filter them on a predicate and much more. Using these building blocks we can now verify properties of recursive functions.

B. Verifying Recursive Functions

For this section we will work with a simple but non-trivial example: *quicksort*. A simple functional implementation of quicksort might look like the following:

```
(define (qsort lst)
 (if (null? lst)
     null
     (let*
      ([pivot (car lst)]
       [rest (cdr lst)]
       [left (qsort (filter (lambda (x) (<= x
           pivot)) rest))]
       [right (qsort (filter (lambda (x) (> x
           pivot)) rest))])
       (append left (cons pivot right)))))
```

This definition is correct, but what if the programmer mistakenly types in < instead of <=, or perhaps uses >= instead of >? We note that (a) for a correct implementation, the length of the output will always be the same as that of the input, and that (b) in either buggy case, the length of the output will be different whenever a pivot is repeated in the rest of the list. So comparing the two lengths is a good property to verify.

Using the method discussed in Section IV-A, we can write `make-qsort` that generates bounded recursive versions of `qsort`. Then we can verify the length property for all input lists up to a certain length `n`.

```
(smt:with-context
 (smt:new-context)
 (define qsort (make-qsort n))
 ; adding 1 to the maximum length is
 ; enough to show inequality
 (define len (make-length (add1 n)))
 (smt:declare-fun xs () (List Int))
 ; set a bound on the length
 (smt:assert (<=/s (len xs) n))
 ; prove the length property by asserting
 ; its negation
 (smt:assert
   (not/s (=/s (len xs)
               (len (qsort xs)))))
 (smt:check-sat))
```

Proving a property is done by checking that its negation is unsatisfiable. A quicksort works *correctly*[5] for lists up to

[4]In general, it is hard to deal with quantified formulas containing even linear arithmetic, because there is no sound and complete decision procedure for them [24].

Z3 has another way to solve quantified assertions, called *E-matching* [25]. E-matching uses patterns based on ground terms to instantiate quantifiers. We have not yet explored this approach.

[5]Here correctness is in the context of the property we are considering, i.e. the length of the output.

length `n` iff the above code returns `'unsat`. For quicksorts that are buggy (`'sat`), we can find a counterexample using `(smt:eval xs)` and `(smt:eval (qsort xs))`. For `n` = 4 on a buggy quicksort with filters <= and >=, Z3 returned us the counterexample with input `'(-3 -2 -1 -2)`, which as expected contains a repeated element.

In this approach, there is nothing specific to quicksort: this can easily be generalized to other functions that operate on lists and other data structures. The properties to prove can be more complex, such as whether a given sorting algorithm is stable. The only limits are computational constraints and the user's imagination.

V. Related Work

A. SMT Integration

Integrating an SMT solver with a language enables programmers in that language to solve whatever logical constraints arise in a program, without needing to resort to hand-coding a backtracking algorithm or other cumbersome methods. The solutions thus obtained can be used in the rest of the program. Thus, it isn't surprising that several such projects exist, most of them available freely on the Internet. These projects differ mainly in the host language, the interface, and the constructs they support.

As most languages support some form of interaction with C functions, they can be said to be already integrated with Z3 (or other SMT solvers) through the C API. However, we do not consider this to be true integration because it doesn't simplify the job of the programmer and, as noted in Section I, it requires her to deal with the internals of the solver.

The integration of Z3 with Scala [27] is one of the most complete implementations available right now. It provides support for adding new theories and procedural abstractions, and also takes advantage of Scala's type system to deal with some type-related errors at compile time. The system has been used to solve several challenging problems, both by the group that developed it and by others. The main disadvantage of this system is that the syntax is quite different from SMT-LIB, and is sometimes almost as verbose as using the C bindings.

Z3Py [28] is a new Python interface bundled with Z3 4.0. It has its own domain-specific language that is different from SMT-LIB. However, it is much more pleasant to use than the C interface and supports virtually all of Z3's features.

SMT Based Verification (SBV) [29] is a Haskell package that can be used to prove properties about bit-precise Haskell programs. Given a constraint in a Haskell program, SBV generates SMT-LIB code that can be run against either Yices or Z3. SBV supports bit-vectors, integers, reals, and arrays, but not lists or other recursive datatypes.

Yices-Painless [30] integrates Haskell with Yices via its C API. This project does not support arrays, tuples, lists and user defined data types yet. Further, the development of the tool seems to have stalled (last change to the repository was in January 2011).

The Z3 documentation page lists bindings to other languages like OCaml. These bindings correspond almost one-to-one with the C API, and thus they suffer from the same disadvantages.

B. Logic Programming and Constraint Programming

Many of the problems SMT solvers can tackle can also be solved within the logic programming paradigm, where programs are written as first-order logic predicates. However, logic programming languages like Prolog typically have well-defined and transparent search strategies, preventing the sorts of automatic heuristics that allow SMT solvers to be fast. Instead, programmers need to manually bound the search space with goals and cuts in the appropriate places.

Racket supports logic programming via Racklog [31], which works in much the same way as Prolog.

Many languages, including most Prolog variants, have access to libraries that allow some form of constraint solving. Advanced toolkits include Gecode [32] for C++ and JaCoP [33] for Java and Scala. Typically, these are limited to problems traditionally associated with constraint programming: Booleans, finite domains, integers and perhaps real numbers. However, they also have built-in support for *optimization* problems, something that is lacking in SMT solvers but can be emulated with a binary search on the cost function.

A classic example deserves a mention here: SICP [34, Section 4.3] describes an `amb` macro for Scheme, which can choose for a variable one out of a set of values given *ambiguously*, so as to satisfy given constraints. `amb` is a simple, lightweight form of logic programming.

C. Bounded Verification

Our work is inspired by the Leon verifier [26], which goes further and alternately considers underapproximations and overapproximations. Where in Section IV-A we simply return a default value if we've reached the limit of our recursion, the Leon verifier alternately always satisfies or always rejects once it gets to that point. We chose to simplify our implementation to avoid being mired in mechanics, since that wasn't the main focus of this paper. In the future, we plan to extend our method with ideas from the Leon verifier.

VI. Conclusions and Future Work

In this paper, we have presented `z3.rkt`, which lets users interact with an SMT solver programmatically. We have demonstrated through examples the simplicity and usefulness of such an interaction. The power of `z3.rkt` comes from the facilities provided by Racket to build abstractions on top of the SMT-solving capabilities of Z3. From the user's perspective, the integration is seamless and fully transparent.

Our implementation is open source and freely available at

http://www.cse.iitk.ac.in/users/karkare/code/z3.rkt/

`z3.rkt` is still a work in progress, and we hope to soon achieve the following:

- Support more Z3 constructs, including bit-vectors and external theories
- Derive new abstractions guided by practical use cases
- Possibly integrate with other SMT solvers

In the long term, we hope the community will find this system useful and will contribute to the project to solve large practical problems.

ACKNOWLEDGEMENTS

We thank Leonardo de Moura at Microsoft Research for his help in understanding the Z3 C API. The Racket community, at `#racket` on Freenode IRC, helped us understand some of the Racket syntax model's intricacies. That helped us avoid dead ends both early on and towards the end. This work was done while the first author was at IIT Kanpur.

REFERENCES

[1] S. A. Cook, "The complexity of theorem-proving procedures," in *STOC*, 1971, pp. 151–158.

[2] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*, R. Miller and J. Thatcher, Eds., 1972, pp. 85–103.

[3] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajamani, "Automatic predicate abstraction of C programs," in *PLDI*, 2001, pp. 203–213.

[4] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *TACAS*, ser. LNCS, 2008, vol. 4963.

[5] B. Dutertre and L. de Moura, "The Yices SMT solver," SRI International, Tech. Rep., 2006.

[6] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli, "Cvc4," in *Proceedings of the 23rd international conference on Computer aided verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 171–177. [Online]. Available: http://dl.acm.org/citation.cfm?id=2032305.2032319

[7] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli, "DPLL(T): Fast Decision Procedures," in *CAV*, ser. LNCS, vol. 3114, 2004, pp. 175–188.

[8] S. Gulwani, S. Srivastava, and R. Venkatesan, "Program analysis as constraint solving," in *PLDI*, 2008, pp. 281–292.

[9] P. Godefroid, M. Y. Levin, and D. A. Molnar, "Automated Whitebox Fuzz Testing," in *NDSS*, 2008. [Online]. Available: http://www.truststc.org/pubs/499.html

[10] A. Armando, J. Mantovani, and L. Platania, "Bounded model checking of software using SMT solvers instead of SAT solvers," *STTT*, vol. 11, no. 1, pp. 69–83, 2009.

[11] C. Barrett and C. Tinelli, "CVC3," in *CAV*, ser. LNCS, W. Damm and H. Hermanns, Eds., vol. 4590, 2007, pp. 298–302.

[12] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, England)*, A. Gupta and D. Kroening, Eds., 2010.

[13] M. Flatt and PLT, "Reference: Racket," PLT Inc., Tech. Rep. PLT-TR-2010-1, 2010, http://racket-lang.org/tr1/.

[14] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *PLDI*, 2011, pp. 132–141.

[15] D. E. Knuth, "The Computer as a Master Mind," *Journal of Recreational Mathematics*, vol. 9, no. 1, pp. 1–6, 1976–77.

[16] J. Stuckman and G. qiang Zhang, "Mastermind is NP-Complete," *INFOCOMP Journal of Computer Science*, vol. 5, pp. 25–28, 2006.

[17] Colin Hughes, "Problem 185 - Project Euler," http://projecteuler.net/problem=185, May 2012 (last accessed).

[18] S. Agarwal, "Numbermind," http://numbermind.less-broken.com, Source available from https://github.com/sid0/numbermind, January 2013 (last accessed).

[19] E. Barzilay, "The Racket Foreign Interface," http://docs.racket-lang.org/foreign/, March 2012 (last accessed).

[20] R. Kelsey, W. Clinger, and J. Rees, Eds., *Fifth Revised Report on the Algorithmic Language Scheme*. ACM SIGPLAN Notices, 1998, vol. 33, no. 9.

[21] J. McCarthy, "Web Applications in Racket," http://docs.racket-lang.org/web-server/, July 2012 (last accessed).

[22] M. Flatt, G. Yu, R. B. Findler, and M. Felleisen, "Adding delimited and composable control to a production programming environment," in *Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, ser. ICFP '07. New York, NY, USA: ACM, 2007, pp. 165–176.

[23] Y. Ge and L. de Moura, "Complete Instantiation for Quantified Formulas in Satisfiability Modulo Theories," in *CAV*, 2009, pp. 306–320.

[24] J. Y. Halpern, "Presburger Arithmetic With Unary Predicates is Π^1_1 Complete," *Journal of Symbolic Logic*, vol. 56, pp. 56–2, 1991.

[25] L. de Moura and N. Bjørner, "Efficient E-Matching for SMT Solvers," in *CADE-21*, 2007, pp. 183–198.

[26] P. Suter, A. S. Köksal, and V. Kuncak, "Satisfiability Modulo Recursive Programs," in *SAS*, 2011, pp. 298–315.

[27] A. S. Köksal, V. Kuncak, and P. Suter, "Scala to the Power of Z3: Integrating SMT and Programming," in *ICAD*, 2011, pp. 400–406.

[28] Microsoft Research, "Z3Py - Python interface for the Z3 Theorem Prover," http://rise4fun.com/z3py/, May 2012 (last accessed).

[29] L. Erkok, "Symbolic bit vectors: Bit-precise verification and automatic C-code generation," http://hackage.haskell.org/package/sbv-0.9.24, March 2012 (last accessed).

[30] D. Stewart, "yices-painless: An embedded language for programming the Yices SMT solver," http://hackage.haskell.org/package/yices-painless-0.1.2, March 2012 (last accessed).

[31] D. Sitaram, "Racklog: Prolog-Style Logic Programming," http://docs.racket-lang.org/racklog/, May 2012 (last accessed).

[32] Gecode Team, "Gecode: Generic Constraint Development Environment," http://www.gecode.org, May 2012 (last accessed).

[33] JaCoP Team, "JaCoP - Java Constraint Programming solver," http://jacop.osolpro.com/, May 2012 (last accessed).

[34] H. Abelson, G. J. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, 1996.