KALASHNIKOV: Fully Automatic Loop Free Program Synthesis by Bounded Model Checking

Matt Lewis

Oxford University

Abstract. We present a method for synthesizing loop free programs using a combination of bounded model checking and explicit enumeration. The user provides a specification in plain C and is not required to interact further with the synthesis processs. In particular the user need not tell the synthesizer which program components to use.
We demonstrate the effectiveness of our method by synthesizing several tricky bitvector programs and a selection of floating point programs.

Keywords: Program synthesis, bitvectors, bounded model checking, CBMC, floating point.

1 Introduction

2 Related Work

3 Basic Synthesis Algorithm

3.1 The Abstract Algorithm

Our task is to find a program which satisfies some specifcation. We can formalize this notion as follows: fix an input set I and an output set O. Specifications σ are then relations and programs P are computable functions:

$$\sigma \subseteq I \times O$$
$$P : I \to O$$

The synthesis problem is then to determine the validity of the following formula, and to find a witness P if the following second-order formula is valid:

$$\exists P. \forall x \in I. \sigma(x, P(x))$$

Depending on the logic needed to express this formula the synthesis problem may be decidable, semi-decidable or undecidable. For many interesting logics, checking the validity of a second order formula, such as the synthesis formula, is undecidable. Even for logics in which the problem is decidable, the quantifier alternation means that we do not have efficient decision procedures to verify our formula, although efficient methods may

exist for solving problems with only one quantifier. In particular if we are working in the logic of propositional satisfiability, we have SAT solvers which are efficient-in-practice at solving queries with a single quantifier. The corresponding problem with one level of quantifier alternation (2-QBF) is complete for NP^{NP}, and current 2-QBF solvers are much less efficient than SAT solvers.

When the first-order fragment of the logic is efficiently decidable and P can be expressed as a ground term of the logic, we can use a refinement loop to check the validity of the synthesis formula as shown in Fig. 3.1 and Fig. 3.1. The algorithm is divided into two components: SYNTH and VERIF. We track a finite (small) set of inputs and SYNTH synthesizes a candidate program which is correct on just those inputs. VERIF then tries to verify the candidate program by searching for an input on which the candidate program does not satisfy the specification. If no such input can be found, the candidate program is correct. Otherwise, the new input is added to the set of inputs we must check in SYNTH. If the input set is finite, this procedure is guaranteed to terminate.

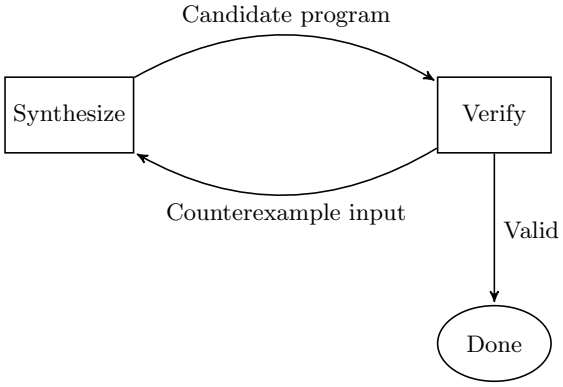

Fig. 1. Abstract synthesis refinement loop

3.2 The Concrete Algorithm for Bitvector Programs

One area in which program synthesis can shine is in producing very small, intricate programs that manipulate bitvectors. An example of such a program is shown in Fig. 3.2. This program takes a machine word as input and clears every bit except for the least significant bit that was set. Even though this program is extremely short, it is fairly difficult for a human to see what it does. It is even more difficult for a human to come up with such minimal code – a natural solution for this problem might be to use a loop which iterates over all of the bits in the word. The program is so concise because it takes advantage of the low level details

```
function synth(inputs) {
  (i_1,...,i_N) = inputs
  query := ∃P.σ(i_1,P(i_1)) ∧ ... ∧ σ(i_N,P(i_N))
  result := decide(query)

  if (result.satisfiable) {
    return result.model
  } else {
    return unsatisfiable
  }
}

function verif(P) {
  query := ∃x.¬σ(x,P(x))
  result := decide(query)

  if (result.satisfiable) {
    return result.model
  } else {
    return valid
  }
}

function refinement_loop() {
  inputs := ∅

  while (true) {
    candidate := synth(inputs)

    if (candidate = unsatisfiable) {
      return unsatisfiable
    }

    res := verif(candidate)

    if (res = valid) {
      return candidate
    } else {
      inputs := inputs ∪ res
    }
  }
```

Fig. 2. Abstract refinement algorithm

```
int isolate_lsb(int x) {
  return x & -x;
}
```

Example:

x	= 1 0 1 1 1 0 1 0
-x	= 0 1 0 0 0 1 1 0
x & -x	= 0 0 0 0 0 0 1 0

Fig. 3. A tricky bitvector program

of the machine, such as the fact that signed integers are stored in two's complement form.

To synthesize tricky bitvector programs like this, it is natural for us to work in the logic of quantifer free propositional formulae and to use a SAT solver as the decision procedure. However, we propose a slightly different tack, which is to use a decidable fragment of C as our underlying logic. The fragment we use has the following restrictions:

- All loops and recursive function calls must terminate after a constant number of iterations. This constant must be statically inferrable.
- Arrays must all be statically allocated with a constant size.

Other than this any C constructs can be used, which still leaves a very expressive language with the nice property that safety is decidable using a single query to a bounded model checker. We will refer to this fragment as C-. To instantiate the abstract synthesis algorithm in C- we must express I, O, σ and P in C-, then ensure that we can express the validity of the synthesis formula as a safety property of the resulting C- program. Our encoding is the following:

- I is the type `int[N]`.
- O is the type `int[M]`.
- σ is a function with signature `int check(int in[N], int out[M])`. This function is the only component supplied by the user.
- P is written in a simple RISC language $\mathcal{L}$. Programs in $\mathcal{L}$ have the type `prog_t`.
- We supply an interpreter for $\mathcal{L}$ which is written in C-. The signature of this interpreter is
  `void exec(prog_t p, int in[N], int out[M])`.

Integer arithmetic instructions:

```
add a b       sub a b       mul a b
div a b       neg a
```

Bitwise logical and shift instructions:

```
and  a b      or   a b      xor  a b
ashr a b      lshr a b      not  a
```

Unsigned and signed comparison instructions:

```
le  a b       lt  a b       sle  a b
slt a b
```

Fig. 4. The language $\mathcal{L}$

We must now express the SYNTH and VERIF formulae as safety properties of C- programs, which is shown in Fig. 3.2.

```
void synth() {
  prog_t p = nondet();
  int in[N], out[M];

  in = test1;
  exec(p, in, out);
  assume(check(in, out));

  ...

  in = testN;
  exec(p, in, out);
  assume(check(in, out));

  assert(false);
}
```

```
void verif(prog_t p) {
  int in[N] = nondet();
  int out[M];

  exec(p, in, out);
  assert(check(in, out));
}
```

Fig. 5. The SYNTH and VERIF formulae expressed as a C- program

4 C-

We use a fragment of C as our underlying logic. We will refer to this fragment as C-. The characteristic property of C- is that safety can be decided for C- programs by using a bounded model checker. A C- program is just a C program with the following syntactic restrictions:

- All loops in the program must have a constant bound.
- All recursion in the program must be limited to a constant depth.
- All arrays must be statically allocated (i.e. not using `malloc`), and be of constant size.

Additionally, C- programs may use:

- Nondeterministic values.
- Assumptions.
- Arbitrary width types.

So for example the programs in Fig. 4 are valid C- programs, but the program in Fig. 4 is not, since it uses a non-constant loop bound.
Since each loop is bounded by a constant, and each recursive function call is limited to a cosntant depth, a C- program necessarily terminates and in fact does so in $O(1)$ time. If we call the largest loop bound k, then a bounded model checker with an unrolling bound of k will be a complete decision procedure for the safety of the program. For a C- program of size l and with largest loop bound k, a bounded model checker will create

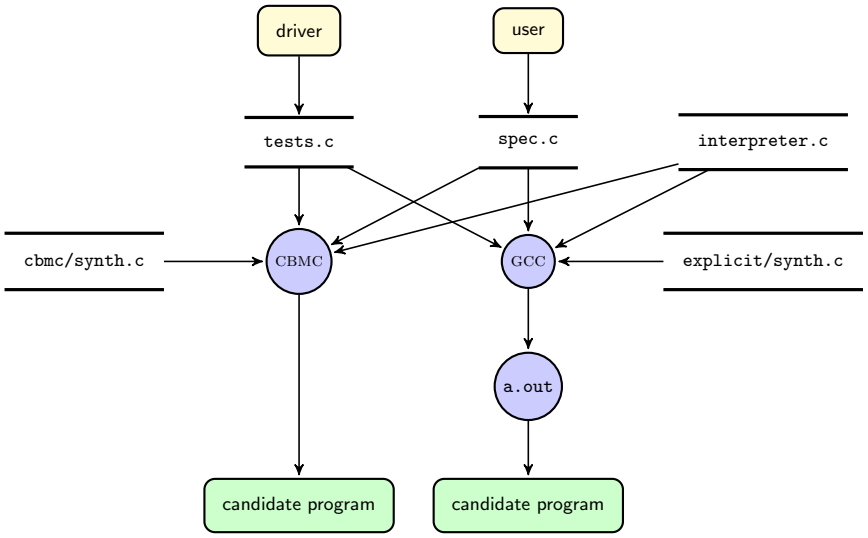

Fig. 6. Schematic diagram of SYNTH

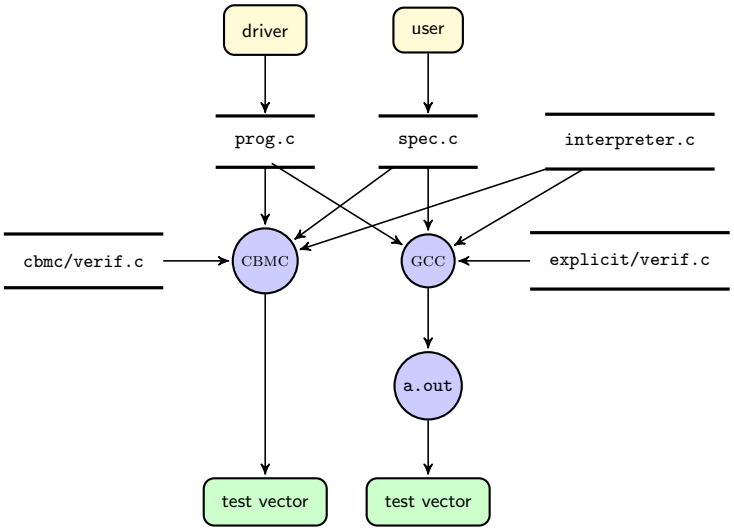

Fig. 7. Schematic diagram of VERIF

```
int count_bits(int x) {
  int i, ret;

  ret = 0;

  for (i = 0; i < 32; i++) {
    if (x & (1 << i))
      ret++;
  }

  return ret;
}
```

```
int common_factor(int A[10]) {
  int factor = nondet();
  int i;

  for (i = 0; i < 10; i++) {
    assume((A[i] % factor) == 0);
  }

  assume(factor > 1);
  return factor;
}
```

Fig. 8. Two C- programs.

```
int fib(int n) {
  int a, b, c;

  a = b = 1;

  while (n--) {
    c = a+b;
    a = b;
    b = c;
  }

  return a;
}
```

Fig. 9. A C program to compute Fibonacci numbers.

a SAT problem of size $O(lk) = O(l)$. Conversely, a SAT problem of size s can be converted trivially into a loop free C- program of size $O(s)$. The safety problem for C- is therefore NP-complete, which means it can be decided fairly efficiently for many practical instances.

4.1 Why use C- instead of SAT?

To misquote David Wheeler: "all problems in computer science can be solved by another level of abstraction." [**?**]. C- serves as an abstraction of SAT. We can think of SAT as an assembly language and CBMC as a compiler for C- to this assembly language. The benefits of using C- as our logic rather than SAT are very similar to the benefits of writing code in a high level language rather than assembly language. Some of the more significant advantages are:

- The code is more readable and easier to debug.
- The code is much more concise.
- We can take advantage of new optimisations, different model checkers and different backends (SAT, SMT, ACDCL, etc.) for free.

To illustrate some of these points, we were able to implement a first version of KALASHNIKOV in around 150 lines of code, which took around two hours to write. The resulting code was easy to modify, optimise and debug. The current version, which includes floating point support, several optimisations and two model checking algorithms, is still less than 1200 lines of code (600 lines of C- and 600 lines of Python).
Another benefit of using C- as our logic is that it would be easy to automatically extract specifications from C programs, which would be useful in the context of superoptimisation. This is because C- is a relatively large fragment of C, so we anticipate that many interesting sections of real code will already be C-.

5 Optimizations

By far the dominant factor in the total runtime of the synthesis algorithm is the runtime of SYNTH. It is not entirely surprising that this runtime is strongly related to the size of the SAT problem generated by CBMC. The size of the SAT problem is influenced by the number of bits needed to encode P, and by the number of bits needed to encode a run of the $\mathcal{L}$-interpreter. The connection to Kolmogorov complexity is fairly obvious. The most successful optimizations we found were related to making fragments of $\mathcal{L}$ and searching progressively larger fragments until a program could be found.

5.1 Limited Program Length

The simplest restriction on programs is to only consider programs of a certain length. We start by considering programs consisting of one instruction and progressively increase this number.

5.2 Word Width

It is often the case that a program which satisfies the specification will continue to satisfy it regardless of the word width of the machine it is impemented on. For example the program in Fig. 3.2 isolates the least significant bit of an 8-bit word, but it will also isolate the least significant bit of a 32-bit or indeed any sized word.
Because of this, we can often find a program that is correct for a large word size by synthesizing a program for a machine with a smaller word size. This reduces the size of all the constants in the program, the search space of possible test inputs and also the size of all the state variables internal to the $\mathcal{L}$-interpreter.
The only wrinkle here is that sometimes a program we synthesize will contain constants. If we have synthesized a program for a machine with k-bit words, the constants in the program will be k bits wide. To generalize the program to a n-bit machine (with $n > k$), we need some way of deriving n-bit wide numbers from k-bit ones. We have several strategies for this and just try each in turn. In the following, a subscript n on a value means that the value is an n-bit word. The dot operator $a_l \cdot b_m$ denotes concatenation, resulting in an $(l+m)$-bit wide word. A superscript on a value means duplication, so a_l^m means concatenating together m copies of the l-bit word a, resulting in an (lm)-bit wide word.
To generalize a number x we use the following rules:

- If $x = k_k$, generalize to n_n.
- If $x = k - 1_k$, generalize to $n - 1_n$.
- If $x = k + 1_k$, generalize to $n + 1_n$.
- Generalize x_k to x_n.
- Generalize x_k to $x_k \cdot 0_{n-k}$.
- Generalize x_k to $x_k^{n/k}$.

5.3 Separate Constants

$\mathcal{L}$ is a SSA, three address instruction set. Destination registers are implicit and a fresh register exists for each instruction to write its output to. A natural way to encode $\mathcal{L}$ instructions is to have an opcode and two operands. The opcode selects which instruction type is being executed and the operands specify which registers should be operated on. We might allow the operands to be either a register (i.e. a program argument or the result of a previous instruction), or an immediate constant. Each opcode requires $\log_2 I$ bits to encode, where I is the number of instruction types in $\mathcal{L}$. Each operand can be encoded using $\log_2 w$ bits, where w is the $\mathcal{L}$-machine word, plus one bit to specify whether the operand is a register name or an immediate constant. One instruction can therefore be encoded using $\log_2 I + 2w + 2$ bits. For an n instruction program, we need

$$n \log_2 I + 2nw + 2n$$

bits to encode the entire program.
If we instead limit the number of constants that can appear in the program, our operands can be encoded using fewer bits. For an n instruction program using k constants and taking a arguments as inputs, each

operand can refer to a program argument, the result of a previous instruction or a constant. This can be encoded using $\log_2(k+a+n-1)$ bits, which means each instruction can be encoded in $\log_2 I + \log_2(k+a+n-1)$ and the full program needs

$$n \log_2 I + n \log_2(k + a + n - 1) + kw$$

bits to encode.
To give an example, $\mathcal{L}$ has 15 instruction types, so each opcode is 4 bits. For a 10 instruction program over 1 argument, using 2 constants on a 32-bit word machine the first encoding requires $10*(4+32+1+32+1) = 700$ bits. Using the second encoding, each operand can be represented using $\log_2(2 + 1 + 10 - 1) = 4$ bits, and the entire program requires 184 bits. This is a substantial reduction in size and when the required program requires few constants this can lead to a very significat speed up.
As with program length we can progressively increase the number of constants in our program. We start by trying to synthesize a program with no constants, then if that fails we try to synthesize using one constant and so on.

5.4 Explicit Search

When we are working on an $\mathcal{L}$-machine with a small word size and a compact program encoding, our state space is very small. We can quickly check the validity of the SYNTH and VERIF formulae by simply enumerating all of the inputs and programs, then explicitly executing each program on each input.
Since everything in our system is written in C, we can add a simple loop to enumerate all programs and inputs, then compile the code using GCC. The resulting binary will explicitly search for programs satisfying the SYNTH formula, or for counterexamples satisfying the VERIF formula. For small state spaces, these explicit search binaries are often faster than symbolically evaluating the SYNTH and VERIF formulae with CBMC.

5.5 Remove nops

Many instructions in $\mathcal{L}$ are nops that do not doing anything. For example the instruction `add x 0` does nothing. Such instructions can be removed from any program they appear in to leave a semantically equivalent, but shorter, program. We can therefore be sure that nops will never appear in any minimal program. By adding constraints saying that each instruction is not a nop, we can help the underlying SAT solver's search, which reduces the runtime of the overall procedure.

5.6 Symmetry Reduction

There are many instructions that are equivalent to each other. For example, `add x y` is equivalent to `add y x` – any program containing one instruction could have it replaced by the other instruction and keep the

same semantics. We choose a single canonical instruction to represent all instructions in a particular equivalence class, then add constraints saying that no non-canonical instructions appear in the program.
Our rules for excluding non-canonical instructions are:

- For commutative operations, the first operand is smaller than the second.
- For unary operations, the second (unused) operand is always 0.
- No instruction may have two constant operands.
- All program constants are distinct.

As with the nop constraints, these additional constraints do increase the size of the resulting SAT instance, but this still ends up as a win in terms of runtime.

Implementing $\mathcal{L}$ With A Stack Machine Three-address code tends to be less compact than code written for a stack machine. It therefore would seem reasonable to use a stack machine to execute $\mathcal{L}$ programs, since we would expect to see smaller programs (in terms of bits used in the encoding), leading to smaller SAT instances and faster runtimes overall.
Unfortunately we saw quite the opposite effect – runtimes when implementing the $\mathcal{L}$-interpreter as a stack machine were much higher than the three-address code implementation. We believe the reason for this effect is that the internal state of a stack based interpreter is much more complex to analyse than an interpreter for three-address code. This is because a three-address code interpreter can be implemented using a single array whose cells are each written to exactly once (each cell corresponds to a register in the program). By contrast a stack machine interpreter has a stack as its central data structure. It is natural to implement this as an array, but the array's contents are accessed in an unpredictable, non-uniform manner as the stack pointer increases and decreases over the lifetime of the program. This leads to a *larger* SAT instance and drastically slows down the operation of the solver.

6 Floating Point

7 Experimental Results

8 Conclusion

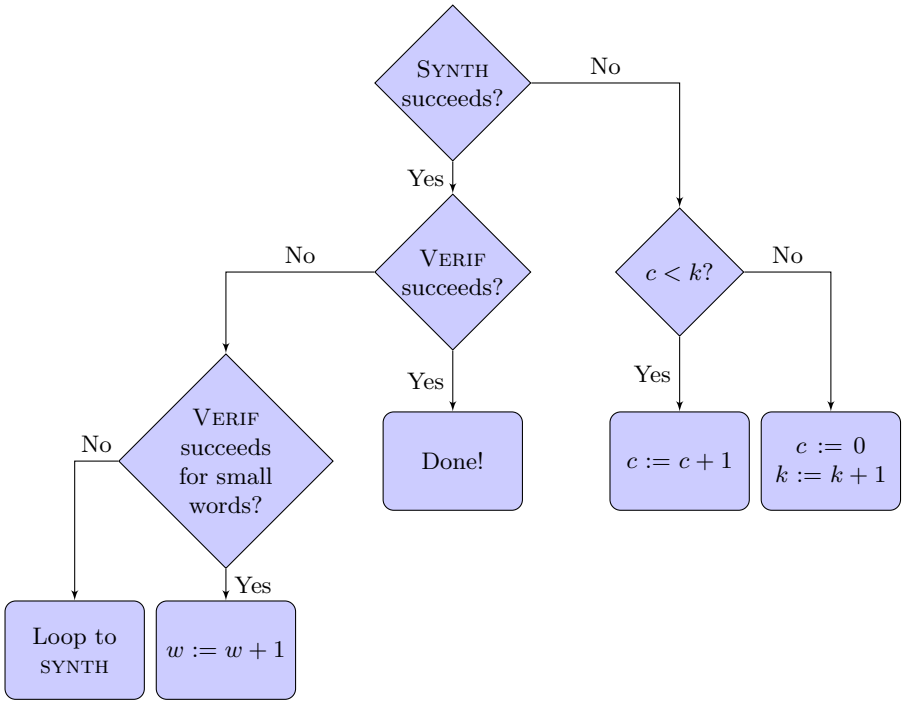

Fig. 10. Decision tree for increasing parameters of $\mathcal{L}$.