# Using Program Synthesis for Program Analysis*

Cristina David[1], Daniel Kroening[1], and Matt Lewis[1,2]

[1] University of Oxford [2] Improbable Worlds Ltd.

**Abstract.** In this paper, we propose a *unified* framework for designing static analysers based on *program synthesis*. For this purpose, we identify a fragment of second-order logic with restricted quantification that is expressive enough to capture numerous static analysis problems (e.g. safety proving, bug finding, termination and non-termination proving, superoptimisation). We call this fragment the *synthesis fragment*. We build a decision procedure for the synthesis fragment over finite domains in the form of a program synthesiser. Given our initial motivation to solve static analysis problems, this synthesiser is specialised for such analyses. Our experimental results show that, on benchmarks capturing static analysis problems, our program synthesiser compares positively with other general purpose synthesisers.

## 1 Introduction

Fundamentally, every static program analysis is searching for a *program proof*. For safety analysers this proof takes the form of a program invariant [1], for bug finders it is a counter-model [2], for termination analysis it can be a ranking function [3], whereas for non-termination it is a recurrence set [4]. Finding each of these proofs was subject to extensive research resulting in a multitude of *specialised* techniques.

In this paper, we propose a *unified* framework for designing static analysers. This framework allows implementing new analyses easily by only providing a description of the corresponding program proofs. This essentially enables a declarative way of designing static analyses, where we specify what we want to achieve rather than the details of how to achieve it.

The theoretical basis for this framework is a fragment of second-order logic with restricted quantification that is expressive enough to capture numerous static analysis problems (e.g. safety proving, bug finding, termination and non-termination proving, superoptimisation). This fragment is decidable over finite domains and we build a decision procedure for it based on *program synthesis*. Accordingly, we call this fragment *the synthesis fragment*.

In our framework, finding a program proof for some static analysis problem amounts to finding a satisfying model for a synthesis formula, where the second-order entities denote the program proofs. If the synthesis formula is satisfiable, a solution consists of a satisfying assignment from the second order variables to

* This research was supported by ERC project 280053 (CPROVER).

*functions over finite domains.* Every function over finite domains is computed by some *program* that can be synthesised.

Our program synthesiser is specialised for program analysis in the following three dimensions (identified as the three key dimensions in program synthesis [5]):
**1. Expression of user intent:** Our specification language is a fragment of C, which results in *concise specifications* of static analyses. Using our tool to build a program analyser only requires providing a generic specification of the problem to solve. The programs to be analysed do not need to be modified, symbolically executed or compiled to an intermediate language. Our experiments show that this results in specifications that are an order of magnitude smaller than the equivalent specifications with other general purpose program synthesisers.
**2. Space of programs over which to search:** The language in which we synthesise our programs is universal, i.e. every finite function is computed by at least one program in our language. Our solution language also has first-class support for *programs computing multiple outputs*, as well as *constants*. The former allows the direct encoding of lexicographic ranking functions of unbounded dimension, whereas the latter improves the efficiency when synthesising programs with non-trivial constants (as shown by our experimental results).
**3. The search technique:** An important aspect of our synthesis algorithm is how we search the space of candidate programs. We parameterise the solution language, which induces a lattice of progressively more expressive languages. As well as giving us an automatic search procedure, this parametrisation greatly increases the efficiency of our system since languages low down the lattice are very easy to decide safety for. Consequently, our solver's runtime is heavily influenced by the *length of the shortest proof*, i.e. the Kolmogorov complexity of the problem. If a short proof exists, then the solver will find it quickly. This is particularly useful for program analysis problems, where, if a proof exists, then most of the time many proofs exist and some are short ([6] relies on a similar remark about loop invariants).

*Our Contributions.*

- We define the synthesis fragment and show that its decision problem over finite domains is NEXPTIME-complete (Sec. 2).
- By using program synthesis, we design a decision procedure for the synthesis fragment. The resulting program synthesiser uses a combination of bounded model checking, explicit-state model checking and genetic programming (Sec. 5).
- We propose the use of second-order tautologies for avoiding unsatisfiable instances when solving program analysis problems with program synthesis (Sec. 8).
- We implemented the program synthesiser and tried it on a set of static analysis problems. Our experimental results show that, on benchmarks generated from static analysis, our program synthesiser compares positively with other general purpose synthesisers (Sec. 9).

*Related Work.* A recent successful approach to program synthesis is Syntax Guided Synthesis (SyGuS) [7]. The SyGuS synthesisers supplement the logical specification with a syntactic template that constrains the space of allowed implementations. Thus, each semantic specification is accompanied by a syntactic specification in the form of a grammar. In contrast to SyGuS, our program synthesiser is optimised for program analysis according to the three aforementioned key dimensions.

Other second-order solvers are introduced in [8, 9]. As opposed to ours, these are specialised for Horn clauses and the logic they handle is undecidable. Wintersteiger et al. present in [10] a decision procedure for a logic related to the synthesis fragment, the Quantified bit-vector logic, which is a many sorted first-order logic formula where the sort of every variable is a bit-vector sort. It is possible to reduce formulae in the synthesis fragment over finite domains to Effectively Propositional Logic [11], but the reduction would require additional axiomatization and would increase the search space, thus defeating the efficiency we are aiming to achieve.

## 2 The Synthesis Fragment

In this section, we identify a fragment of second-order logic with a constrained use of quantification that is expressive enough to encode numerous static analysis problems. We will suggestively refer to the fragment as the *synthesis fragment*:

**Definition 1 (Synthesis Fragment ($SF$)).** *A formula is in the* synthesis fragment *iff it is of the form*

$$\exists P_1 \ldots P_m . Q_1 x_1 \ldots Q_n x_n . \sigma(P_1, \ldots, P_m, x_1, \ldots, x_n)$$

*where the $P_i$ range over functions, the $Q_i$ are either $\exists$ or $\forall$, the $x_i$ range over ground terms and $\sigma$ is a quantifier-free formula.*

If a pair $(\vec{P}, \vec{x})$ is a satisfying model for the synthesis formula, then we write $(\vec{P}, \vec{x}) \models \sigma$. For the remainder of the presentation, we drop the vector notation and write $x$ for $\vec{x}$, with the understanding that all quantified variables range over vectors.

## 3 Program Analysis Specifications in the Synthesis Fragment

Program analysis problems can be reduced to the problem of finding solutions to a second-order constraint [12, 8, 13]. The goal of this section is to show that the synthesis fragment is expressive enough to capture many interesting such problems. For brevity reasons, we will only express safety, termination and non-termination. When we describe analyses involving loops, we will characterise each loop as having initial state $I$, guard $G$ and transition relation $B$.

*Safety Invariants.* Given a safety assertion $A$, a safety invariant is a set of states $S$ which is inductive with respect to the program's transition relation, and which excludes an error state. A predicate $S$ is a safety invariant iff it satisfies the following criteria:

$$\begin{aligned} \exists S.\forall x, x'.I(x) \to S(x) \wedge & \quad (1)\\ S(x) \wedge G(x) \wedge B(x,x') \to S(x') \wedge & \quad (2)\\ S(x) \wedge \neg G(x) \to A(x) & \quad (3)\end{aligned}$$

(1) says that each state reachable on entry to the loop is in the set $S$, and in combination with (2) shows that every state that can be reached by the loop is in $S$. The final criterion (3) says that if the loop exits while in an $S$-state, the assertion $A$ is not violated.

*Termination and non-termination.* As shown in [13], termination of a loop can be encoded as the following formula, where $W$ is an inductive invariant of the loop that is established by the initial states $I$ if the loop guard $G$ is met, and $R$ is a ranking function as restricted by $W$:

$$\begin{aligned}&\exists R, W.\forall x, x'.I(x) \wedge G(x) \to W(x) \wedge \\ &\qquad G(x) \wedge W(x) \wedge B(x,x') \to W(x') \wedge R(x){>}0 \wedge R(x){>}R(x')\end{aligned}$$

Similarly, non-termination can be expressed in the synthesis fragment as follows:

$$\exists N, C, x_0.\forall x.N(x_0) \wedge N(x) \to G(x) \wedge N(x) \to B(x, C(x)) \wedge N(C(x))$$

Here, $N$ denotes a recurrence set, i.e. a nonempty set of states such that for each $s \in N$ there exists a transition to some $s' \in N$, and $C$ is a Skolem function that chooses the successor $x'$. More details on the formulations for termination and non-termination can be found in [13].

## 4 The Synthesis Fragment over Finite Domains

When interpreting the ground terms over a finite domain $\mathcal{D}$, the synthesis fragment is decidable and its decision problem is NEXPTIME-complete. We use the notation $SF_{\mathcal{D}}$ to denote the synthesis fragment over a finite domain $\mathcal{D}$.

**Theorem 1 ($SF_{\mathcal{D}}$ is NEXPTIME-complete).** *For an instance of Definition 1 with $n$ first-order variables, where the ground terms are interpreted over $\mathcal{D}$, checking the truth of the formula is NEXPTIME-complete.*

*Proof.* In the extended version [14].

Next, we are concerned with building a solver for $SF_{\mathcal{D}}$. A satisfying model for a formula in $SF_{\mathcal{D}}$ is an assignment mapping each of the second-order variables to some function of the appropriate type and arity. When deciding whether a particular $SF_{\mathcal{D}}$ instance is satisfiable, we should think about how solutions are encoded and in particular how a function is to be encoded. The functions all have a finite domain and co-domain, so their canonical representation would be a finite set of ordered pairs. Such a set is exponentially large in the size of the domain, so we would prefer to work with a more compact representation if possible.

We will generate *finite state programs* that compute the functions and represent these programs as finite lists of instructions in SSA form. This representation has the following properties, proofs for which can be found in the extended version [14].

**Theorem 2.** *Every total, finite function is computed by at least one finite state program.*

**Theorem 3.** *Furthermore, this representation as finite lists of instructions in SSA form is optimally concise – there is no encoding that gives a shorter representation to every function.*

*Finite State Program Synthesis* To formally define the finite state synthesis problem, we need to fix some notation. We will say that a program $P$ is a finite list of instructions in SSA form, where no instruction can cause a back jump, i.e. our programs are loop free and non-recursive. Inputs $x$ to the program are drawn from some finite domain $\mathcal{D}$. The synthesis problem is given to us in the form of a specification $\sigma$ which is a function taking a program $P$ and input $x$ as parameters and returning a boolean telling us whether $P$ did "the right thing" on input $x$. Basically, the finite state synthesis problem checks the truth of Definition 2.

**Definition 2 (Finite Synthesis Formula).**

$$\exists P. \forall x \in \mathcal{D}. \sigma(P, x)$$

To express the specification $\sigma$, we introduce a function $\texttt{exec}(P, x)$ that returns the result of running program $P$ with input $x$. Since $P$ cannot contain loops or recursion, `exec` is a total function.

*Example 1.* The following finite state synthesis problem is satisfiable:

$$\exists P. \forall x \in \mathbb{N}_8 . \texttt{exec}(P, x) \geq x$$

One such program $P$ satisfying the specification is `return 8`, which just returns 8 for any input.

We now present our main theorem, which says that satisfiability of $SF_{\mathcal{D}}$ can be reduced to finite state program synthesis. The proof of this theorem can be found in the extended version [14].

**Theorem 4 ($SF_{\mathcal{D}}$ is Polynomial Time Reducible to Finite Synthesis).** *Every instance of Definition 1, where the ground terms are interpreted over $\mathcal{D}$ is polynomial time reducible to a finite synthesis formula (i.e. an instance of Definition 2).*

**Corollary 1.** *Finite-state program synthesis is NEXPTIME-complete.*

We are now in a position to sketch the design of a decision procedure for $SF_{\mathcal{D}}$: we will convert the $SF_{\mathcal{D}}$ satisfiability problem to an equisatisfiable finite synthesis problem, which we will then solve with a finite state program synthesiser. This design will be elaborated in Section 5.

## 5 Deciding $SF_{\mathcal{D}}$ via Finite-State Program Synthesis

In this section we will present a sound and complete algorithm for finite-state synthesis that we use to decide the satisfiability of formulae in $SF_{\mathcal{D}}$. We begin by describing a general purpose synthesis procedure (Section 5.1), then detail how this general purpose procedure is instantiated for synthesising finite-state programs. We then describe the algorithm we use to search the space of possible programs (Sections 5.3 and 6).

### 5.1 General Purpose Synthesis Algorithm

**Algorithm 1** Abstract refinement algorithm

```
1: function SYNTH(inputs)
2:     $(i_1, \ldots, i_N) \leftarrow$ inputs
3:     query $\leftarrow \exists P.\sigma(i_1, P) \wedge \ldots \wedge \sigma(i_N, P)$
4:     result $\leftarrow$ decide(query)
5:     if result.satisfiable then
6:         return result.model
7:     else
8:         return UNSAT

9: function VERIF(P)
10:    query $\leftarrow \exists x.\neg\sigma(x, P)$
11:    result $\leftarrow$ decide(query)
12:    if result.satisfiable then
13:        return result.model
14:    else
15:        return valid

16: function REFINEMENT LOOP
17:    inputs $\leftarrow \emptyset$
18:    loop
19:        candidate $\leftarrow$ SYNTH(inputs)
20:        if candidate = UNSAT then
21:            return UNSAT
22:        res $\leftarrow$ VERIF(candidate)
23:        if res = valid then
24:            return candidate
25:        else
26:            inputs $\leftarrow$ inputs $\cup$ res
```

We use Counterexample Guided Inductive Synthesis (CEGIS) [15, 16] to find a program satisfying our specification. Algorithm 1 is divided into two procedures: SYNTH and VERIF, which interact via a finite set of test vectors INPUTS. The SYNTH procedure tries to find an existential witness $P$ that satisfies the partial specification: $\exists P.\forall x \in \text{INPUTS}.\sigma(x, P)$

If SYNTH succeeds in finding a witness $P$, this witness is a candidate solution to the full synthesis formula. We pass this candidate solution to VERIF which determines whether it does satisfy the specification on all inputs by checking satisfiability of the verification formula: $\exists x.\neg\sigma(x, P)$

If this formula is unsatisfiable, the candidate solution is in fact a solution to the synthesis formula and so the algorithm terminates. Otherwise, the witness $x$ is an input on which the candidate solution fails to meet the specification. This witness $x$ is added to the INPUTS set and the loop iterates again. It is worth noting that each iteration of the loop adds a new input to the set of inputs being used for synthesis. If the full set of inputs is finite, this means that the refinement loop can only iterate a finite number of times.

## 5.2 Finite-State Synthesis

We will now show how the generic construction of Section 5.1 can be instantiated to produce a finite-state program synthesiser. A natural choice for such a synthesiser would be to work in the logic of quantifier-free propositional formulae and to use a propositional SAT or SMT-$\mathcal{BV}$ solver as the decision procedure. However we propose a slightly different tack, which is to use a decidable fragment of C as a "high level" logic. We call this fragment $C^-$.

The characteristic property of a $C^-$ program is that safety can be decided for it using a single query to a Bounded Model Checker. A $C^-$ program is just a C program with the following syntactic restrictions:
(i) all loops in the program must have a constant bound;
(ii) all recursion in the program must be limited to a constant depth;
(iii) all arrays must be statically allocated (i.e. not using `malloc`), and be of constant size.
$C^-$ programs may use nondeterministic values, assumptions and arbitrary-width types.

Since each loop is bounded by a constant, and each recursive function call is limited to a constant 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 a SAT problem of size $O(lk)$. 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.

## 5.3 Candidate Generation Strategies

A candidate solution $P$ is written in a simple RISC-like language $\mathcal{L}$, whose syntax is given in the extended version [14]. We supply an interpreter for $\mathcal{L}$ which is written in $C^-$. The specification function $\sigma$ will include calls to this interpreter, by which means it will examine the behaviour of a candidate $\mathcal{L}$ program.

For the SYNTH portion of the CEGIS loop, we construct a $C^-$ program SYNTH.C which takes as parameters a candidate program $P$ and test inputs. The program contains an assertion which fails iff $P$ meets the specification for each of the inputs. Finding a new candidate program is then equivalent to checking the safety of SYNTH.C. There are many possible strategies for finding these candidates; we employ the following strategies in parallel:

*(i) Explicit Proof Search.* The simplest strategy for finding candidates is to just exhaustively enumerate them all, starting with the shortest and progressively increasing the number of instructions.

*(ii) Symbolic Bounded Model Checking.* Another complete method for generating candidates is to simply use BMC on the SYNTH.C program.

*(iii) Genetic Programming and Incremental Evolution.* Our final strategy is genetic programming (GP) [17, 18]. GP provides an adaptive way of searching through the space of $\mathcal{L}$-programs for an individual that is "fit" in some sense. We

measure the fitness of an individual by counting the number of tests in INPUTS for which it satisfies the specification.

To bootstrap GP in the first iteration of the CEGIS loop, we generate a population of random $\mathcal{L}$-programs. We then iteratively evolve this population by applying the genetic operators CROSSOVER and MUTATE. CROSSOVER combines selected existing programs into new programs, whereas MUTATE randomly changes parts of a single program. Fitter programs are more likely to be selected.

Rather than generating a random population at the beginning of each subsequent iteration of the CEGIS loop, we start with the population we had at the end of the previous iteration. The intuition here is that this population contained many individuals that performed well on the $k$ inputs we had before, so they will probably continue to perform well on the $k + 1$ inputs we have now. In the parlance of evolutionary programming, this is known as incremental evolution [19].

## 6 Searching the Space of Possible Solutions

An important aspect of our synthesis algorithm is the manner in which we search the space of candidate programs. The key component is parametrising the language $\mathcal{L}$, which induces a lattice of progressively more expressive languages. We start by attempting to synthesise a program at the lowest point on this lattice and increase the parameters of $\mathcal{L}$ until we reach a point at which the synthesis succeeds. Note that this parametrisation applies to all three strategies in the previous section.

As well as giving us an automatic search procedure, this parametrisation greatly increases the efficiency of our system since languages low down the lattice are very easy to decide safety for. If a program can be synthesised in a low-complexity language, the whole procedure finishes much faster than if synthesis had been attempted in a high-complexity language.

### 6.1 Parameters of language $\mathcal{L}$

*Program Length: $l$.* The first parameter we introduce is program length, denoted by $l$. At each iteration we synthesise programs of length exactly $l$. We start with $l = 1$ and increment $l$ whenever we determine that no program of length $l$ can satisfy the specification. When we do successfully synthesise a program, we are *guaranteed that it is of minimal length* since we have previously established that no shorter program is correct.

*Word Width: $w$.* An $\mathcal{L}$-program runs on a virtual machine (the $\mathcal{L}$-machine) that is parametrised by the *word width*, that is, the number of bits in each internal register and immediate constant.

*Number of Constants: $c$.* Instructions in $\mathcal{L}$ take up to three operands. Since any instruction whose operands are all constants can always be eliminated (since its result is a constant), we know that a loop-free program of minimal length will not contain any instructions with two constant operands. Therefore the number of constants that can appear in a minimal program of length $l$ is at most $l$. By minimising the number of constants appearing in a program, we are able to use

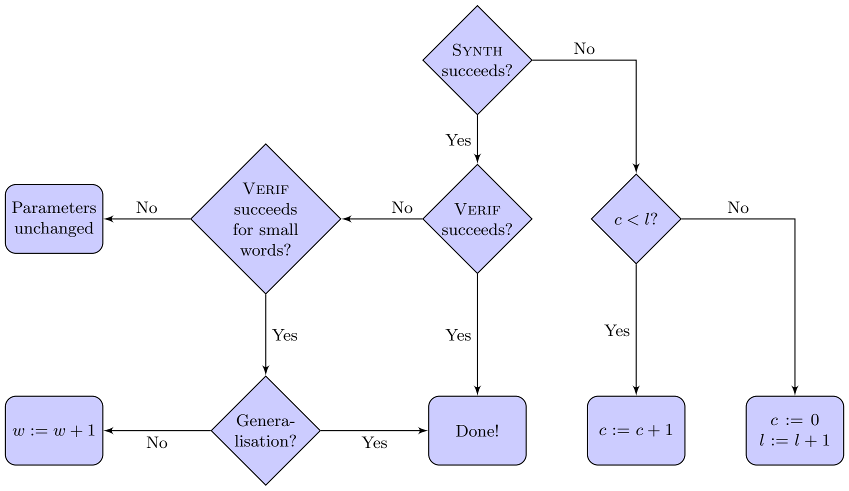

Fig. 1: Decision tree for increasing parameters of $\mathcal{L}$.

a particularly efficient program encoding that speeds up the synthesis procedure substantially.

## 6.2 Searching the Program Space

The key to our automation approach is to come up with a sensible way in which to adjust the $\mathcal{L}$-parameters in order to cover all possible programs. Two important components in this search are the adjustment of parameters and the generalisation of candidate solutions. We discuss them both next.

*Adjusting the search parameters.* After each round of SYNTH, we may need to adjust the parameters. The logic for these adjustments is given as a tree in Fig. 1.

Whenever SYNTH fails, we consider which parameter might have caused the failure. There are two possibilities: either the program length $l$ was too small, or the number of allowed constants $c$ was. If $c < l$, we just increment $c$ and try another round of synthesis, but allowing ourselves an extra program constant. If $c = l$, there is no point in increasing $c$ any further. This is because no minimal $\mathcal{L}$-program has $c > l$, for if it did there would have to be at least one instruction with two constant operands. This instruction could be removed (at the expense of adding its result as a constant), contradicting the assumed minimality of the program. So if $c = l$, we set $c$ to 0 and increment $l$, before attempting synthesis again.

If SYNTH succeeds but VERIF fails, we have a candidate program that is correct for some inputs but incorrect on at least one input. However, it may be the case that the candidate program is correct for *all* inputs when run on an $\mathcal{L}$-machine with a small word size. Thus, we try to generalise the solution to a bigger word size, as explained in the next paragraph. If the generalisation is able to find a correct program, we are done. Otherwise, we need to increase the word width of the $\mathcal{L}$-machine we are currently synthesising for.

*Generalisation of candidate solutions.* It is often the case that a program which satisfies the specification on an $\mathcal{L}$-machine with $w = k$ will continue to satisfy the specification when run on a machine with $w > k$. For example, the program in Fig. 2 isolates the least-significant bit of a word. This is true irrespective of the word size of the machine it is run on – it will isolate the least-significant bit of an 8-bit word just as well as it will a 32-bit word. An often successful strategy is to synthesise a program for an $\mathcal{L}$-machine with a small word size and then to check whether the same program is correct when run on an $\mathcal{L}$-machine with a full-sized word.

The only wrinkle here is that we will sometimes synthesise a program containing constants. If we have synthesised a program with $w = k$, the constants in the program will be $k$-bits wide. To extend the program to an $n$-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. Our strategies are shown in Fig. 3. $\mathcal{BV}(v, n)$ denotes an $n$-bit wide bitvector holding the value $v$ and $b \cdot c$ means the concatenation of bitvectors $b$ and $c$. For example, the first rule says that if we have the 8-bit number with value 8, and we want to extend it to some 32-bit number, we'd try the 32-bit number with value 32. These six rules are all heuristics that we have found to be fairly effective in practice.

```
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. 2: A tricky bitvector program

$$\begin{array}{rl} \mathcal{BV}(m, m) \to \mathcal{BV}(n, n) & \mathcal{BV}(x, m) \to \mathcal{BV}(x, n) \\ \mathcal{BV}(m-1, m) \to \mathcal{BV}(n-1, n) & \mathcal{BV}(x, m) \to \mathcal{BV}(x, m) \cdot \mathcal{BV}(0, n-m) \\ \mathcal{BV}(m+1, m) \to \mathcal{BV}(n+1, n) & \mathcal{BV}(x, m) \to \underbrace{\mathcal{BV}(x, m) \cdot \ldots \cdot \mathcal{BV}(x, m)}_{\frac{n}{m}\text{ times}} \end{array}$$

Fig. 3: Rules for extending an $m$-bit wide number to an $n$-bit wide one.

## 6.3 Stopping Condition for Unsatisfiable Specifications

If a specification is unsatisfiable, we would still like our algorithm to terminate with an "unsatisfiable" verdict. To do this, we can observe that any total function taking $n$ bits of input is computed by some program of at most $2^n$ instructions (a consequence of Theorems 2 and 3). Therefore every satisfiable specification has a solution with at most $2^n$ instructions. This means that if we ever need to increase the length of the candidate program we search for beyond $2^n$, we can terminate, safe in the knowledge that the specification is unsatisfiable.

Although this gives us a theoretical termination condition for unsatisfiable instances, in practice the program synthesiser may not terminate. In order to avoid such cases, we use the approach described in Sec 8.

## 7 Soundness, Completeness and Efficiency

We will now state soundness and completeness results for the $SF_{\mathcal{D}}$ solver. Proofs for each of these theorems can be found in the extended version [14].

**Theorem 5.** *Alg 1 is sound – if it terminates with witness $P$, then $P \models \sigma$.*

**Theorem 6.** *Alg 1 with the stopping condition described in Section 6.3 is complete when instantiated with $C^-$ as a background theory – it will terminate for all specifications $\sigma$.*

Since safety of $\mathrm{C}^-$ programs is decidable, Algorithm 1 is sound and complete when instantiated with $\mathrm{C}^-$ as a background theory and using the stopping condition of Section 6.3. This construction therefore gives as a decision procedure for $SF_{\mathcal{D}}$.

*Runtime as a Function of Solution Size.* We note that the runtime of our solver is heavily influenced by the length of the shortest program satisfying the specification, since we begin searching for short programs. We will now show that the number of iterations of the CEGIS loop is a function of the Kolmogorov complexity of the synthesised program. Let us first recall the definition of the Kolmogorov complexity of a function $f$:

**Definition 3 (Kolmogorov complexity).** *The Kolmogorov complexity $K(f)$ is the length of the shortest program that computes $f$.*

We can extend this definition slightly to talk about the Kolmogorov complexity of a synthesis problem in terms of its specification:

**Definition 4 (Kolmogorov complexity of a synthesis problem).** *The Kolmogorov complexity of a program specification $K(\sigma)$ is the length of the shortest program $P$ such that $P$ is a witness to the satisfiability of $\sigma$.*

Let us consider the number of iterations of the CEGIS loop $n$ required for a specification $\sigma$. Since we enumerate candidate programs in order of length, we are always synthesising programs with length no greater than $K(\sigma)$ (since when we enumerate the first correct program, we will terminate). So the space of solutions we search over is the space of functions computed by $\mathcal{L}$-programs of length no greater than $K(\sigma)$. Let's denote this set $\mathcal{L}(K(\sigma))$. Since there are $O(2^{K(\sigma)})$ *programs* of length $K(\sigma)$ and some functions will be computed by more than one program, we have $|\mathcal{L}(K(\sigma))| \leq O(2^{K(\sigma)})$.

Each iteration of the CEGIS loop distinguishes at least one incorrect function from the set of correct functions, so the loop will iterate no more than $|\mathcal{L}(K(\sigma))|$ times. Therefore another bound on our runtime is $NTIME\left(2^{K(\sigma)}\right)$.

## 8 Avoiding Unsatisfiable Instances

As described in the previous section, our program synthesiser is efficient at finding satisfying assignments, when such assignments have low Kolmogorov complexity. However, if a formula is unsatisfiable, the procedure may not terminate in practice. This illustrates one of the current shortcomings of our program synthesis based decision procedure: we can only conclude that a formula is unsatisfiable once we have examined candidate solutions up to a very high length bound.

However, we note that many interesting properties of programs can be expressed as tautologies. For illustration, let us consider that we are trying to prove that a loop $L$ terminates. Thus, as shown in Sec 3, we can construct two formulae: one that is satisfiable iff $L$ is terminating and another that is satisfiable iff $L$ is non-terminating. We will call these formulae $\phi$ and $\psi$, respectively, and we denote by $P_N$ and $P_T$ the proofs of non-termination and termination, respectively: $\exists P_T.\forall x, x'.\phi(P_T, x, x')$ and $\exists P_N.\forall x.\psi(P_N, x)$.

We can combine these: $(\exists P_T.\forall x, x'.\phi(P_T, x, x')) \vee (\exists P_N.\forall x.\, \psi(P_N, x))$.

Which simplifies to: $\exists P_T, P_N.\forall x, x', y.\, \phi(P_T, x, x') \vee \psi(P_N, y)$.

Since $L$ either terminates or does not terminate, this formula is a tautology in the synthesis fragment. Thus, either $P_N$ or $P_T$ must exist. Similarly, when proving safety, a program is either safe of has a bug. In this manner we avoid the bad case where we try to synthesise a solution for an unsatisfiable specification.

## 9 Experiments

We implemented our decision procedure for $SF_{\mathcal{D}}$ as the Kalashnikov tool. We used Kalashnikov to solve formulae generated from a variety of problems taken from superoptimisation, code deobfuscation, floating point verification, ranking function and recurrent set synthesis, safety proving, and bug finding. The superoptimisation and code deobfuscation benchmarks were taken from the experiments of [20]; the termination benchmarks were taken from SVCOMP'15 [21] and they include the experiments of [13]; the safety benchmarks are taken from the experiments of [22].

We ran our experiments on a 4-core, 3.30 GHz Core i5 with 8 GB of RAM. Each benchmark was run with a timeout of 180 s. The results are shown in Table 1. For each category of benchmarks, we report the total number of benchmarks in that category, the number we were able to solve within the time limit, the average solution size (in instructions), the average number of iterations of the CEGIS loop, the average time and total time taken. The deobfuscation and floating point benchmarks are considered together with the superoptimisation ones.

For the termination benchmarks, Kalashnikov must prove that the input program is either terminating or non-terminating, i.e. it must synthesise either ranking functions and supporting invariants, or recurrence sets. For the safety benchmarks, Kalashnikov must prove that the program is either safe or unsafe. For this purpose, it synthesises either a safety invariant or a compact representations of an error trace.

*Discussion of the experimental results.* The timings show that for the instances where we can find a satisfying assignment, we tend to do so quite quickly (on the order of a few seconds). Furthermore the programs we synthesise are often short, even when the problem domain is very complex, such as for liveness and safety.

| Category | #Benchmarks | #Solved | Avg. solution size | Avg. iterations | Avg. time (s) | Total time (s) |
|---|---|---|---|---|---|---|
| Superoptimisation | 29 | 22 | 4.1 | 2.7 | 7.9 | 166.1 |
| Termination | 47 | 35 | 5.7 | 14.4 | 11.2 | 392.9 |
| Safety | 20 | 18 | 8.3 | 7.1 | 11.3 | 203.9 |
| Total | 96 | 75 | 5.9 | 9.2 | 10.3 | 762.9 |

Table 1: Experimental results.

To help understand the role of the different solvers involved in the synthesis process, we provide a breakdown of how often each solver "won", i.e. was the first to return an answer. This breakdown is given in Table 2a. We see that GP and explicit account for the great majority of the responses, with the load spread fairly evenly between them. This distribution illustrates the different strengths of each solver: GP is very good at generating candidates, explicit is very good at finding counterexamples and CBMC is very good at proving that candidates are correct. The GP and explicit numbers are similar because they are approximately "number of candidates found" and "number of candidates refuted" respectively. The CBMC column is approximately "number of candidates proved correct". The spread of winners here shows that each of the search strategies is contributing something to the overall search and that the strategies are able to co-operate with each other.

| CBMC | Explicit | GP |
|---|---|---|
| 21% | 46% | 31% |

(a) How often each solver "wins".

| SYNTH | VERIF | GENERALIZE |
|---|---|---|
| 86% | 14% | 7% |

(b) Where the time is spent.

Table 2: Statistics about the experimental results.

To help understand where the time is spent in our solver, Table 2b how much time is spent in SYNTH, VERIF and constant generalization. Note that generalization counts towards VERIF's time. We can see that synthesising candidates takes much longer than verifying them, which suggests that improved procedures for candidate synthesis will lead to good overall performance improvements. However, the times considered for this table include all the runs that timed out, as well as those that succeeded. We have observed that runs which time out spend more time in synthesis than runs which succeed, so the distribution here is biased by the cost of timeouts.

## 9.1 Comparison to SyGuS

In order to compare KALASHNIKOV to other program synthesisers, we translated the 20 safety benchmarks into the SyGuS format [7] (for the bitvector theory) and ran the enumerative CEGIS solver ESOLVER, winner of the SyGuS 2014 competition (taken from the SyGuS Github repository on 5/7/2015), as well as the program synthesiser in CVC4 [23] (the version for the SyGuS 2015 competition on the StarExec platform [24]), winner of the SyGuS 2015 competition. We could not compare against ICE-DT [25], the winner of the invariant generation category in the SyGuS 2015 competition, as it does not seem to offer support for bitvectors. Our comparison only uses 20 of the 96 benchmarks as we had to manually convert from our specification format (a subset of C) into the SyGuS format. Moreover, our choice of benchmarks was also restricted by the fact that we could not express lexicographic ranking functions of unbounded dimension in the SyGuS format, which we require for our termination benchmarks.

The results of these experiments are given in Table 3, which contains the number of benchmarks solved correctly, the number of timeouts, the number of crashes (exceptions thrown by the solver), the mean time to successfully solve and the total number of lines in the 20 specifications.

Since the eSolver tool crashed on many of the instances we tried, we reran the experiments on the StarExec platform to check that we had not made mistakes setting up our environment, however the same instances also caused exceptions on StarExec.

An important point to notice in Table 3 is that Kalashnikov specifications are significantly more concise than SyGuS specifications, as witnessed by the total size of the specifications: the Kalashnikov specifications are around 11% of the size of the SyGuS ones. Overall, we can see that Kalashnikov performs better on these benchmarks than eSolver and CVC4, which validates our claim that Kalashnikov is suitable for program analysis problems.

We noticed that for a lot of the cases in which eSolver and CVC4 timed out, Kalashnikov found a solution that involved non-trivial constants. Since the SyGuS format represents constants in unary (as chains of additions), finding programs containing constants, or finding existentially quantified first order variables is expensive. Kalashnikov's strategies for finding and generalising constants make it much more efficient at this subtask.

| | #Solved | #TO | #Crashes | Avg. time (s) | Spec. size |
|---|---|---|---|---|---|
| Kalashnikov | 18 | 2 | 0 | 11.3 | 341 |
| eSolver | 7 | 5 | 8 | 13.6 | 3140 |
| CVC4 | 5 | 13 | 2 | 61.7 | 3140 |

Table 3: Comparison of Kalashnikov, eSolver and CVC4 on the safety benchmarks.

## 10 Conclusions

We have shown that the synthesis fragment is well-suited for program verification by using it to directly encode safety, liveness and superoptimisation properties.

We built a decision procedure for $SF_{\mathcal{D}}$ via a reduction to finite state program synthesis. The synthesis algorithm is optimised for program analysis and uses a combination of symbolic model checking, explicit state model checking and stochastic search. An important strategy is generalisation – we find simple solutions that solve a restricted case of the specification, then try to generalise to a full solution. We evaluated the program synthesiser on several static analysis problems, showing the tractability of the approach.

## References

1. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. (1977)
2. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC. (2003) 368–371
3. Floyd, R.W.: Assigning meanings to programs. (1967)
4. Gupta, A., et al.: Proving non-termination. In: POPL. (2008)
5. Gulwani, S.: Dimensions in program synthesis. In: Formal Methods in Computer-Aided Design, FMCAD. (2010) 1
6. Kong, S., Jung, Y., David, C., Wang, B., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: APLAS. (2010)
7. Alur, R., et al.: Syntax-guided synthesis. In: FMCAD. (2013)
8. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI. (2012) 405–416
9. Beyene, T.A., Popeea, C., Rybalchenko, A.: Solving existentially quantified Horn clauses. In: CAV. (2013) 869–882
10. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. In: FMCAD. (2010)
11. Piskac, R., de Moura, L.M., Bjørner, N.: Deciding effectively propositional logic using DPLL and substitution sets. J. Autom. Reasoning **44**(4) (2010) 401–424
12. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI. (2008) 281–292
13. David, C., Kroening, D., Lewis, M.: Unrestricted termination and non-termination proofs for bit-vector programs. In: ESOP. (2015)
14. David, C., Kroening, D., Lewis, M.: Using program synthesis for program analysis. CoRR **abs/1508.07829** (2015)
15. Solar-Lezama, A.: Program sketching. STTT **15**(5-6) (2013) 475–495
16. Brain, M., et al.: TOAST: Applying answer set programming to superoptimisation. In: ICLP. (2006)
17. Langdon, W.B., Poli, R.: Foundations of Genetic Programming. Springer (2002)
18. Brameier, M., Banzhaf, W.: Linear Genetic Programming. Springer (2007)
19. Gomez, F., Miikkulainen, R.: Incremental evolution of complex general behavior. Adaptive Behavior (1997)
20. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: PLDI. (2011) 62–73
21. SV-COMP: http://sv-comp.sosy-lab.org/2015/.
22. David, C., Kroening, D., Lewis, M.: Danger invariants. CoRR (2015)
23. Reynolds, A., Deters, M., Kuncak, V., Tinelli, C., Barrett, C.W.: Counterexample-guided quantifier instantiation for synthesis in SMT. In: Computer Aided Verification, CAV. (2015) 198–216
24. StarExec: https://www.starexec.org.
25. Garg, P., Löding, C., Madhusudan, P., Neider, D.: ICE: A robust framework for learning invariants. In: Computer Aided Verification, CAV. (2014) 69–87