Modular Synthesis of Sketches using Models

Rohit Singh, Rishabh Singh, Zhilei Xu, Rebecca Krosnick, and Armando Solar-Lezama [⋆]

Massachusetts Institute of Technology

Abstract. One problem with the constraint-based approaches to synthesis that have become popular over the last few years is that they only scale to relatively small routines, on the order of a few dozen lines of code. This paper presents a mechanism for modular reasoning that allows us to break larger synthesis problems into small manageable pieces. The approach builds on previous work in the verification community of using high-level specifications and partially interpreted functions (we call them models) in place of more complex pieces of code in order to make the analysis modular.

The main contribution of this paper is to show how to combine these techniques with the counterexample guided synthesis approaches used to efficiently solve synthesis problems. Specifically, we show two new algorithms; one to efficiently synthesize functions that use models, and another one to synthesize functions while ensuring that the behavior of the resulting function will be in the set of behaviors allowed by the model. We have implemented our approach on top of the open-source Sketch synthesis system, and we demonstrate its effectiveness on several Sketch benchmark problems.

1 Introduction

Over the last few years, constraint-based approaches to synthesis based on sketches or templates have become quite popular [9, 17, 24, 26, 27]. In these approaches, the user specifies her intent with a template of the desired solution leaving parts of the code unspecified (a sketch), and the synthesizer finds the unknown code fragments such that the completed template conforms to a given set of behavioral constraints (the spec). One problem with such approaches, however, is that they only scale to relatively simple routines, on the order of a few dozen lines of code. Scaling such methods to more complex programs requires a mechanism for modular reasoning.

The idea of modular reasoning has been quite successful and is widely used today in verification of software and hardware, where pre and post-conditions are commonly used to model complex functions (*e.g.* MAGIC [5],DAFNY [12]), and where uninterpreted or partially interpreted functions play an important role in abstracting away complex functional units [3, 4]. These assume-guarantee

[⋆] This work was funded by NSF grants NSF-1116362, NSF-1139056, and DOE grant DE-SC0005372.

reasoning based approaches perform compositional verification by breaking down the verification task of a system into smaller tasks that involve verification of individual components, which enables the verification tools to then compose the proofs to verify the whole system [16, 28].

In this paper, we present a mechanism of modular reasoning for synthesizing complex sketches, where we use *function models* to specify the behavior of constituent function components. A function model consists of three components: pre-processing code that canonicalizes the input, an uninterpreted function that models the function behavior, and a post-condition that specifies the desired properties of a function. A model can also have a pre-condition that specifies which parameters are legal for the function. However, we place two restriction on models: 1) they cannot have any unknown code fragment (holes) and 2) they cannot post-process the output of the uninterpreted function before returning it. These function models are more general than pre and post-conditions without uninterpreted functions, but are less general than pre and post-conditions with quantifiers. This intermediate generality of function models provides us the expressiveness to specify many complex functions and at the same time allows us to efficiently use them for synthesis.

The function models in sketches introduce two new synthesis problems: 1) synthesis with models and 2) synthesis for adherence. The synthesis with models problem requires us to solve for unknown control parameters such that for any uninterpreted function that satisfies the model's post-condition, the specification of main function should also hold. This problem in principle can be solved by the traditional Cegis [22] algorithm but Cegis performs poorly in practice. We present a new algorithm Cegis+ that combines the Cegis algorithm with an approach that selectively uses existential quantification for some inputs [9] for efficiently solving this problem. The sketch will use the function model in place of a more complex function that itself may have unknowns. Therefore, we also need to solve a second synthesis problem: synthesis for adherence, which ensures that this more complex function is synthesized in a way that matches the behavior promised by the model. This synthesis problem introduces a standard doubly quantified constraint but with an existentially quantified function to find a function that satisfies the model post-condition on all valid inputs and has an equivalent input-output relationship with the original function. To solve this problem, we present an approach to eliminate the existentially quantified function by taking advantage of the specific form of the constraint and do not require additional function templates unlike previous approaches [22, 31].

We have implemented the algorithms in the Sketch synthesis system [23] and present the evaluation of the algorithm on several benchmark problems. We show that function models enable synthesis of several complex benchmarks, and our algorithm outperforms both Cegis and previously published algorithms.

Specifically, the paper makes the following contributions:

- We show that the existing approaches to counterexample guided synthesis break down in the presence of models and present a new algorithm that can efficiently synthesize functions that use models.

– We present a new way to encode the problem of synthesizing a function that behaves according to a model without the need for existentially quantified functions.
– We present the evaluation of our algorithm on several benchmark problems and show how it enables solving of complex sketches.

2 Motivating Examples

We use two running examples to motivate the need of function models: integer square root and big integer multiplication. They represent different kinds of models, namely fully specified models and partially specified models, which will expose different aspects of our algorithm.

2.1 Example 1: Square root for primality testing

For the first example, consider the sketch of an algorithm for primality testing shown in Figure 1(a). The sketch requires the synthesizer to discover many of the details of a fast primality testing algorithm that computes much smaller number of divisibility checks than $\sqrt{\texttt{p}}$ for every input `p`. Most importantly for our purposes, the sketch calls a `sqrt` function to compute the integer square root. This `sqrt` function is also sketched—not shown in the figure—so the synthesizer would normally have to derive the details of the primality test and of the `sqrt` function simultaneously. The correctness of the resulting implementation will be established by comparing the result to that of a linear time primality test, one that simply tries to divide `p` against all integers less than `p`. SKETCH uses bounded reasoning when deriving the details; in the case of this example, it will only consider values of `p` with up to 8 bits. The sketch uses the `optimize` function to ensure that the bound `bnd` is minimized for any value of `p`. The `linexp` construct is not a function but a generator that must be replaced by the synthesizer with a linear expression over its arguments and the `minrepeat` construct repeats its argument statement minimum number of times (with different holes) such that the sketch becomes satisfiable.

Instead of solving for the primality test and the square root simultaneously, we can write a model of the `sqrt` function to express its main properties. In general, a model calls an uninterpreted function and then asserts some properties of the return value(s), which corresponds to the post-conditions of the modeled function. The model allows us to break the problem into two sub-problems: 1) we need to solve the main sketch using the model in place of the more complex square root function, 2) we need to solve for the details of the square root function under the constraint that it behaves according to the model. Note that in addition to establishing the post-conditions, the model also expresses the fact that when called twice with the same input, `msqrt` will produce the same value. For this example the property is not very important because the function `sqrt` is only invoked once. The post-condition fully constrains the output for any input, so the model is said to be fully specified.

```
harness void fastPrimalityCheck(int p){
  // all primes are of form 6k±1 except 2,3
  bit isPrime = false;
  if(p>??){
    isPrime = true;
    // repeat minimally with different holes
    // check divisibility by 2 or 3
    minrepeat{if(p%?? == 0) isPrime = false;}
    // minimize loop bound: l1(sqrt(l2(p)))/n
    int bnd = linexp(sqrt(linexp(p))) / ??;
    optimize(bnd,p);
    for(int i=??; i < bnd; ++i){
      minrepeat{
        if(p % linexp(i) == 0) isPrime = false;}
    }
  }
  assert isPrime==checkPrimalityLinear(p);
}
```

(a)

```
int msqrt(int i) models sqrt{
    int rv = sqrtuf(i);
    if(i<=0){
        assert rv == 0;
    }else{
       assert rv*rv <= i;
       assert (rv+1)*(rv+1)>i;
    }
    return rv;
}
```

(b)

Fig. 1. (a) A sketch harness using the `sqrt` function to find the fast primality check algorithm(en.wikipedia.org/wiki/Primality_test) that requires much lesser than $\sqrt{p}$ divisibility checks, (b) a model for the `sqrt` function encoding the square root property.

2.2 Example 2: Big integer multiplication

Models don't have to be fully specified; in many cases, only a handful of properties of a function are relevant to synthesize a piece of code. As an example, consider an application that requires big-integer multiplication; Section 5 describes a couple of such functions we have explored, one of which is taking derivatives of polynomials with big-integer coefficients. For all of these experiments, we were interested in synthesizing implementations that used the Karatsuba algorithm for big-integer multiplication, whose details we also wanted to synthesize as was done in [23].

Without models, solving the sketch requires reasoning about the main function and the big integer multiplication in tandem. However, to reason about the polynomial derivative function, we only need to know that multiplication is commutative, and the zero property of multiplication (a number times zero is equal to zero). We describe those properties in the model in Figure 2(b). Just like with `sqrt`, the model breaks the problem into two independent sub-problems, but there are important differences between the `msqrt` model and the `mmul` model. The `mmul` model is under-specified, so there may be many functions that satisfy it. This means that when synthesizing karatsuba, the constraint that the solution is represented by the model must be combined with additional constraints that ensure that it is indeed implementing big-integer multiplication. Also, the model uses `min` and `max` to canonicalize the input so that `mmul(a,b)` will call the

```
harness void main(int[n] x1, int[n] x2){
  ...
  t = mul(x1, x2);
}

int[n] mul(int[n] x1, int[n] x2){
  // karatsuba algorithm
    ...
}
```

(a)

```
int[n] mmul(int[n] x1, int[n] x2)
                    models mul{
        int[n] xa = min(x1, x2);
        int[n] xb = max(x1, x2);
        int[n] rv = muluf(xa, xb);
        if(x1 == 0 || x2==0){
          assert rv == 0;
        }
        return rv;
}
```

(b)

Fig. 2. (a) A sketch harness using the `mul` function that uses the karatsuba algorithm for multiplying two integers represented by integer arrays, and (b) a model for the `mul` function encoding the commutativity and zero properties.

uninterpreted function with the same arguments as `mmul(b,a)` and therefore the model will be commutative.

Both the multiplication and the square root model use the same basic mechanisms, but as we will see they interact very differently with the counterexample guided inductive synthesis algorithm Cegis. One of the challenges addressed by our work is to define new general algorithms that work efficiently for both fully specified and partially specified models with combinations of interpreted and uninterpreted functions.

3 Problem Definition

A sketch is a function with missing code fragments marked by placeholders; however, all sketches can be represented as parametrized functions $Sk[c](in)$ or simply $Sk(c, in)$, where the parameter c controls the choice of code fragments to use in place of the unknowns. The role of synthesizer is therefore to find a value of c such that for all inputs $in \in E$ in a given input space E, the assertions in the sketch will be satisfied. In some cases, we may also want to assert that the sketch is functionally equivalent to a separately provided specification. This definition comes from [25], and using this definition as a starting point, we can formalize our support for function models.

Figure 3(a) shows a canonical representation of the models supported by our system. A model $\mathcal{M}$ is defined to be a 3-tuple $\mathcal{M} \equiv (\alpha, f_u, \mathtt{P}_{\mathtt{model}})$, where α denotes the canonicalization function that canonicalizes the input to the model before passing it to the uninterpreted function, f_u denotes an uninterpreted function whose output `rv` is the output of the model, and $\mathtt{P}_{\mathtt{model}}(\mathtt{rv}, \mathtt{in}_{\mathtt{model}})$ denotes a predicate that establishes properties of the return value with respect to the input. The predicate $\mathtt{P}_{\mathtt{model}}$ encodes the function's post-condition. We assume that models do not have explicit preconditions as they can be added using additional if statements inside the $\mathtt{P}_{\mathtt{model}}$ predicate, and they do not add much to the formalism.

```
FModel(in_model) models f_orig{
  /* canonicalization function */
  x = α(in_model);
  /* uninterpreted function */
  rv = f_u(x);
  /* post-condition */
  assert P_model(rv, in_model);
  return rv;
}
```

(a)

```
harness void Main(in){
  c_1 = ??; // unknown control
  /* arbitrary computation */
  t_1 = h(in, c);
  /* original function call */
  t_2 = f_orig(t_1);
  /* sketch assertion */
  assert P_main(t_2, in, c);
}
  f_orig(in){
     c_2 = ??;
     ...
  }
```

(b)

Fig. 3. (a) A simple canonical model for a function $f_{\texttt{orig}}$, and (b) a sketch function `Main` using the function $f_{\texttt{orig}}$.

Example 1. For the `sqrt` function model in Figure 1, the uninterpreted function f_u is `sqrtuf`, the canonicalization function is the identity function $\alpha(i) = i$, and the predicate is $\mathsf{P}_{\texttt{model}} \equiv (i \le 0 \to \texttt{rv} = 0) \wedge (i > 0 \to \texttt{rv}^2 \le i \wedge (\texttt{rv}+1)^2 > i)$. For the big integer multiplication model in Figure 2, the uninterpreted function f_u is `muluf`, the canonicalization function is $\alpha(x_1, x_2) = (\texttt{min}(x_1, x_2), \texttt{max}(x_1, x_2))$, and the predicate is $\mathsf{P}_{\texttt{model}} \equiv (x_1 = 0 \vee x_2 = 0) \to rv = 0$.

We now formalize the problem in terms of a stylized sketch shown in Figure 3(b) which uses a function $f_{\texttt{orig}}$ for which a model will be provided. In general, sketches can have a large number of unknowns, but in our stylized function, we use the variable c to represent the set of unknown values that the synthesizer must discover. The function $h(\texttt{in}, c)$ represents an arbitrary computation on the inputs to the main function to generate the inputs to the model. The unknown values may flow to the $f_{\texttt{orig}}$ function, but as far as that function is concerned, they are just another input. The function $f_{\texttt{orig}}$ itself may have additional unknowns, but the model cannot. In the constraint formulas, we will use $f_{\texttt{orig}}(\texttt{in}, c)$ to denote the fact that the unknown values in $f_{\texttt{orig}}$ will also be discovered by the synthesizer. The correctness of the overall function is represented by a set of assertions which can be represented by a predicate $\mathsf{P}_{\texttt{main}}(t_2, \texttt{in}, c)$, which can be expanded to $\mathsf{P}_{\texttt{main}}(f_{\texttt{orig}}(h(\texttt{in}, c), c), \texttt{in}, c)$. Without loss of generality, we assume that the main function includes a single call to $f_{\texttt{orig}}$, but our actual implementation supports multiple calls to $f_{\texttt{orig}}$. Also, in real sketches, the assertions can be scattered throughout the function, but this stylized sketch function will illustrate all the key issues in supporting models.

Now, to compute the value of unknown parameter c from the sketch, we need to solve the following two constraints.

1. Correctness of main under the model *(Correctness constraint)*

$$\exists c_1 \forall \texttt{in} \forall f_u \; \mathsf{P}_{\texttt{model}}(\texttt{rv}, \texttt{in}_{\texttt{model}}) \to \mathsf{P}_{\texttt{main}}(\texttt{rv}, \texttt{in}, c_1) \tag{1}$$

where $\mathtt{in}_{\mathtt{model}} \equiv h(\mathtt{in}, c_1)$ and $\mathtt{rv} \equiv f_u(\alpha(\mathtt{in}_{\mathtt{model}}))$. The constraint establishes that we want to find unknowns c_1 to complete the sketch such that for any function f_u, if the function satisfies the assertions in the model, on a given input, then the assertions inside main will also be satisfied.

2. Adherence of the original function to the model *(Adherence constraint)*

$$\exists c_2 \exists f_u \forall x \ \mathtt{P}_{\mathtt{model}}(f_u(\alpha(x)), x) \wedge f_u(\alpha(x)) = f_{\mathtt{orig}}(x, c_2) \tag{2}$$

The constraint establishes that there exists a function f_u that satisfies the assertions on all valid inputs ($\alpha(x)$), and that has an equivalent input-output relationship with the original function $f_{\mathtt{orig}}$.

As the following theorem explains, finding c_1 and c_2 that satisfy the two constraints above is equivalent to finding a solution to the original sketch problem.

Theorem 1. *If both Correctness constraint (Eq. 1) and Adherence constraint (Eq. 2) are satisfied, then*

$$\exists c_1, c_2 . \forall in . \mathtt{P}_{main}(f_{orig}(h(\mathtt{in}, c_1), c_2), \mathtt{in}, c_1)$$

Proof. Since the Adherence constraint (Eq. 2) is satisfied, we can use the second conjunct $f_u(\alpha(x)) = f_{\mathtt{orig}}(x, c_2)$ to substitute $f_u(\alpha(x))$ with $f_{\mathtt{orig}}(x, c_2)$ in the first conjunct of Eq. 2 to obtain

$$\exists c_2 \forall x \ \mathtt{P}_{\mathtt{model}}(f_{\mathtt{orig}}(x, c_2), x) \tag{3}$$

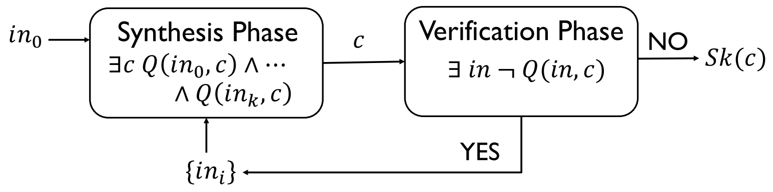
Since the Correctness constraint (Eq. 1) holds for all f_u, it should hold for the f_u in the solution of the Adherence constraint, and therefore $f_u(\alpha(\mathtt{in}_{\mathtt{model}}))$ can be substituted with $f_{\mathtt{orig}}(\mathtt{in}_{\mathtt{model}}, c_2)$ in Eq. 1 to obtain:

$$\exists c_1, c_2 \forall \mathtt{in} \ \mathtt{P}_{\mathtt{model}}(f_{\mathtt{orig}}(\mathtt{in}_{\mathtt{model}}, c_2), \mathtt{in}_{\mathtt{model}}) \rightarrow \mathtt{P}_{\mathtt{main}}(f_{\mathtt{orig}}(\mathtt{in}_{\mathtt{model}}, c_2), \mathtt{in}, c_1) \tag{4}$$

In the above constraint (Eq. 4), we know the left hand side of implication holds from Eq. 3, therefore the right hand side of implication should also hold, i.e. $\exists c_1, c_2 \forall \mathtt{in} \ \mathtt{P}_{\mathtt{main}}(f_{\mathtt{orig}}(\mathtt{in}_{\mathtt{model}}, c_2), \mathtt{in}, c_1)$ holds where $\mathtt{in}_{\mathtt{model}} \equiv h(\mathtt{in}, c_1)$.

4 Solving Correctness and Adherence Constraints

In previous work [22], we have used a counterexample guided inductive synthesis (CEGIS) approach to solve the doubly quantified constraints that arise in synthesis. Given a constraint of the form $\exists c. \ \forall \mathtt{in}. \ \mathtt{Q}(\mathtt{in}, c)$, CEGIS solves an inductive synthesis problem of the form $\exists c. \ \mathtt{Q}(\mathtt{in}_0, c) \wedge \mathtt{Q}(\mathtt{in}_1, c) \cdots \wedge \mathtt{Q}(\mathtt{in}_k, c)$, where $\{\mathtt{in}_0 \cdots, \mathtt{in}_k\}$ is a small set of representative inputs. If the equation above is unsatisfiable, the original equation will be unsatisfiable as well. If the equation provides a solution, we can check that solution by solving the following equation $\exists \mathtt{in} \ \neg \mathtt{Q}(\mathtt{in}, c)$. The algorithm, shown in Figure 4, consists of two phases: synthesis phase and verification phase. The algorithm first starts with a random assignment of inputs $\mathtt{in}_0$ and solves for the constraint $\exists c \ \mathtt{Q}(\mathtt{in}_0, c)$. If no solution exists, then it reports that the sketch can not be synthesized. Otherwise, it passes on the solution c to the verification phase to check if the solution works for all inputs using the constraint $\exists$`in` $\neg$`Q(in,` c`)`. If the verifier can't find a counterexample input, then the sketch $Sk(c)$ is returned as the desired solution. Otherwise, the verifier finds a counterexample input `in`$_1$ which is then added to the synthesis phase. The synthesis phase now solves for the constraint $\exists c$ `Q(in`$_0$`,` c`)` $\wedge$ `Q(in`$_1$`,` c`)`. This loop between the synthesis and verification phases continues until either the synthesis or the verification constraint becomes unsatisfiable. The algorithm returns "no solution" when the synthesis constraint becomes unsatisfiable whereas it returns the sketch solution when the verification constraint becomes unsatisfiable.



in_0
Synthesis Phase
$\exists c\ Q(in_0, c) \wedge \cdots \wedge Q(in_k, c)$
c
Verification Phase
$\exists\ in \neg\ Q(in, c)$
NO
$Sk(c)$
YES
$\{in_i\}$

Fig. 4. The CounterExample Guided Inductive Synthesis Algorithm (Cegis).

4.1 Limitations of Cegis for the Correctness constraint

We can apply the same Cegis approach to solve the Correctness constraint in Eq. 1, but as several authors [9, 31] have pointed out, the Cegis algorithm tends to perform poorly when there are strong assumptions in the sketch that depend on the values of the unknown control parameters. The problem is that when the verifier finds a counterexample, it is relatively easy for the inductive synthesizer to avoid the problem by changing the assumptions rather than by correcting the problem. To illustrate this issue, consider the example in Figure 5.

```
harness void main(in){
    int j = in + c1;
    int t = msqrt(j*j);
    assert t == in + c2;
}
```

Fig. 5. A simple msqrt example.

The example is artificial, but it illustrates an effect that happens in the primality check example as well. The example has two unknown integer values, c_1 and c_2, and one can easily see that as long as c_1 and c_2 are equal, the program will be correct. In this case, a counterexample from the verifier would include the value of `in`, as well as a function `sqrtuf`. Now, suppose that the Cegis algorithm starts with an initial guess of $c_1 = 3$ and $c_2 = 6$. The verifier can immediately produce the following counterexample: `in` $= 2,$ `sqrtuf` $= (25 \to 5,$ `else` $\to 7)$. The function `sqrtuf` in the counterexample evaluates to 5 when the input is 25, and to 7 otherwise. In this case, the strong sketch assumption (the model assertion) $t{*}t \leq (in{+}c_1){*}(in{+}c_1) \wedge (t{+}1){*}(t{+}1) > (in{+}c_1){*}(in{+}c_1)$ depends on the value of the control parameter c_1. The problem now is that for any value of $c_1 \neq 3$, the `sqrtuf` function in the counterexample will fail the model assertions. Say the synthesizer picked a value $c_1 = 4$, then the model assertion $\mathtt{P}_{\mathtt{model}} \equiv 7 * 7 \leq (2+4) * (2+4)$ becomes false. The synthesizer can easily pick a value for $c_1 \neq 3$ that makes the model assertions $\mathtt{P}_{\mathtt{model}}$ false, and therefore vacuously satisfy the correctness condition $\mathtt{P}_{\mathtt{model}} \rightarrow \mathtt{P}_{\mathtt{main}}$. Therefore, the CEGIS loop needs to perform $O(2^n)$ iterations before converging to the desired solution, where n is bound on the number of bits in the input integer `in`.

A previously proposed solution to this problem has been to identify that some inputs are actually dependent on other inputs, and should therefore not be a part of the counterexample, but instead the values of the dependent inputs should be existentially quantified [9]; *i.e.* they should be chosen *angelically* to use the terminology of [**?**,**?**]. In this case, for example, that approach would suggest that the `sqrtuf` function should not be part of the counterexample, since it is fully determined by the assertions in the model and the values of `in` and c_1. Following this approach, the inductive synthesis problem would then be

$$\exists c, f_{u_0}, \cdots, f_{u_k}.\ \mathtt{Q}(c, \mathtt{in}_0, f_{u_0}) \wedge \cdots \wedge \mathtt{Q}(c, \mathtt{in}_k, f_{u_k}) \tag{5}$$

where $\mathtt{Q}(c, \mathtt{in}, f_u) \equiv \mathtt{P}_{\mathtt{model}}(f_u(\alpha(h(\mathtt{in}, c))), h(\mathtt{in}, c)) \wedge \mathtt{P}_{\mathtt{main}}(f_u(\alpha(h(\mathtt{in}, c))), \mathtt{in}, c)$. Note that here we have replaced the implication in the correctness constraint with the conjunction, enforcing that the angelically selected f_{u_i} values always satisfy the model assertions $\mathtt{P}_{\mathtt{model}}$. The function f_u is no longer part of the counterexample, since it is fully defined by the assertions from the values of input `in` and unknown control c. This approach has an implicit assumption that there exist functions f_{u_i} that satisfies $\mathtt{P}_{\mathtt{model}}$ for corresponding inputs.

However, a big problem with this approach is that it may fail to converge in some cases. This happens in cases when the predicates in the model do not fully constrain the function, or they constrain it fully on only some of the inputs. For example, this is the case with the big integer multiplication example, where the predicate only constrains the function when one of the inputs is zero. Consider the scenario where for a given value of unknown c_i and an input $\mathtt{in}_i$, there are two functions f_u and f'_u that both satisfy the model assertions $\mathtt{P}_{\mathtt{model}}$, but only f_u satisfies the `main` assertions $\mathtt{P}_{\mathtt{main}}$, i.e. $\mathtt{Q}(c_i, \mathtt{in}_i, f_u)$ is satisfiable whereas $\mathtt{Q}(c_i, \mathtt{in}_i, f'_u)$ is unsatisfiable. Since the synthesizer is solving the existential (angelic) problem in Eq. 5, it will satisfy the equation by selecting the function f_u for $\mathtt{in}_i$ and c_i. However, the verifier needs to ensure that the completed sketch satisfies the correctness predicate for all functions f_u, and it will produce a counterexample $(\mathtt{in}_i, f'_u)$ for the given control value c_i. Since the synthesizer ignores the function f'_u of the counterexample and only considers the input $\mathtt{in}_i$, this algorithm as a result goes into an infinite loop and does not converge.

4.2 Our algorithm CEGIS+

Our algorithm CEGIS+ combines the benefits of the angelic approach while also ensuring that it converges in all cases. For the inductive synthesis phase, we use the following constraint:

Definition 1 (Inductive synthesis constraint).

$$\exists c, f_{u_0}, \cdots, f_{u_k} \quad \bigwedge_{(\mathtt{in}_i, f_{u_i}^{\mathsf{cex}})} \begin{array}{l} \mathtt{let}\ y = h(\mathtt{in}_i, c), x = \alpha(y)\ \mathtt{in} \\ \mathtt{let}\ t = \mathtt{ite}(\mathtt{P}_{\mathtt{model}}(f_{u_i}^{\mathsf{cex}}(x), y), f_{u_i}^{\mathsf{cex}}(x), f_{u_i}(x))\ \mathtt{in} \\ \mathtt{P}_{\mathtt{model}}(t, y) \ \wedge \ \mathtt{P}_{\mathtt{main}}(t, \mathtt{in}_i, c) \end{array}$$

where $\mathtt{ite}(c, a, b)$ *is the standard if-then-else function such that* $\mathtt{ite}(\mathtt{true}, a, b) = a$ *and* $\mathtt{ite}(\mathtt{false}, a, b) = b$*. The functions* $f_{u_i}^{\mathsf{cex}}$ *are obtained from the verifier counterexamples, and the functions* f_{u_i} *are determined angelically.*

The key idea behind this approach is that if the function from the counterexample satisfies the model assertions in the synthesis phase, then the synthesis constraint will use the counterexample function in the model. This will often be the case when the assertions in the model are under-constrained (weak), as is the case in the big integer multiplication example. On the other hand, if the assertions in the model are strong, as is the case with the square root model, the counterexample function will be ignored, and instead the synthesizer will use one of the angelically determined functions. The verification phase still solves the same correctness constraint in Eq. 1. The soundness of our algorithm follows from the soundness of the CEGIS and angelic algorithms. We now show that CEGIS+ algorithm always converges.

Theorem 2. *Assuming there exists a function* f_u *that satisfies the model assertions for all inputs, the* CEGIS+ *algorithm is guaranteed to converge to the solution of correctness constraint in Eq. 1.*

Proof. Since all sketches are solved with a bounded size on inputs, the set of possible counterexamples $(\mathtt{in}, f_u^{\mathsf{cex}}) \in \mathcal{IN} \times \mathcal{F}_{\mathcal{U}}$ is bounded where input `in` and function f_u^{cex} take values from the finite sets $\mathcal{IN}$ and $\mathcal{F}_{\mathcal{U}}$ respectively. In each iteration of the CEGIS+ algorithm a new counterexample $(\mathtt{in}, f_u^{\mathsf{cex}})$ is added. The only case for the algorithm to iterate forever is when the inductive synthesizer can produce a c that fails for one of the previously found counterexamples $(\mathtt{in}_i, f_{u_i}^{\mathsf{cex}})$ (i.e. it ignores the $f_{u_i}^{\mathsf{cex}}$ value and selects the angelic value f_{u_i} instead) and the verifier generates the counterexample $(\mathtt{in}_i, f_{u_i}^{\mathsf{cex}})$. This can't happen, because our synthesis constraint only ignores the counterexample function when the model assertion $\mathtt{P}_{\mathtt{model}}(f_{u_i}^{\mathsf{cex}}(x), y)$ becomes `false` and therefore $(\mathtt{in}_i, f_{u_i}^{\mathsf{cex}})$ cannot be a valid counterexample as the implication $\mathtt{P}_{\mathtt{model}} \rightarrow \mathtt{P}_{\mathtt{main}}$ will be true vacuously.

4.3 Solving the Adherence Constraint

Once we know that the main function is correct under the model, we need to show that the original function actually matches the behavior promised by the model. The adherence of model to the original function can be established by the following constraint:

$$\exists c\ \exists f_u\ \forall x.\ \mathtt{P}_{\mathtt{model}}(f_u(\alpha(x)), x) \wedge f_u(\alpha(x)) = f_{\mathtt{orig}}(x, c) \tag{6}$$

This constraint looks similar to the standard doubly quantified constraint usually solved by CEGIS, but one crucial difference is that it contains an existentially quantified function. The Z3 SMT solver [31] uses two main approaches to get rid of these kinds of uninterpreted functions; one is to treat assignments of the form $\forall x. f(x) = t[x]$ as macros and rewrite all occurrences of $f(x)$ to $t[x]$ in the formula. We can use this technique to eliminate f_u from the first part of the equation above, but the equality $f_u(\alpha(x)) = f_{\mathtt{orig}}(x, c)$ cannot in general be treated as a macro because of the presence of α. Another approach used by Z3 and inspired by Sketch is to ask the user to provide a template for f_u that allows it to do existential quantification exclusively over values instead of over functions. However, in our case we can do a lot better than that by taking advantage of the specific form of the constraints and the fact that we do not actually care about what f_u is; we only care to know that it exists.

We can efficiently solve the Adherence constraint in Eq. 6 by instead solving the following equivalent constraint:

$$\exists c \forall x \ \mathsf{P}_{\mathtt{model}}(f_{\mathtt{orig}}(x, c), x) \ \wedge \ \forall x_1, x_2 \ \alpha(x_1) = \alpha(x_2) \rightarrow f_{\mathtt{orig}}(x_1, c) = f_{\mathtt{orig}}(x_2, c) \tag{7}$$

The constraint states that the original function should satisfy the model constraints for all input values x, and if two inputs x_1 and x_2 cannot be distinguished by the canonicalization function α then the original function $f_{\mathtt{orig}}$ should also produce the same outputs on the two inputs. This equation does not involve any uninterpreted functions of any kind, and can be solved efficiently by the standard CEGIS algorithm because the left hand side of the implication does not depend on the unknown values c.

Theorem 3. *The constraint in Eq. 7 is equivalent to the Adherence constraint in Eq. 6.*

Proof. It is easy to see that the Adherence constraint (Eq. 6) implies the constraint in Eq. 7. If $f_{\mathtt{orig}}$ does not satisfy the assertions in the model, then it can not be equal to $f_u(\alpha(x))$. Also, if there are two inputs that cannot be distinguished by α, but for which $f_{\mathtt{orig}}$ produces different outputs, then it would not be possible to find an f_u such that $f_u(\alpha(x))$ equals $f_{\mathtt{orig}}$.

The converse is a little trickier. We have to show that if Eq. 7 is satisfied, then the Adherence constraint will be satisfied as well. The key is to show that for a given value of c if $\forall x_1, x_2.\ \alpha(x_1) = \alpha(x_2) \rightarrow f_{\mathtt{orig}}(x_1, c) = f_{\mathtt{orig}}(x_2, c)$, then $\exists f_u \forall x \ f_u(\alpha(x)) = f_{\mathtt{orig}}(x, c)$. Let $f_u(t)$ be a function computed as follows:

$$f_u(t) = \begin{cases} f_{\mathtt{orig}}(x_1, c) & \text{if } \exists x_1.\ \alpha(x_1) = t \\ 0 & \text{Otherwise} \end{cases}$$

Now, we have to show that such an f_u is well defined and satisfies $\forall x f_u(\alpha(x)) = f_{\mathtt{orig}}(x, c)$. Consider two values x_1 and x_2 such that $\alpha(x_1) = \alpha(x_2) = t$, then $\alpha(x_1) = \alpha(x_2) \rightarrow f_{\mathtt{orig}}(x_1, c) = f_{\mathtt{orig}}(x_2, c)$ gives us $f_{\mathtt{orig}}(x_1, c) = f_{\mathtt{orig}}(x_2, c)$. So the function $f_u(t)$ returns the same value for both x_1 and x_2 and is therefore well defined. The function f_u satisfies the constraint $\forall x f_u(\alpha(x)) = f_{\mathtt{orig}}(x, c)$ by definition.

A final point to note is that if a function $f_{\texttt{orig}}$ satisfies the Adherence constraint for a given model, then it must be true that there exists an f_u such that the model satisfies its assertions for all inputs, which was one of the assumptions of the algorithm to solve the Correctness constraint.

5 Evaluation

We now present the evaluation of our algorithms on a set of Sketch benchmark problems. All these benchmark problems consists of sketches that use complex functions such as integer square root, big integer multiplication, sorting (array, topological) etc. In our evaluation, we run each benchmark problem for 20 runs and we present the median values for the running times and the number of iterations of the synthesis-verification loop. The experiments were run (for parallelization) on virtual machines with physical cores using Intel Xeon L5640 2.27GHz processors, each virtual machine comprising of 4 virtual CPUs (2 physical cores) and 16 GB of RAM.

5.1 Implementation and Benchmarks

We have implemented our algorithms for solving the Correctness and Adherence constraints on top of the open-source Sketch solver. Our benchmark problems can be found on the Sketch server[1]. A brief description of the set of sketch benchmarks that we use for our evaluation is given below.

– `calc-toposort`: A function for evaluating a Boolean DAG using topological sort function. A more detailed case study is presented in Section 6 for this benchmark.
– `bsearch-sort`: A binary search algorithm to find an element in an array that uses the sort function.
– `gcd-n-nums`: An algorithm to compute the gcd of n numbers that uses the gcd function.
– `lcm-n-nums`: An algorithm to compute the lcm of n numbers that uses the lcm function.
– `matrix-exp`: An algorithm to compute matrix exponentiation using the matrix multiplication function.
– `polyderiv-mult`: An algorithm to compute the derivative of a polynomial whose coefficients are represented using big integer representation and that uses karatsuba multiplication.
– `polyeval-mult-exp`: An algorithm to compute the value of a polynomial on a given value that uses the karatsuba multiplication and exponentiation functions.
– `power-root-sqrt`: An algorithm to compute the 2^kth integer root of a number using the integer square root function.
– `primality-sqrt`: An algorithm to check if a number is prime that uses the integer square root function.

[1] http://sketch1.csail.mit.edu/Dropbox/models/experiments/

Experimental Setup All our benchmarks include a larger `main` function sketch which calls another function $f_{\texttt{orig}}$ which we would like to model and perform modular synthesis efficiently. In most of the cases, the inner function $f_{\texttt{orig}}$ is a sketch which comes with an imperative specification `f-spec` and a declarative model `f-model`. We perform our experiments based on the strength of the models:

1. If `f-model` enforces strong constraints (fully specifying the function, e.g. the `sqrt` model) then we use `f-model` to synthesize both `main` and $f_{\texttt{orig}}$. We compare this with synthesis of $f_{\texttt{orig}}$ with the imperative `f-spec` and then using `f-spec` or the synthesized $f_{\texttt{orig}}$ function to synthesize `main`.
2. If `f-model` enforces weak constraints (partially specifying the function e.g. the `mult` model) then we will have to fallback to synthesizing $f_{\texttt{orig}}$ using `f-spec` in any case. So, we don't show the time for this synthesis process and simply compare the median times for synthesis of `main` using `f-model`, `f-spec` or synthesized $f_{\texttt{orig}}$.

5.2 Scaling Sketch solving using Models

We first show the need of using function models for solving large complex sketches. The last columns in Table 1 and Table 2 show the time required by the SKETCH solver to synthesize the `main` function using the synthesized code for inner function $f_{\texttt{orig}}$, which involves solving two sketch harnesses. As we can see from the tables, most of these sketches either timeout because of overshooting the memory limits or by going over the timeout limit, which we set to 15 minutes for all the benchmark runs except the `calc-toposort` benchmark for which we use a 5 hour limit. We observed that even when we let these sketches run for a longer time, they often run out of memory and do not terminate. Other alternative options to solve such complex sketch problems is to synthesize the function $f_{\texttt{orig}}$ using its imperative specification `f-spec`, and then synthesize the harness function using `f-spec`. The middle column(s) in Table 1 and Table 2 (`f-spec`) report the time taken to synthesize the `main` function using `f-spec`. We observe that this cleaner separation allows some of the harness functions to be synthesized, but it typically takes a very long time. Some of these benchmarks do not terminate when we use more complex functions, e.g. when we use merge sort (instead of bubble sort) for the sorting function. The first columns in Table 1 and Table 2 (Using `f-model`) show the results of using function models for solving `main` or both of these sketches. We observe a big improvement in synthesis times of sketches for cases in which they depend on only some partial property of $f_{\texttt{orig}}$ and in cases where the models are exponentially succinct. For example, for the `matrix-exp` benchmark, the sketch harness only needs to know the exponentiation property of multiplication. For some benchmarks such as `primality-sqrt`, the synthesis times are quite similar to the synthesis time of the second approach because in this case the function model expresses the complete property of the sqrt function, and the constraints generated by the model are almost equal in size to the constraints generated by the linear square root search. We note that in all the benchmarks, the model based solving is always faster than the chained synthesis (Synthesizing `main` using synthesized $f_{\texttt{orig}}$) and in most cases, it is also better than or as good as using the imperative specification `f-spec`.

Benchmark	**Solving Time (in s) for Synthesis of `main`**									
	Using `f-model`				**Using `f-spec`**			**Using synthesized $f_{\texttt{orig}}$**		
	`main`	$f_{\texttt{orig}}$	**Adh.**	**Total**	`main`	$f_{\texttt{orig}}$	**Total**	`main`	$f_{\texttt{orig}}$	**Total**
`calc-toposort`	246.5	-	1974.6*	**2221**	×	-	×	×	-	×
`gcd-n-nums`	2.4	7.1	0.4	**9.9**	8.4	11.7	20	1.7	11.7	13.4
`lcm-n-nums`	1.5	17.2	0.4	**19.2**	×	30.2	×	×	30.2	×
`power-root-sqrt`	1.04	93.7	0.3	**95.1**	×	57	×	×	57	×
`primality-sqrt`	438.1	64.9	0.3	503.4	302.9	37.8	**340.7**	×	37.8	×

Table 1. The sketch solving times for three approaches in the presence of a strong model: i) using models (`f-model`) ii) using imperative specification `f-spec`, and iii) Synthesis of $f_{\texttt{orig}}$ using `f-spec` and `main` using synthesized $f_{\texttt{orig}}$. The × values in the table entries denote timeout (> 15 mins), *timeout for calc-toposort set to 5 hours.

Benchmark	**Solving Time (in s) for Synthesis of `main` using**				
	`f-model` + Adherence			**`f-spec`**	**synthesized $f_{\texttt{orig}}$**
`bsearch-sort`	8.45	0.87	**9.32**	29.2	83.265
`matrix-exp`	17.14	64.2	**81.34**	×	×
`polyderiv-mult`	5.61	0.9	**6.51**	12.601	×
`polyeval-mult-exp`	2.6	0.9	**3.5**	8.657	10.644

Table 2. The sketch solving times for three approaches in the presence of a weak model: i) using models (`f-model`) with adherence check, ii) using imperative specification `f-spec`, and iii) synthesis of `main` using synthesized $f_{\texttt{orig}}$. The × values in the table entries denote time-out (> 15 mins).

5.3 Comparison with CEGIS and Angelic Synthesis

In this experiment, we compare the performance of our Cegis+ algorithm with that of Cegis and the angelic synthesis algorithm on two metrics: 1) the solving time and 2) the number of synthesis-verification iterations. We expect the Angelic algorithm to perform poorly on benchmarks where the function models are under-constrained and similarly we expect the Cegis algorithm to perform poorly on benchmarks that are over-constrained. Figure 6 shows the logarithmic graph of running times of the three algorithms on our benchmarks. As expected, we see two benchmarks where the Angelic algorithm times out (set to 15 minutes) whereas the Cegis algorithm times out on two different benchmarks. The Cegis+ algorithm solves each of the problem within 440 seconds each and in general has a faster or comparable performance on problems where other algorithms don't timeout. Figure 7 shows the logarithmic graph of the number of synthesis-verification iterations performed by each one of the algorithms on the

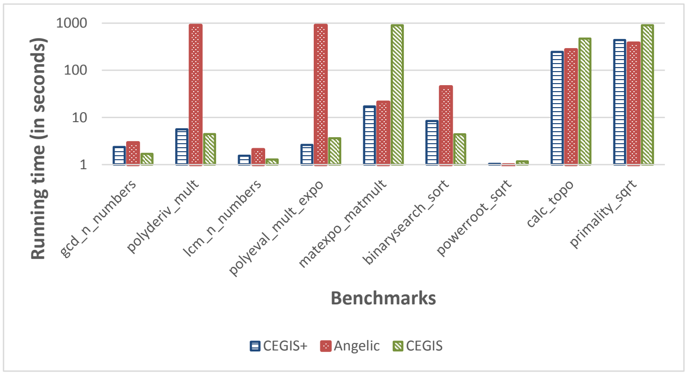


Fig. 6. The solving times of the three algorithms: Cegis+, Angelic, and Cegis on the benchmark problems.

benchmark problems. The Cegis+ algorithm performs lesser number of iterations than the Cegis algorithm for all benchmarks and performs lesser iterations than the Angelic algorithm on all but two benchmarks.

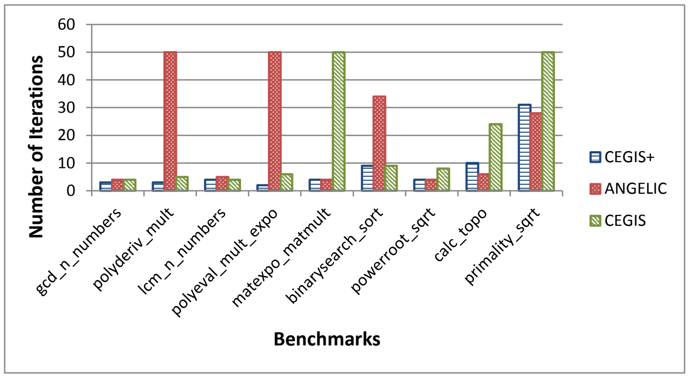


Fig. 7. The number of synthesis-verification iterations performed by the Cegis+, Angelic, and Cegis algorithms on the benchmark problems.

6 Case Study: Boolean DAG Calculator

We present a case study of using function models for synthesizing a calculator that interprets a circuit representing a Boolean DAG (directed acyclic graph). As shown in Figure 8, the interpreter has two main components: a *calculator* and a *parser*, with auxiliary functions like `cmain` that just calls `calc` and `parse`, and `test` that is the test harness.

```
int[n] mtopo(int n, int[2][n] parent)
      models toposort {
  int[n] sorted = topo_uf(n, parent);
  for (int i=0; i<n; i++) {
    int u = sorted[i];
 // node id in sorted must be valid
    assert u>=0 && u<n;
    for (int j=0; j<=i; j++) {
      int v = sorted[j];
 // sorted contains no duplicated node ids
      if (i<j) { assert u != v; }
 // if u occurs after v, u cannot be v's parent
      assert u != parent[v][0] && u != parent[v][1];
    }
  }
}

bit[n] calc(int n, int[2][n] parent, int[n] opr) {
 int[n] sorted = toposort(n, parent);
 bit[n] result;

 for (int i=0; i<n; i++) {
  int u = sorted[i];
  minrepeat { if (opr[u] == ??) {
    result[u] = {| ?? | !result[parent[u][0]] |
      result[parent[u][0]] || result[parent[u][1]]|
      result[parent[u][0]] && result[parent[u][1]]|};
  } }
 }
 return result;
}
```

```
int NOT = 2, OR = 3, AND = 4;

void parse(int n, int[3][n] input,
   ref int[2][n] parent, ref int[n] opr){
// input is an array of 3-tuples
//  of the form {Operator,Src1,Src2}.
// parse converts it to separate parent
//  and opr, and sets unused parent[u][j]
// to -1. parse is also synthesized, using
//  minrepeat, holes, and unknown choices.
}

bit[n] cmain(int n, int[3][n] input){
  int[2][n] parent;
  int[n] opr;
  parse(n, input, parent, opr);
  return calc(n, parent, opr);
}

harness void test(int n, int[3][n] input){
  // test for execution safety
  cmain(n, input);

  // test for functional correctness
  assert cmain(5,{{1},{0},{AND,1,3},
    {OR,0,4},{NOT,1}}) == {1,0,0,1,1};
  // a few more test cases, omitted here
}
```

Fig. 8. The sketch for the Boolean DAG calculator.

The `calc` function takes as input a DAG that defines a Boolean circuit, and calculates the value of every Boolean node: the DAG is represented in an *internal representation* consisting of an array `opr` that defines the Boolean operator at each node (we encode the CONST 0 and 1, NOT, OR, and AND operators using integer values), and an array `parent` that denotes the source operands of each node's operator, i.e. `parent[u][j]` stores the node id of node `u`'s `j`-th operand. We assume at most 2 operands for each operator for simplicity and in the case where a node `u` has fewer than 2 operands, some `parent[u][j]` will be set to -1. The output of `calc` is `result`, a bitvector of size `n`, where `result[u]` stores the calculated value of node `u`.

We first need to get a topological order of the DAG to calculate the node values. The simplest imperative `toposort` function (omitted here) is too complex for the solver to reason about, but the declarative model of `toposort` (`mtopo`) is simple and solver-friendly. The main part of `calc` is for calculating each node's value according to the node operator and is based on the previously calculated node values. This calculation is usually performed using a "big switch" (or several `if` statements). The cases for different operators are very similar: depending on the operator, fetch the values of different number (can be 0) of parents, and calculate the result, which are tedious to write. Here `calc` relies on synthesis to reduce this burden (see the `minrepeat` block): it abstracts the common structure of all cases and leaves the differences to unknown constant choices, which are solved by the synthesizer. The use of synthesis also allows the function to adapt to small changes in its requirements. For example, if the programmer decides to no longer support OR because it is redundant with AND and NOT, the synthesizer can adjust the function accordingly without the need to modify the code. Similarly, new operators can be added or encoding of existing operators can be modified just as easily.

The `parse` function takes as input the more readable format of the DAG (where the operator and operands for each node are grouped together as a 3-tuple), and converts it to the internal representation used by `calc`. It needs to copy the right number of operands from `input` to `parent`, and set the remaining `parent` values to -1 depending on the kind of operator, which we specify as choices to be synthesized. The body of `parse` (omitted here) is also sketched with unknown choices to solve similar to `calc`.

An interesting question in this case is how to provide a specification. The `test` function is a harness testing two aspects of the program: execution safety (for any input the program should execute without any assertion failure, array out of bounds error, or reading uninitialized value error) and functional correctness (for a set of known inputs the program should produce the known correct outputs). The two aspects together are sufficient for the Sketch solver to determine all the unknown constants.

As we can see from Table 1 and Figure 6, the use of function models enable the synthesis for this complex program. Without the model, the solver timed out after 5 hours and couldn't synthesize the program; whereas with the model, it solves the program in about 5 minutes (less than 40 minutes even after adding the adherence checking time). We can also see that the CEGIS+ algorithm is much faster than CEGIS because the inputs to the function model `mtopo` are significantly influenced by the unknown holes in `parse`, and CEGIS+ performs slightly better than the pure angelic model.

7 Related Work

The idea of using function models for synthesis is very related to the work on component-based synthesis and is inspired from modular reasoning techniques used in verification. The work on efficiently solving QBF (Quantified Boolean Formulas) is also related to our technique of solving Adherence constraints. We briefly describe some of the related work in each of these areas.

Component-based Synthesis: The work on component-based synthesis considers the problem of synthesizing larger systems using components as building blocks, which is a central motive for our work of introducing function models in SKETCH. The closest related work to ours is that of synthesizing loop free programs using a library of components [9]. This work assumes that all library components have complete logical specifications and it employs a constraint-based synthesis algorithm similar to the angelic algorithm for solving the Correctness constraint in the synthesis phase. Recently this approach has been applied for synthesizing efficient SIMD implementation of performance critical loops [1]. As we have observed for many benchmark problems, often times a partial specification of the library component suffices for synthesizing the correct client code. In the presence of partial function specifications (under-constrained specifications), the angelic algorithm may not converge and is inefficient, whereas the CEGIS+ algorithm efficiently converges to the solution for both partially-specified and fully-specified function models. The work on LTL synthesis from libraries of reusable components [13] assumes that the components are specified in the form of transducers (finite state machines with outputs). Our work, on the other hand, considers the problem of functional synthesis and uses constraint-based synthesis algorithms.

Efficient QBF solving: Efficient solving of Quantified Boolean Formulas (QBF) has been a big research challenge for a long time and the constraints generated by SKETCH are too large for current state-of-the-art QBF solver to handle [22]. Recently, word-level simplifications (inspired from automated theorem proving and model finding techniques based on sketches) have been proposed to handle quantified bit-vector formulas in an SMT solver [31]. We can also use this technique to solve the Adherence constraint, but it would require us to provide function templates for the unknown uninterpreted function. Our reduction allows the CEGIS algorithm to efficiently solve the constraint without the need of a function template.

Compositional Verification: The idea of using function models for synthesis is inspired from modular verification techniques used for model checking [6]. This idea of modular reasoning using pre-conditions and post-conditions of functions is widely used today in many verification tools such as DAFNY [12] and MAGIC [5] to enable verification of large complex systems. These Assume-guarantee reasoning based techniques applies the divide-and-conquer approach to reduce the problem of analyzing the whole system into verification of individual components [16, 28]. For verifying individual components, it uses assumptions to capture the context the component makes about its environment and uses guarantees as properties that will hold after the component execution. It then composes the assumptions and guarantees to prove properties about the whole system. Our function models apply these ideas in the context of software synthesis.

Program Synthesis: Program synthesis has been an intriguing research question from a long time back [14, 15]. With the recent advances in SAT/SMT solvers and computational power, the area of program synthesis is gaining a renewed interest. It has been used successfully in various domains such as synthesizing efficient low-level code [24], data-structures [21], string transformations [7, 8], table lookup transformations [18] and number transformations [19] from input-output examples, implicit declarative computations in Scala [11], graph algorithms [10], multicore cache coherence protocols [29], automated grading of programming assignments [20], automated inference of synchronization in concurrent programs [30], and for solving games on infinite graphs [2]. We believe our technique can complement the approaches used in many of these domains.

8 Conclusion

In this paper, we presented a technique to perform modular synthesis in Sketch using function models. This technique enables solving of sketches when they call complex functions and when the correctness of the main harness function depends on a partially interpreted version of the complex function (which we call models). We show that both the Cegis and the angelic algorithm are inefficient and potentially incomplete, and we present a complete and terminating algorithm to efficiently solve sketches with all kinds of function models. On the basis of promising preliminary results, we believe that this technique will prove very useful in using Sketch for synthesizing complex and large synthesis problems.

9 Acknowledgments

We thank Sanjit Seshia and anonymous reviewers for their valuable feedback.

References

1. G. Barthe, J. M. Crespo, S. Gulwani, C. Kunz, and M. Marron. From relational verification to simd loop synthesis. In *PPoPP*, 2013.
2. T. A. Beyene, S. Chaudhuri, C. Popeea, and A. Rybalchenko. A constraint-based approach to solving games on infinite graphs. In *POPL*, 2014. (To appear).
3. R. E. Bryant, S. K. Lahiri, and S. A. Seshia. Modeling and verifying systems using a logic of counter arithmetic with lambda expressions and uninterpreted functions. In *CAV*, pages 78–92, 2002.
4. J. R. Burch and D. L. Dill. Automatic verification of pipelined microprocessor control. In *CAV*, pages 68–80, 1994.
5. S. Chaki, E. M. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in c. In *ICSE*, pages 385–395, 2003.
6. O. Grumberg and D. E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16, 1991.
7. S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.
8. S. Gulwani, W. R. Harris, and R. Singh. Spreadsheet data manipulation using examples. In *CACM*, 2012.
9. S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.
10. S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, 2010.
11. V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. PLDI, 2010.
12. K. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, pages 348–370, 2010.
13. Y. Lustig and M. Y. Vardi. Synthesis from component libraries. In *FOSSACS*, pages 395–409, 2009.
14. Z. Manna and R. Waldinger. Synthesis: Dreams => programs. *IEEE Transactions on Software Engineering*, 5(4):294–328, 1979.

15. Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, 2(1):90–121, 1980.
16. K. L. McMillan. A compositional rule for hardware design refinement. In *CAV*, pages 24–35, 1997.
17. S. A. Seshia. Sciduction: combining induction, deduction, and structure for verification and synthesis. In *DAC*, pages 356–365, 2012.
18. R. Singh and S. Gulwani. Learning semantic string transformations from examples. *PVLDB*, 5, 2012.
19. R. Singh and S. Gulwani. Synthesizing number transformations from input-output examples. In *CAV*, 2012.
20. R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *PLDI*, 2013.
21. R. Singh and A. Solar-Lezama. Synthesizing data structure manipulations from storyboards. In *SIGSOFT FSE*, 2011.
22. A. Solar-Lezama. *Program Synthesis By Sketching.* PhD thesis, EECS Dept., UC Berkeley, 2008.
23. A. Solar-Lezama. Program sketching. *STTT*, 15(5-6), 2013.
24. A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.
25. A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, pages 404–415, 2006.
26. S. Srivastava, S. Gulwani, S. Chaudhuri, and J. S. Foster. Path-based inductive synthesis for program inversion. In *PLDI*, pages 492–503, 2011.
27. S. Srivastava, S. Gulwani, and J. Foster. From program verification to program synthesis. *POPL*, 2010.
28. E. W. Stark. A proof technique for rely/guarantee properties. In *FST & TCS85, LNCS 206*, pages 369–391, 1986.
29. A. Udupa, A. Raghavan, J. V. Deshmukh, S. Mador-Haim, M. M. K. Martin, and R. Alur. Transit: specifying protocols with concolic snippets. In *PLDI*, pages 287–296, 2013.
30. M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, New York, NY, USA, 2010. ACM.
31. C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. Efficiently solving quantified bit-vector formulas. *Formal Methods in System Design*, 42(1):3–23, 2013.