Synthesis of Loop-free Programs

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Susmit Jha *
University of California, Berkeley
jha@eecs.berkeley.edu

Ashish Tiwari †
SRI International, Menlo Park
tiwari@csl.sri.com

Ramarathnam Venkatesan
Microsoft Research, Redmond
venkie@microsoft.com

Abstract

We consider the problem of synthesizing loop-free programs that implement a desired functionality using components from a given library. Specifications of the desired functionality and the library components are provided as logical relations between their respective input and output variables. The library components can be used at most once, and hence the library is required to contain a reasonable overapproximation of the multiset of the components required.

We solve the above *component-based synthesis problem* using a constraint-based approach that involves first generating a *synthesis constraint*, and then solving the constraint. The synthesis constraint is a first-order $\exists\forall$ logic formula whose size is quadratic in the number of components. We present a novel algorithm for solving such constraints. Our algorithm is based on counterexample guided iterative synthesis paradigm and uses off-the-shelf SMT solvers.

We present experimental results that show that our tool `Brahma` can efficiently synthesize highly nontrivial 10-20 line loop-free bitvector programs. These programs represent a state space of approximately 20^{10} programs, and are beyond the reach of the other tools based on sketching and superoptimization.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Program Synthesis

General Terms Algorithms, Theory, Verification

Keywords Program Synthesis, Component-based Synthesis, SMT

1. Introduction

Composition has played a key role in enabling configurable and scalable *design and development* of efficient hardware as well as software systems. Hardware developers find it useful to design specialized hardware using some base components, such as adders and multiplexers, rather than having to design everything using universal gates at bit-level. Similarly, software developers prefer to use library features and frameworks. Composition has also played a key role in enabling scalable *verification* of systems that have been designed in a modular fashion. This involves verifying specifications of the base/constituent components in isolation, and then assuming these specifications to verify specification of the higher-level system made up of these components.

In this paper, we push the above-mentioned applications of composition to another dimension, that of synthesis. We focus on the *component-based synthesis* problem where the goal is to build a system by composing base components. Automating component-based synthesis is attractive for many reasons. First, the designed system is correct by construction, which obviates the need for verification. Second, the designed system can be guaranteed to be optimal in terms of using the fewest possible number of components. Third, automation improves developer's productivity, since finding the exact set of components and their correct composition can be a daunting task, especially when the base component library is huge.

We present a synthesis procedure that takes as input a specification for the desired program and specifications of the base components, and synthesizes a circuit-style composition – a straight-line program – using these base components. Our procedure is semi-automated as it relies on the user to provide one additional piece of information: an over-approximation of the number of times each base component is used in the desired program. Our synthesis procedure views the base component library as a multiset whose elements are treated as resources that can be used at most once. Hence, our procedure can be seen as solving a *resource-bounded component-based synthesis* problem. Our procedure is also restricted to discovering only straight-line programs. Note that loop-free control-flow can be encoded as a straight-line program by providing the ite (if-then-else) operator as a base component, but synthesizing loops is left for future work. The reliance on the user for making copies of the base components and for discovering loops is indicative that the true success of program synthesis may lie in an interactive process that combines higher-level insights of humans with computational capabilities of computers.

While our focus on synthesizing loop-free programs may appear narrow, it is nevertheless a significant goal in itself [12]. Techniques for synthesizing loop-free code will be needed when synthesizing more complex code patterns. Synthesis of loops requires techniques that are mostly orthogonal to the techniques for synthesizing straight-line code, and hence it is fruitful to study them separately [13]. Loop-free program synthesis has several independent applications too that have been pursued in different communities recently; such as, optimizing the core of many compute

* Work done while visiting SRI International.

† Research supported in part by NSF grants CNS-0720721, CSR-EHCS-0834810 and CSR-0917398.

PLDI'11, June 4–8, 2011, San Jose, California, USA.

intensive loops (superoptimizers) [28], synthesizing API call sequence (e.g., Jungloid [26]), bitvector algorithms, geometry constructions [18, 14], text-editing [24, 38] and table-manipulation programs [16]. Finally, despite its appearance, loop-free program synthesis is challenging as the search space of loop-free programs is still huge.

While we foresee several applications of component-based program synthesis, most of the examples in this paper relate to discovering intricate bitvector programs, which combine arithmetic and bitwise operations. Bitvector programs can be quite unintuitive and extremely difficult for average, or sometimes even expert, programmers to discover methodically. Consider, for example, the problem of turning-off the rightmost 1-bit in a bitvector x. This can be achieved by computing $x\&(x-1)$, which involves composing the bitwise $\&$ operator and the arithmetic subtraction operator in an unintuitive manner. In fact, the upcoming 4th volume of the classic series *art of computer programming* by Knuth has a special chapter on bitwise tricks and techniques [23]. In this paper, we demonstrate how to semi-automate the discovery of small, but intricate, bitvector programs using the currently available formal verification technology. This semi-automation can be exposed to users in at least two different ways. First, software development environments can provide this capability to help programmers write correct and efficient code. Alternatively, compilers can use the synthesis procedure to optimize implementations or make them more secure. Superoptimizers, for example, perform automatic translation of a given sequence of instructions into an optimal sequence of instructions for performing aggressive peephole optimizations [5] or binary translation [6]. Rather than achieve efficiency, the goal of such translation can also be to reduce vulnerability in software. For example, any piece of code that computes the average of two numbers, x and y, by evaluating $(x+y)/2$ is inherently flawed and vulnerable since it can overflow. However, using some bitwise tricks, the average can be computed without overflowing (e.g., $(x|y) - ((x \oplus y) >> 1)$). Compilers can replace vulnerable snippets of code by the discovered equivalent secure code.

The number of straight-line programs that can be constructed using a given set of base library components is exponential in the number of base components. Rather than performing a naive exponential search for finding the correct program, our synthesis algorithm relegates all exponential reasoning to tools that have been engineered for efficiently performing exponential search, namely the Satisfiability (SAT) and Satisfiability Modulo Theory (SMT) solvers[1]. SMT solvers use intelligent backtracking and learning to overcome the complexity barrier. SMT solvers can be used to verify that a given (loop-free) system meets a given specification. In this paper, we show how to use the same SMT solving technology to synthesize systems that meet a given specification.

Existing synthesis techniques based on superoptimizers [28, 11, 22] and sketching [32, 33] can also be used to solve the component-based synthesis problem. Superoptimizers explicitly perform an exponential search. Sketching solves a more general program synthesis problem, and is not designed for solving the component-based synthesis problem. When they are used to solve the component-based synthesis problem, both superoptimizers and sketching were empirically found to not scale. In contrast, our technique leaves the inherent exponential nature of the problem to the underlying SMT solver, whose engineering advances over the years have made them effective at solving the "usually-not-so-hard" instances that arise in practice, and hence end up not requiring exponential reasoning.

[1] SMT solving is an extension of SAT solving technology to work with theory facts, rather than just propositional facts. In fact, there is a SMT solving competition that is now held every year, and it has stimulated improvement in solver implementations [1].

Our assumption that we are given (by developers) logical specifications of the desired program may appear to be unrealistic. It is not. The logical specification can be given in the form of an (unoptimized) program, which are easy to write. In fact, we did exactly this in our experiments. In many domains, writing such logical specifications is much less time-consuming than discovering optimized straight-line code.

Our synthesis algorithm is based on a *constraint-based* approach that involves reducing the synthesis problem to that of solving a constraint. This involves the two key steps of constraint generation and constraint solving.

In the constraint generation phase, the synthesis problem is encoded as a constraint, referred to as *synthesis constraint*. Our synthesis constraint has two interesting aspects.

- The synthesis constraint is a first-order logic formula.
  The synthesis problem can be viewed as a generalization of the verification problem. It is well known that verification of a straight-line program can be reduced to proving validity of a first-order logic formula, and hence the synthesis problem can be reduced to finding satisfiability of a second-order logic formula. But, the non-trivial aspect of our encoding is that it generates a first-order logic formula. This is significant because off-the-shelf constraint solvers cannot effectively solve second-order formulas.
- The size of the synthesis constraint is quadratic in the number of components.
  One way to generate a first-order logic constraint would be to use the constraint generation methodology used inside the sketching technique, which is also a constraint-based technique. However, the size of the constraint generated by the sketching technique could potentially be exponential in the number of components. In contrast, our encoding yields a constraint that is guaranteed quadratic in the number of components.

In the constraint solving phase, we use a refined form of the classic counterexample guided iterative synthesis technique [10, 30] built on top of off-the-shelf SMT solvers. The synthesis constraint obtained from our encoding is an $\exists\forall$ formula, which cannot be effectively solved using off-the-shelf SMT solvers directly. The counterexample guided iterative synthesis technique involves choosing some initial set of test values for the ($\forall$) universally quantified variables and then solving for the ($\exists$) existentially quantified variables in the resulting constraint using SMT solvers. If the solution for the existentially quantified variables works for all choices of universally quantified variables, then a solution has been found. Else, a counterexample is discovered and the process is repeated after adding the counterexample to the set of test values for the universally quantified variables. We use this method, suitably adapted to handle *definitions* correctly, to solve our synthesis constraint.

We have implemented our constraint generation and constraint solving technique in a tool called `Brahma`. We have applied `Brahma` to several different examples. Majority of the examples come from the domain of bitvector program synthesis using a set of components that implement basic bitvector operations. These programs typically involve unintuitive composition of the bitvector operations, and are quite challenging to synthesize manually. `Brahma` is able to synthesize (equivalent variants of) a variety of bitvector programs picked up from a classic book [37] in time ranging from 1.0 to 2778.7 seconds. In contrast, the Sketch and AHA tools, based respectively on sketching and super-optimization, time-out on 9 and 12 of the 25 examples respectively where timeout was set to 3600 seconds. Sketch is slower by an average factor of 20 on the remaining examples.

Contributions and Organization

- We define the problem of component-based synthesis using a set of base components (Section 3).
- We present an encoding that reduces the synthesis problem to that of finding a satisfying assignment to a first-order logic constraint with quantifier alternation, whose size is at most quadratic in the number of base components. (Section 5).
- We present a novel technique for solving first-order logic constraints with quantifier alternation using off-the-shelf SMT solvers (Section 6).
- We apply our constraint generation and solving technique to a set of benchmark examples. We also experimentally compare our technique with other existing techniques, namely sketching and superoptimization, that can be used to synthesize bitvector programs (Section 7). Tools based on other techniques either perform order of magnitude slower or timeout and fail to yield a solution.

2. Running Example

First, we introduce a small example to give a high-level overview of our technique. We also use this example as a running example to illustrate several details of our technique in following sections.

Consider the task of designing a bitvector program that masks off the right-most significant 1-bit in an input bitvector. More formally, the bitvector program takes as input one bitvector I and outputs a bitvector O such that O is obtained from I by setting the right-most significant 1-bit in I to 0. For example, the bitvector program should transform the bitvector 01100 into 01000.

A simple method to accomplish this would be to iterate over the input bitvector starting from the rightmost end until a 1 bit is found and then set it to 0. However, this algorithm is worst-case linear in the number of bits in the input bitvector. Furthermore, it uses undesirable branching code inside a loop.

There is a non-intuitive, but elegant, way to achieving the desired functionality in constant time by using a tricky composition of the standard subtraction operator and the bitwise logical & operator, which are supported by almost every architecture. The desired functionality can be achieved using the following composition:

$$I \ \& \ (I - 1)$$

The reason why we can do this seemingly worst-case linear task in unit time using the subtraction operator and the logical bitwise-and operator is because the hardware implementations of these operators manipulate the constituent bits of the bitvectors in parallel in constant time.

One way to discover the above tricky composition would be exhaustive enumeration. Let f_1 denote a unary component that implements the subtract-one operation, and let f_2 denote a binary component that implements a binary bitwise-and operation. Suppose we knew that the desired functionality can be achieved by some unknown composition of these two components f_1 and f_2. We can then simply enumerate all different ways of composing a unary operator and a binary operator, and then verify which one of them meets the functional specification with the help of an SMT solver (using the process described in Section 4). Figure 1 shows the six different straight-line-programs that can be obtained from composition of one unary and one binary operator. Of these the programs shown in 1(e) and 1(f) provide the desired functionality. There is a major problem with this explicit enumeration approach; it is too expensive. In fact, superoptimizers [28] do such an exhaustive enumeration, and hence fail to scale beyond composition of 4 components.

In contrast, our technique encodes (instead of explicitly enumerating) the space of all (six) possible straight-line programs for composing the two operations f_1 and f_2 using a logical formula $\psi_{\texttt{wfp}}$. The formula $\psi_{\texttt{wfp}}$ uses (five) integer variables, each corresponding to an input or output of some component. Intuitively, the integer variable corresponding to the output of some component denotes the line number at which the component is positioned. The integer variable corresponding to an input of some component denotes the line number where the actual parameter corresponding to that input is defined. The formula $\psi_{\texttt{wfp}}$ is such that the satisfying assignments to the integer variables have a one-to-one correspondence with the different straight-line programs that can be obtained from composition of these operators. In conjunction with some other constraints that encode the functional specifications of the base component programs and the desired program, our technique generates a formula that we refer to as the *synthesis constraint*. A satisfying assignment to the integer variables that satisfies the synthesis constraint corresponds to the desired straight-line program. The synthesis constraint is a first-order logic constraint with quantifier alternation, and is not amenable to solving directly using off-the-shelf constraint solvers. One of the key technical contributions of the paper is an algorithm to find satisfying assignments to such synthesis constraints by using an off-the-shelf SMT solver.

Even though there is no provable polynomial time guarantee associated with our technique, there is a crucial difference between the exponential exhaustive enumeration technique and our technique based on synthesis-constraint generation and solving. The number of variables in the synthesis constraint is linear in the number of components and the size of the synthesis constraint is quadratic in the number of components. The winning advantage comes from the fact that we ride upon the recent engineering advances made in SMT solving technology to solve a constraint with a linear number of unknowns as opposed to explicitly performing an exhaustive enumeration over an exponential search space.

3. Problem Definition

The goal of this paper is to synthesize a program by using a given set of base software components. The program as well as the base components are specified using their functional description. This description is given in the form of a logical formula that relates the inputs and the outputs.

For simplicity of presentation, we assume that all components have exactly one output. We also assume that all inputs and the output have the same *type*. These restrictions can be easily removed.

Formally, the synthesis problem requires the user to provide:

- A specification $\langle \vec{I}, O, \phi_{\texttt{spec}}(\vec{I}, O)\rangle$ of the program, which includes
  - a tuple of input variables $\vec{I}$ and an output variable O.
  - an expression $\phi_{\texttt{spec}}(\vec{I}, O)$ over the variables $\vec{I}$ and O that specifies the desired input-output relationship.
- A set of specifications $\{\langle \vec{I}_i, O_i, \phi_i(\vec{I}_i, O_i)\rangle \mid i = 1, \ldots, N\}$, called a *library*, where $\phi_i(\vec{I}_i, O_i)$ is a specification for base component f_i. All variables $\vec{I}_i, O_i$ are assumed distinct.

The goal of the synthesis problem is to discover a program `f_impl` that correctly implements the specification $\phi_{\texttt{spec}}$ using only the components provided in the library. The program `f_impl` is essentially a straight-line program that takes as input $\vec{I}$ and uses the set $\{O_1, \ldots, O_N\}$ as temporary variables in the following form:

`f_impl`$(\vec{I})$:
  $O_{\pi_1} := f_{\pi_1}(\vec{V}_{\pi_1})$; ... ; $O_{\pi_N} := f_{\pi_N}(\vec{V}_{\pi_N})$;
  `return` O_{π_N};

$f_{\phi_{\texttt{impl}}}(I)$: 1 $O_2 := f_2(I, I);$ 2 $O_1 := f_1(O_2);$ `return` O_1;	$f_{\phi_{\texttt{impl}}}(I)$: 1 $O_2 := f_2(I, I);$ 2 $O_1 := f_1(I);$ `return` O_1;	$f_{\phi_{\texttt{impl}}}(I)$: 1 $O_1 := f_1(I);$ 2 $O_2 := f_2(I, I);$ `return` O_2;	$f_{\phi_{\texttt{impl}}}(I)$: 1 $O_1 := f_1(I);$ 2 $O_2 := f_2(O_1, O_1);$ `return` O_2;	$f_{\phi_{\texttt{impl}}}(I)$: 1 $O_1 := f_1(I);$ 2 $O_2 := f_2(O_1, I);$ `return` O_2;	$f_{\phi_{\texttt{impl}}}(I)$: 1 $O_1 := f_1(I);$ 2 $O_2 := f_2(I, O_1);$ `return` O_2;
$l_{I_1} = 1$ $l_{O_1} = 2$ $l_{I_2} = 0$ $l_{O_2} = 1$ $l_{I_2'} = 0$	$l_{I_1} = 0$ $l_{O_1} = 2$ $l_{I_2} = 0$ $l_{O_2} = 1$ $l_{I_2'} = 0$	$l_{I_1} = 0$ $l_{O_1} = 1$ $l_{I_2} = 0$ $l_{O_2} = 2$ $l_{I_2'} = 0$	$l_{I_1} = 0$ $l_{O_1} = 1$ $l_{I_2} = 1$ $l_{O_2} = 2$ $l_{I_2'} = 1$	$l_{I_1} = 0$ $l_{O_1} = 1$ $l_{I_2} = 1$ $l_{O_2} = 2$ $l_{I_2'} = 0$	$l_{I_1} = 0$ $l_{O_1} = 1$ $l_{I_2} = 0$ $l_{O_2} = 2$ $l_{I_2'} = 1$
(a)	(b)	(c)	(d)	(e)	(f)

Figure 1. The first row shows six different ways of composing a unary component f_1 and a binary component f_2 to synthesize a straight-line program $f_{\phi_{\texttt{impl}}}$ with one input I. Second row shows an integer encoding of the corresponding program using *location variables*.

where

- each variable in $\vec{V}_{\pi_i}$ is either an input variable from $\vec{I}$, or a temporary variable O_{π_j} such that $j < i$,
- $\pi_1, \ldots, \pi_N$ is a permutation of $1, \ldots, N$, and
- the following correctness criteria holds:

$$\forall \vec{I}, O_1, \ldots, O_N : \Big(\phi_{\pi_1}(\vec{V}_{\pi_1}, O_{\pi_1}) \wedge \cdots \wedge \phi_{\pi_N}(\vec{V}_{\pi_N}, O_{\pi_N})\Big) \Rightarrow \phi_{\texttt{spec}}(\vec{I}, O_{\pi_N}) \quad (1)$$

The last formula above is called the *verification constraint*. It states the correctness criterion for the output program: for all inputs $\vec{I}$, if O_{π_N} is the output of the implementation on $\vec{I}$, then O_{π_N} should also be the output of the specification on $\vec{I}$; that is, the implementation should imply the specification.

We note that the implementation above is using *all* components from the library. We can assume this without any loss of generality. Even when there is a correct implementation using fewer components, that implementation can always be extended to an implementation that uses all components by adding dead code. Dead code can be easily identified and removed in a post-processing step.

We also note that the implementation above is using each base component only once. If there is an implementation using *multiple* copies of the same base component, we assume that the user provides multiple copies explicitly in the library (discussed further in Section 7.3). Such a restriction of using each base component only once is interesting in two regards: It can be used to enforce efficient or minimal designs. This restriction also prunes down the search space of possible designs making the problem finite and tractable.

Informally, the synthesis problem seeks to come up with an implementation – using only the base components in the given library – that implies the given specification.

EXAMPLE 1 (Problem Definition). *The problem definition for the running example in Sec. 2 can be stated as:*

- *The formal specification of the desired program to be synthesized is given by the following relationship $\phi_{\texttt{spec}}$ between the input bitvector I and the output bitvector O. We use b to denote the total number of bits in the bitvectors, and $I[j]$ to denote the bit at j^{th} position in bitvector I, when viewed as an array of bits.*

$$\phi_{\texttt{spec}}(I, O) \quad := \quad \bigwedge_{t=1}^{b} \left(\left(I[t] = 1 \wedge \bigwedge_{j=t+1}^{b} I[j] = 0 \right) \Rightarrow \left(O[t] = 0 \wedge \bigwedge_{j \neq t} O[j] = I[j] \right) \right)$$

- *The number of base components in the library is $N = 2$. One of them is a unary component f_1 that implements the subtract-one operation, and its formal specification is given by the following relationship ϕ_1 between its input parameter I_1 and output O_1.*

$$\phi_1(I_1, O_1) \quad := \quad O_1 = (I_1 - 1)$$

The other component is a binary component that implements the bitwise-and operation, and its formal specification is given by the following relationship ϕ_2 between its input parameters I_2, I_2' and output O_2.

$$\phi_2(I_2, I_2', O_2) \quad := \quad O_2 = (I_2 \;\&\; I_2')$$

4. Revisiting Verification Constraint

Before we describe our approach for solving the synthesis problem – consisting of the synthesis constraint generation phase and the constraint solving phase – we will perform two steps in this section to support the transition to these two phases. First, we will rewrite the verification constraint in Eq. 1 so that it resembles the synthesis constraint. Second, we discuss solving of the verification constraint, which is a small part of the process of solving the synthesis constraint.

Consider the verification constraint in Eq. 1. We can replace each atomic fact $\phi_{\pi_i}(\vec{V}_{\pi_i}, O_{\pi_i})$ in the antecedent by $\phi_{\pi_i}(\vec{I}_{\pi_i}, O_{\pi_i}) \wedge \vec{I}_{\pi_i} = \vec{V}_{\pi_i}$. We can also replace the fact $\phi_{\texttt{spec}}(\vec{I}, O_{\pi_N})$ in the consequent by $\phi_{\texttt{spec}}(\vec{I}, O)$ provided we add $O = O_{\pi_N}$ in the antecedent. Hence, the verification constraint can be rewritten as:

$$\forall \vec{I}, O, \vec{I}_1, .., \vec{I}_n, O_1, .., O_N : \left((O = O_{\pi_N}) \wedge \bigwedge_{i=1}^{N} (\phi_i(\vec{I}_i, O_i) \wedge \vec{I}_i = \vec{V}_i) \right) \Rightarrow \phi_{\texttt{spec}}(\vec{I}, O)$$

We now split the antecedent in the above formula into two parts $\phi_{\texttt{lib}}$ and $\phi_{\texttt{conn}}$. We also group together the formal inputs and outputs of the base components into two sets $\mathbf{P}$ and $\mathbf{R}$ to rewrite the above verification constraint as:

$$\forall \vec{I}, O, \mathbf{P}, \mathbf{R} : (\phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) \wedge \phi_{\texttt{conn}}(\vec{I}, O, \mathbf{P}, \mathbf{R})) \Rightarrow \phi_{\texttt{spec}}(\vec{I}, O) \quad (2)$$

where

$$\phi_{\texttt{lib}} := (\bigwedge_{i=1}^{N} \phi_i(\vec{I}_i, O_i)), \quad \phi_{\texttt{conn}} := (O = O_{\pi_N}) \wedge (\bigwedge_{i=1}^{N} \vec{I}_i = \vec{V}_i),$$

Here $\mathbf{P}$ and $\mathbf{R}$ denote the union of all formal inputs (Parameters) and formal outputs (Return variables) of the components:

$$\mathbf{P} := \bigcup_{i=1}^{N} \vec{I}_i \qquad \mathbf{R} := \bigcup_{i=1}^{N} \{O_i\} = \{O_1, \ldots, O_N\}$$

Note that $\phi_{\texttt{lib}}$ represents the specifications of the base components, and $\phi_{\texttt{conn}}$ represents the interconnections that includes the *mappings from formals to actuals* and from the return variable of some component to the output of the program. Observe that $\phi_{\texttt{conn}}$ is a conjunction of equalities between a variable in $\mathbf{P} \cup \{O\}$ and a variable in $\mathbf{R} \cup \vec{I}$. The connectivity constraint $\phi_{\texttt{conn}}$ determines:

- the order in which base components occur in the program.
- the value of each input parameter of each base component.

EXAMPLE 2 (Verification Constraint). *The verification constraint for the program in Figure 1(e) when regarded as a solution to the running example formally described in Example 1 is the following formula.*

$$\forall I, O, I_1, I_2, I_2', O_1, O_2\, (\phi_{\texttt{lib}} \wedge \phi_{\texttt{conn}} \Rightarrow \phi_{\texttt{spec}})$$

$$\textit{where} \quad \phi_{\texttt{lib}} \quad := \quad \phi_1(I_1, O_1) \ \wedge \ \phi_2(I_2, I_2', O_2)$$
$$\textit{and} \quad \phi_{\texttt{conn}} \quad := \quad I_1 = I \ \wedge \ I_2 = O_1 \ \wedge \ I_2' = I \ \wedge \ O = O_2$$

and $\phi_1, \phi_2, \phi_{\texttt{spec}}$ are as defined in Example 1.

We now briefly discuss the process of solving the verification constraint, which is a universally quantified formula. The complexity of deciding the validity of the formula in Eq. 2 depends on the expression language used for defining $\phi_{\texttt{spec}}$ and ϕ_i's. If this expression language is a subset of the language that can be handled by Satisfiability Modulo Theory (SMT) solvers, then we can use off-the-shelf SMT solvers to decide the formula in Eq. 2 and thus solve the verification problem. Specifically, we can check validity of a (universal) formula by asking an SMT solver for checking satisfiability of the negation of that formula.

5. Synthesis Constraint

In this section, we show how to reduce the problem of straight-line-program synthesis to that of finding a satisfying assignment to a first order logic constraint. Given a library of base components, and a specification for the desired program, we show how to generate a formula that encodes the existence of a program that is constructed using the base components and that meets the given specification.

Consider the verification constraint in Eq. 2. We are given $\phi_{\texttt{spec}}$ and $\phi_{\texttt{lib}}$ as part of the synthesis problem. However, we do not know the interconnections $\phi_{\texttt{conn}}$ between the inputs and outputs of the base components. Hence, the synthesis problem is equivalent to solving the following constraint:

$$\exists \phi_{\texttt{conn}} : \forall \vec{I}, O, \mathbf{P}, \mathbf{R} : \\ (\phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) \wedge \phi_{\texttt{conn}}(\vec{I}, O', \mathbf{P}, \mathbf{R})) \Rightarrow \phi_{\texttt{spec}}(\vec{I}, O)$$

where we have a second-order existential quantifier over the set of all possible interconnections.

In the remaining part of this section, we show how to convert the second-order existential quantifier into a first-order existential quantifier. The basic idea is to introduce new first-order integer-valued variables, referred to as *location variables*, whose values decide the interconnections between the various components. To describe a program, we have to determine *which component goes on which location (line-number), and from which location (line-number or program input) does it get its input arguments*. This information can be described by a set of *location variables* L

$$L := \{l_x \ \mid \ x \in \mathbf{P} \cup \mathbf{R}\}$$

that contains one new variable l_x for each variable x in $\mathbf{P} \cup \mathbf{R}$ with the following interpretation associated with each of these variables.

- If x is the output variable O_i of component f_i, then l_x represents the line in the program where the component f_i is positioned.
- If x is the j^{th} input parameter of component f_i, then l_x represents the *location* "from where component f_i gets its j^{th} input".

A *location* above refers to either a line of the program, or to some program input. To represent different possible locations, we use integers in the set $\{0, \ldots, M-1\}$, where M is the sum of the number N of components in the library and the number $|\vec{I}|$ of program inputs, i.e., $M = N + |\vec{I}|$, with the following interpretation.

- The j^{th} input is identified with the location $j-1$.
- The j^{th} line or the assignment statement in the program is identified with the location $j + |\vec{I}| - 1$.

EXAMPLE 3 (Location Variables). *For our running example formally described in Example 1, the set L of location variables consists of 5 integer variables. $L = \{l_{O_1}, l_{O_2}, l_{I_1}, l_{I_2}, l_{I_2'}\}$. The variables l_{O_1} and l_{O_2} denote the location at which the components f_1 and f_2 are positioned respectively. The variable l_{I_1} denotes the location of the definition of the input to the unary component f_1. The variables l_{I_2} and $l_{I_2'}$ denote the locations of the definitions of the first and the second input respectively of the binary component f_2. Since there are two components and one input, we have $N = 2$ and $M = 3$. The variables l_{O_1}, l_{O_2} take values from the set $\{1, 2\}$, while the variables $l_{I_1}, l_{I_2}, l_{I_2'}$ take values from the set $\{0, 1, 2\}$.*

The synthesis constraint, which uses the location variables L, is given in Eq. 4 in Section 5.3. We next discuss the key constituents of the synthesis constraint. For notational convenience (for the discussion below), we also define l_x for the global inputs $\vec{I}$ and output O. We define l_O to be equal to $M-1$, denoting that the output O of the program is defined on the last line of the program. For the j^{th} input x to the program, we define l_x to be $j-1$, which is the integer location that we associated with the j^{th} program input.

5.1 Encoding Well-formed Programs: $\psi_{\texttt{wfp}}$

We noted above that every straight-line program can be encoded by assigning appropriate values from the set $\{0, \ldots, M-1\}$ to variables in L. On the other hand, any possible assignment to variables in L from the set $\{0, \ldots, M-1\}$ does not necessarily correspond to a well-formed straight-line program. We require the variables in L to satisfy certain constraints to guarantee that they define well-formed programs. The following two constraints guarantee this.

Consistency Constraint : Every line in the program has at most one component. In our encoding, l_{O_i} encodes the line number where component f_i is positioned. Hence for different i, l_{O_i} should be different. Thus we get the following *consistency constraint*.

$$\psi_{\texttt{cons}} \ := \bigwedge_{x,y \in \mathbf{R}, x \not\equiv y} (l_x \neq l_y)$$

Acyclicity Constraint : In a well-formed program, every variable is initialized *before* it is used. In our encoding, component f_i is positioned at location l_{O_i} and its inputs are coming from locations $\{l_x \mid x \in \vec{I}_i\}$. Thus, we get the following *acyclicity constraint*.

$$\psi_{\texttt{acyc}} \ := \bigwedge_{i=1}^{N} (\bigwedge_{x \in \vec{I}_i, y \equiv O_i} l_x < l_y)$$

The acyclicity constraint says that, for every component, if x is an input of that component and y is an output of that component, then the location l_x where the input is defined, should be earlier than the location l_y where the component is positioned and its output is defined.

We now define $\psi_{\texttt{wfp}}(L)$ to be following constraint that encodes the interpretation of the location variables l_x along with the consistency and acyclicity constraints.

$$\psi_{\texttt{wfp}}(L) \quad := \quad \bigwedge_{x \in \mathbf{P}} (0 \leq l_x \leq M-1) \ \wedge \ \bigwedge_{x \in \mathbf{R}} (|\vec{I}| \leq l_x \leq M-1) \wedge$$

$$\psi_{\texttt{cons}}(L) \ \wedge \ \psi_{\texttt{acyc}}(L)$$

We note that if the location variables L satisfy $\psi_{\texttt{wfp}}$, then L defines a well-formed straight-line program in static single assignment (SSA) form [7], whose assignments make calls to the components in the library. Specifically, the function `Lval2Prog` returns the program corresponding to a given valuation L as follows: in the i^{th} line of `Lval2Prog`(L), we have the assignment $O_j := f_j(O_{\sigma(1)}, \ldots, O_{\sigma(t)})$ if $l_{O_j} = i, l_{I_j^k} = l_{\sigma(k)}$ for $k = 1, \ldots, t$, where t is the arity of component f_j, and $(I_j^1, \ldots, I_j^t)$ is the tuple of input variables $\vec{I}_j$ of f_j.

Our encoding has the following natural property.

THEOREM 1. *Let $\mathbf{L}$ be the set of all valuations of L that satisfy the well-formedness constraint $\psi_{\texttt{wfp}}$. Let $\mathbf{\Pi}$ be the set of all straight-line programs in SSA form that take input $\vec{I}$ and contain the N assignments, $O_i := f(\vec{V_i})$, such that every variable is defined before it is used. Then, the mapping `Lval2Prog` goes from $\mathbf{L}$ to $\mathbf{\Pi}$ and it is bijective.*

EXAMPLE 4 (Well-formedness Constraint). *For our running example formally described in Example 1, the constraint $\psi_{\texttt{wfp}}$ is:*

$$\psi_{\texttt{wfp}} \ := \ \psi_{\texttt{cons}} \wedge \psi_{\texttt{acyc}} \wedge \bigwedge_{x \in \mathbf{P}} (0 \leq l_x \leq 2) \ \wedge \ \bigwedge_{x \in \mathbf{R}} (1 \leq l_x \leq 2)$$

$$\textit{where} \quad \psi_{\texttt{cons}} := (l_{O_1} \neq l_{O_2})$$
$$\textit{and} \quad \psi_{\texttt{acyc}} := (l_{I_1} < l_{O_1}) \ \wedge \ (l_{I_2} < l_{O_2}) \ \wedge \ (l_{I_2'} < l_{O_2})$$

Here $\mathbf{P} = \{\vec{I}_1, \vec{I}_2, \vec{I}_2'\}$ and $\mathbf{R} = \{O_1, O_2\}$. There are 6 solutions for $l_{I_1}, l_{I_2}, l_{I_2'}, l_{O_1}, l_{O_2}$ that satisfy the constraint $\psi_{\texttt{wfp}}$. Each of these solutions correspond to a syntactically distinct and well-formed straight-line program obtained by composition of unary component f_1 and binary component f_2. These 6 solutions and the corresponding straight-line-programs are shown in Figure 1.

5.2 Encoding Dataflow in Programs: $\psi_{\texttt{conn}}$

Given an interconnection among components specified by values of location variables L, we can relate the input/output variables of the components and the program by the following *connectivity constraint*:

$$\psi_{\texttt{conn}} := \bigwedge_{x,y \in \mathbf{P} \cup \mathbf{R} \cup \vec{I} \cup \{O\}} (l_x = l_y \ \Rightarrow \ x = y)$$

The constraint $\psi_{\texttt{conn}}$ will play the role of $\phi_{\texttt{conn}}$ later.

5.3 Putting it all together

We are now ready to present the (first-order) *synthesis constraint* that encodes the synthesis problem.

We showed how the set of all valid programs can be described by valuations of the location variables L. Hence, the synthesis problem reduces to finding a value for the variables L such that
(1) this valuation corresponds to a well-formed program and
(2) the corresponding well-formed program is correct, as described by the verification constraint (Eq. 2).
In other words, we get the following *synthesis constraint*:

$$\exists L : (\psi_{\texttt{wfp}}(L) \wedge \forall \vec{I}, O, \mathbf{P}, \mathbf{R} : \phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) \ \wedge \ \psi_{\texttt{conn}}(\vec{I}, O, \mathbf{P}, \mathbf{R}, L) \Rightarrow \phi_{\texttt{spec}}(\vec{I}, O)) \quad (3)$$

We merge the (temporary) variables $\mathbf{P}$ and $\mathbf{R}$ and call it the set T. We rewrite the formula in Eq. 3 by pulling out the universal quantifier to get the following *synthesis constraint*.

$$\exists L \forall \vec{I}, O, T : \psi_{\texttt{wfp}}(L) \wedge (\phi_{\texttt{lib}}(T) \wedge \psi_{\texttt{conn}}(\vec{I}, O, T, L) \Rightarrow \phi_{\texttt{spec}}(\vec{I}, O)) \quad (4)$$

EXAMPLE 5 (Synthesis Constraint). *Of the 6 solutions to the location variables L described in Example 4, there are 2 solutions that satisfy the entire synthesis constraint. These two solutions are shown in Figure 1(e) and Figure 1(f).*

The following theorem states that the synthesis constraint in Eq. 4 is quadratic in size and it exactly encodes our synthesis problem. Hence, solving the synthesis problem is equivalent to solving the synthesis constraint. The proof of the theorem follows from the definition of `Lval2Prog`, Theorem 1, and the definitions of the verification and synthesis constraints.

THEOREM 2 (Synthesis Constraint). *Let $(\phi_{\texttt{spec}}, \phi_{\texttt{lib}})$ be the given specifications. Let ψ be the corresponding synthesis constraint, defined in Eq. 4, that is derived from the given specifications. The size of ψ is $O(n + m^2)$ where n is the size of $(\phi_{\texttt{spec}}, \phi_{\texttt{lib}})$ and m is the number of base components in the library. Furthermore, ψ is valid if and only if there is a straight-line program that implements the specification $\phi_{\texttt{spec}}$ using only the components in $\phi_{\texttt{lib}}$.*

PROOF: The number of variables in L is $O(m)$ and hence the size of ψ is seen to be $O(n + m^2)$.

($\Rightarrow$): Suppose ψ is valid. This implies that there exists a value for L, say L_0, such that $\psi_{\texttt{wfp}}(L_0)$ holds and the formula $\forall \vec{I}, O, \mathbf{P}, \mathbf{R} : \phi_{\texttt{lib}}(\mathbf{P}, \mathbf{R}) \wedge \psi_{\texttt{conn}}(\vec{I}, O, \mathbf{P}, \mathbf{R}, L_0) \Rightarrow \phi_{\texttt{spec}}(\vec{I}, O)$ is valid. Since $\psi_{\texttt{wfp}}(L_0)$ holds, we can use Theorem 1 to get a Program `Lval2Prog`(L), call it P. Now, the definition of `Lval2Prog` and the constraint $\psi_{\texttt{conn}}$ together guarantee that the connectivity constraint $\phi_{\texttt{conn}}$ defined by P and the connectivity constraint $\psi_{\texttt{conn}}(L_0)$ are equivalent. Since we know $\forall \vec{I}, O, \mathbf{P}, \mathbf{R} : \phi_{\texttt{lib}} \wedge \psi_{\texttt{conn}} \Rightarrow \phi_{\texttt{spec}}$ is valid, it follows that the formula $\forall \vec{I}, O, \mathbf{P}, \mathbf{R} : \phi_{\texttt{lib}} \wedge \phi_{\texttt{conn}} \Rightarrow \phi_{\texttt{spec}}$ is also valid. This shows that the verification constraint for correctness of P is valid.

($\Leftarrow$): Suppose there is a straight-line program, say P, that correctly implements the given specification $\phi_{\texttt{spec}}$ using only the components in $\phi_{\texttt{lib}}$. Given a program P, we can immediately define values for the location variables L such that $\phi_{\texttt{conn}}$ is equivalent to $\psi_{\texttt{conn}}(L)$. Since the program P is assumed to be well-formed, this valuation of L will satisfy $\psi_{\texttt{wfp}}$. Furthermore, since P is correct, the verification constraint is valid. Replacing $\phi_{\texttt{conn}}$ in the verification constraint by $\psi_{\texttt{conn}}$ shows that the synthesis constraint is also valid. □

6. Synthesis Constraint Solving

In this section, we show how to solve the synthesis constraint (Eq. 4 in Section 5.3). In particular, we show how to find an assignment to the decision variables L that would witness the validity of the synthesis constraint.

Our procedure is built over the standard counterexample-guided iterative refinement paradigm [10, 30]. It solves $\exists L \forall \vec{I} : \phi(L, \vec{I})$ by iteratively finding values for L that work for more and more values of $\vec{I}$. Let $\mathcal{S}$ denote a finite set of valuations of $\vec{I}$. In each iteration, our procedure first finds a valuation for L that works for the choices in $\mathcal{S}$. It then tests if the discovered valuation for L works *for all* $\vec{I}$. If it does not, then a valuation of $\vec{I}$ for which *it does not work* is added to the set $\mathcal{S}$ and the process is repeated.

The Procedure `ExAllSolver` shown in Figure 2 is a high-level description of our solver for the synthesis constraint. It alternates

```
ExAllSolver(ψ_wfp, φ_lib, ψ_conn, φ_spec):
1  // ∃L∀Ī, O, T : ψ_wfp ∧ (φ_lib ∧ ψ_conn ⇒ φ_spec)
   //      is a synthesis constraint
2  // Output: synthesis failed or values for L
3  S := {Ī_0} // Ī_0 is an arbitrary input
4  while (1) {
5     model := T-SAT(∃L, O_1, ..., O_n, T_1, ..., T_n : ψ_wfp(L) ∧
                        ⋀_{Ī_i ∈ S} (φ_lib(T_i) ∧ ψ_conn(Ī_i, O_i, T_i, L)
                                    ∧ φ_spec(Ī_i, O_i)));
      if (model ≠ ⊥) {
6        currL := model|_L
      } else {
7        return("synthesis failed");
      }
8     model := T-SAT(∃Ī, O, T : ψ_conn(Ī, O, T, currL) ∧
                                 φ_lib(T) ∧ ¬φ_spec(Ī, O));
      if (model ≠ ⊥) {
9        Ī_1 := model|_Ī; S := S ∪ {Ī_1};
      } else {
10       return(currL);
      }
11 }
```

Figure 2. Counterexample guided $\exists\forall$ solver for solving the synthesis constraint. Note that Line 5 and Line 8 use different formulas. If successful, the procedure outputs values for L that can be used to extract the desired straight-line program (Theorem 1).

between *finite synthesis* – finding a valuation for L that works some finite choices of $\vec{I}$ – and *verification* – checking that the valuation works for all $\vec{I}$ – and, in each iteration, the two steps learn from each other. The new value for L is always guided by a set of inputs on which the previous choice for L failed.

A technical issue worth pointing out in Procedure `ExAllSolver` is that it handles variables O, T differently from the variables $\vec{I}$ even though they are all universally quantified in the synthesis constraint. Intuitively, this is because O, T are not "independent" variables, but their values depend on $\vec{I}$ and L. Specifically, Procedure `ExAllSolver` uses the following two *different* variants of the synthesis constraint in its two steps. The formula $(\mathtt{F}_{\mathtt{ver}})$ is same as the synthesis constraint, while the formula $(\mathtt{F}_{\mathtt{syn}})$ is a weaker version of the synthesis constraint.

$$(\mathtt{F}_{\mathtt{ver}}) \quad \exists L\, \forall \vec{I}, O, T : (\psi_{\mathtt{wfp}} \wedge (\phi_{\mathtt{lib}} \wedge \psi_{\mathtt{conn}} \Rightarrow \phi_{\mathtt{spec}}))$$
$$(\mathtt{F}_{\mathtt{syn}}) \quad \exists L\, \forall \vec{I}\, \exists O, T : (\psi_{\mathtt{wfp}} \wedge (\phi_{\mathtt{lib}} \wedge \psi_{\mathtt{conn}} \wedge \phi_{\mathtt{spec}}))$$

The two phases of the procedure can now be described as follows.

Finite Synthesis (Lines 5-7): In this step, we synthesize a design that works for finitely many inputs. Specifically, the procedure finds values for L that work for all the inputs in $\mathcal{S}$ (Line 5,6). If no such values are found, we terminate and declare that no design could be found (Line 7). Line 5 is effectively solving for Formula $(\mathtt{F}_{\mathtt{syn}})$, which is different from the synthesis constraint.

Verification (Lines 8-10): In this step, we verify if the synthesized design – that we know works for the inputs in $\mathcal{S}$ – also works for all inputs. Specifically, if the generated value curr$L$ for L work for all inputs, then we terminate with success. If not, then we find an input $\vec{I}_1$ on which it does not work and add $\vec{I}_1$ to $\mathcal{S}$ (Line 9). Line 8 is verifying Formula $(\mathtt{F}_{\mathtt{ver}})$, which is the synthesis constraint.

The function `T-SAT` checks for satisfiability modulo theory of an existentially quantified formula. If the formula is satisfiable, then it returns a model, i.e., values for the existential variables that make the formula true. Note that the function `T-SAT` is essentially a call to the SMT solver.

We need to argue that Procedure `ExAllSolver` always returns the correct answer on termination. This is stated in Theorem 3. But, before that, we need a lemma that relates the two formula $(\mathtt{F}_{\mathtt{syn}})$ and $(\mathtt{F}_{\mathtt{ver}})$. Under the assumption that the implementations f_i's of the base components in the library are all *terminating*, we can prove that $(\mathtt{F}_{\mathtt{ver}})$ logically implies $(\mathtt{F}_{\mathtt{syn}})$.

LEMMA 1. *Suppose the implementation of each base component f_i in the library is terminating. Then, $(\mathtt{F}_{\mathtt{ver}})$ logically implies $(\mathtt{F}_{\mathtt{syn}})$.*

PROOF: Suppose $(\mathtt{F}_{\mathtt{ver}})$ holds. Let L_0 be the values of L that show validity of $(\mathtt{F}_{\mathtt{ver}})$. We need to prove that $(\mathtt{F}_{\mathtt{syn}})$ also holds. We will show that the values L_0 will also make the formula $(\mathtt{F}_{\mathtt{syn}})$ valid. Let $\vec{I}$ be an arbitrary input. We need to show that there are values for $\mathbf{P}, \mathbf{R}$ and O such that $\phi_{\mathtt{lib}}(\mathbf{P}, O) \wedge \psi_{\mathtt{conn}}(\vec{I}, O, \mathbf{P}, \mathbf{R}, L_0)$ holds. Since $\psi_{\mathtt{wfp}}(L_0)$ is true, it follows from Theorem 1 that there is a well-formed program P. Since all components in the library are assumed to be terminating, the program P on input $\vec{I}$ will compute at least one value for each variable in the program. These values will make the formula $\phi_{\mathtt{lib}}(\mathbf{P}, O) \wedge \psi_{\mathtt{conn}}(\vec{I}, O, \mathbf{P}, \mathbf{R}, L_0)$ true. □

The following theorem states the correctness of our constraint solving procedure, and its proof follows from Lemma 1.

THEOREM 3. *Suppose that Procedure* `ExAllSolver` *is called with the input $\psi_{\mathtt{wfp}}(L)$, $\phi_{\mathtt{lib}}(T)$, $\psi_{\mathtt{conn}}(\vec{I}, O, T, L)$, and $\phi_{\mathtt{spec}}(\vec{I}, O)$, where $T := (\mathbf{P} \cup \mathbf{R})$. Then,*

(a) If the procedure terminates with answer `synthesis successful`*, then the synthesis constraint is valid.*

(b) If the procedure terminates with answer `synthesis failed`*, then the synthesis constraint is not valid.*

PROOF: Proof of Part (a): First, since curr$L$ is (a part of) a model for the formula in Line 6, the value of curr$L$ in the program always satisfies the constraint $\psi_{\mathtt{wfp}}(L)$. Second, the procedure returns `synthesis successful` only when the constraint $\exists \vec{I}, O, T : \phi_{\mathtt{lib}} \wedge \psi_{\mathtt{conn}} \wedge \neg\phi_{\mathtt{spec}}$ is unsatisfiable. This means that the verification constraint, $\forall \vec{I}, O, T : \phi_{\mathtt{lib}} \wedge \psi_{\mathtt{conn}} \Rightarrow \phi_{\mathtt{spec}}$, is valid. This completes the proof of Part (a).

Proof of Part (b): The procedure returns `synthesis failed` only when the constraint $\exists L, O_1, \ldots, O_n, T_1, \ldots, T_n : \psi_{\mathtt{wfp}}(L) \wedge \bigwedge_{\mathbf{P}_i \in \mathcal{S}} (\phi_{\mathtt{lib}}(T_i) \wedge \psi_{\mathtt{conn}}(\vec{I}_i, O_i, T_i, L) \wedge \phi_{\mathtt{spec}}(\vec{I}_i, O_i))$ is unsatisfiable. By Lemma 1, this implies that the verification constraint is unsatisfiable. □

Now we have all the components – synthesis constraint generation (Eq. 4), synthesis constraint solving (Figure 2), and the mapping from values of L to programs (`Lval2Prog`) – to describe our overall approach. Our complete synthesis procedure is described in Figure 3, and its correctness follows from the correctness of the three steps, namely Theorem 1, Theorem 2 and Theorem 3.

7. Experimental Results

In this section, we present an experimental evaluation of our synthesis technique. We also experimentally compare our technique with other existing techniques.

Benchmarks We selected 25 examples, numbered P1-P25, from the book *Hacker's Delight*, commonly referred to as the Bible of

P1(x) : Turn-off rightmost 1 bit. This is the running example in the paper.

1 o_1:=bvsub (x,1)
2 res:=bvand (x,o_1)

P2(x) : Test whether an unsigned integer is of the form 2^{n-1}

1 o_1:=bvadd (x,1)
2 res:=bvand (x,o_1)

P3(x) : Isolate the rightmost 1-bit

1 o_1:=bvneg (x)
2 res:=bvand (x,o_1)

P4(x) : Form a mask that identifies the rightmost 1 bit and trailing 0s

1 o_1:=bvsub (x,1)
2 res:=bvxor (x,o_1)

P5(x) : Right propagate rightmost 1-bit

1 o_1:=bvsub (x,1)
2 res:=bvor (x,o_1)

P6(x) : Turn on the rightmost 0-bit in a word

1 o_1:=bvadd (x,1)
2 res:=bvor (x,o_1)

P7(x) : Isolate the rightmost 0-bit

1 o_1:=bvnot (x)
2 o_2:=bvadd (x,1)
3 res:=bvand (o_1,o_2)

P8(x) : Form a mask that identifies the trailing 0's

1 o_1:=bvsub (x,1)
2 o_2:=bvnot (x)
3 res:=bvand (o_1,o_2)

P9(x) : Absolute Value Function

1 o_1:=bvshr (x,31)
2 o_2:=bvxor (x,o_1)
3 res:=bvsub (o_2,o_1)

P10(x, y) : Test if nlz(x) == nlz(y) where nlz is number of leading zeroes

1 o_1:=bvand (x,y)
2 o_2:=bvxor (x,y)
3 res:=bvule (o_2,o_1)

P11(x, y) : Test if nlz(x) $<$ nlz(y) where nlz is number of leading zeroes

1 o_1:=bvnot (y)
2 o_2:=bvand (x,o_1)
3 res:=bvugt (o_2,y)

P12(x, y) : Test if nlz(x) $<=$ nlz(y) where nlz is number of leading zeroes

1 o_1:=bvnot (y)
2 o_2:=bvand (x,o_1)
3 res:=bvule (o_2,y)

P13(x) : Sign Function

1 o_1:=bvshr (x,31)
2 o_2:=bvneg (x)
3 o_3:=bvshr (o_2,31)
4 res:=bvor (o_1,o_3)

P14 (x, y) : Floor of average of two integers without over-flowing

1 o_1:=bvand (x,y)
2 o_2:=bvxor (x,y)
3 o_3:=bvshr (o_2,1)
4 res:=bvadd (o_1,o_3)

P15 (x, y) : Ceil of average of two integers without over-flowing

1 o_1:=bvor (x,y)
2 o_2:=bvxor (x,y)
3 o_3:=bvshr (o_2,1)
4 res:=bvsub (o_1,o_3)

P16 (x, y) : Compute max of two integers

1 o_1:=bvxor (x,y)
2 o_2:=bvneg (bvuge (x,y))
3 o_3:=bvand (o_1,o_2)
4 res:=bvxor (o_3,y)

P17(x) : Turn-off the rightmost contiguous string of 1 bits

1 o_1:=bvsub (x,1)
2 o_2:=bvor (x,o_1)
3 o_3:=bvadd (o_2,1)
4 res:=bvand (o_3,x)

P18(x) : Determine if an integer is a power of 2 or not

1 o_1:=bvsub (x,1)
2 o_2:=bvand (o_1,x)
3 o_3:=bvredor (x)
4 o_4:=bvredor (o_2)
5 o_5:=!(o_4)
6 res:=(o_5 && o_3)

P19(x, m, k) : Exchanging 2 fields A and B of the same register x where m is mask which identifies field B and k is number of bits from end of A to start of B

1 o_1:=bvshr (x,k)
2 o_2:=bvxor (x,o_1)
3 o_3:=bvand (o_2,m)
4 o_4:=bvshl (o_3,k)
5 o_5:=bvxor (o_4,o_3)
6 res:=bvxor (o_5,x)

P20(x) : Next higher unsigned number with same number of 1 bits

1 o_1:=bvneg (x)
2 o_2:=bvand (x,o_1)
3 o_3:=bvadd (x,o_2)
4 o_4:=bvxor (x,o_2)
5 o_5:=bvshr (o_4,2)
6 o_6:=bvdiv (o_5,o_2)
7 res:=bvor (o_6,o_3)

P21(x, a, b, c) : Cycling through 3 values a,b,c

1 o_1:=bvneg (bveq (x,c))
2 o_2:=bvxor (a,c)
3 o_3:=bvneg (bveq (x,a))
4 o_4:=bvxor (b,c)
5 o_5:=bvand (o_1,o_2)
6 o_6:=bvand (o_3,o_4)
7 o_7:=bvxor (o_5,o_6)
8 res:=bvxor (o_7,c)

P22(x) : Compute Parity

1 o_1:=bvshr (x,1)
2 o_2:=bvxor (o_1,x)
3 o_3:=bvshr (o_2,2)
4 o_4:=bvxor (o_2,o_3)
5 o_5:=bvand (o_4,0x11111111)
6 o_6:=bvmul (o_5,0x11111111)
7 o_7:=bvshr (o_6,28)
8 res:=bvand (o_7,0x1)

P23(x) : Counting number of bits

1 o_1:=bvshr (x,1)
2 o_2:=bvand (o_1,0x55555555)
3 o_3:=bvsub (x,o_2)
4 o_4:=bvand (o_3,0x33333333)
5 o_5:=bvshr (o_3,2)
6 o_6:=bvand (o_3,0x33333333)
7 o_7:=bvadd (o_4,o_6)
8 o_8:=bvshr (o_7,4)
9 o_9:=bvadd (o_8,o_7)
10 res:=bvand (o_9,0x0F0F0F0F)

P24(x) : Round up to the next highest power of 2

1 o_1:=bvsub (x,1)
2 o_2:=bvshr (o_1,1)
3 o_3:=bvor (o_1,o_2)
4 o_4:=bvshr (o_3,2)
5 o_5:=bvor (o_3,o_4)
6 o_6:=bvshr (o_5,4)
7 o_7:=bvor (o_5,o_6)
8 o_8:=bvshr (o_7,8)
9 o_9:=bvor (o_7,o_8)
10 o_{10}:=bvshr (o_9,16)
11 o_{11}:=bvor (o_9,o_{10})
12 res:=bvadd (o_{10},1)

P25(x, y) : Compute higher order half of product of x and y

1 o_1:=bvand (x,0xFFFF)
2 o_2:=bvshr (x,16)
3 o_3:=bvand (y,0xFFFF)
4 o_4:=bvshr (y,16)
5 o_5:=bvmul (o_1,o_3)
6 o_6:=bvmul (o_2,o_3)
7 o_7:=bvmul (o_1,o_4)
8 o_8:=bvmul (o_2,o_4)
9 o_9:=bvshr (o_5,16)
10 o_{10}:=bvadd (o_6,o_9)
11 o_{11}:=bvand (o_{10},0xFFFF)
12 o_{12}:=bvshr (o_{10},16)
13 o_{13}:=bvadd (o_7,o_{11})
14 o_{14}:=bvshr (o_{13},16)
15 o_{15}:=bvadd (o_{14},o_{12})
16 res:=bvadd (o_{15},o_8)

Figure 4. Benchmark Examples. The functions used in the examples have the usual semantics defined in SMTLIB QF_BV logic [2].

```
CompositionSynthesis(φ_spec, {φ_i | i = 1,...,N}):
   // Input: φ_spec: component specification
   //        {φ_i | i = 1,...,N}: library specification
   // Output: Failure/Program implementing φ_spec
1  Let ∃L∀Ī, O, P, R : ψ_wfp ∧ (φ_lib ∧ ψ_conn ⇒ φ_spec)
       be the synthesis constraint.
2  L := ExAllSolver(ψ_wfp, φ_lib, ψ_conn, φ_spec);
3  if (L ≠ "synthesis failed") {
      return(Lval2Prog(L));
4  } else {
      return("synthesis failed")};
5  }
```

Figure 3. Algorithm for the component-based synthesis problem.

Q1(a, b, c): Evaluate $a * h^2 + b * h + c$

```
1  o1:=a * h
2  o2:=o1+ b
3  o3:=o2* h
4  res:=o3+ c
```

Q2(x): Compute x^{31}

```
1  o1:=x * x
2  o2:=o1* o1
3  o3:=x * o2
4  o4:=o2* o3
5  o5:=o4* o2
6  o6:=o5* o4
7  res:=o6* o4
```

Figure 5. Representative Arithmetic Benchmark Examples. The arithmetic functions used in the examples have the usual semantics.

bit twiddling hacks [37]. We picked 2 non-bitvector benchmarks Q1-Q2.

The bitvector examples are described in Figure 4. The examples, P1-P25, are numbered in increasing order of complexity: P1 is a 2-line program and P25 is a 16-line program. Example Q1-Q2 in Figure 5 involve arithmetic. For each example, we provided the specification of the desired circuit by specifying the functional relationship between the inputs and output of the circuit. We also provided the set of base components (in the form of their functional specifications) used in these examples.

We chose 25 bitvector programs because they use ingenious little programming tricks that can "sometimes stall programmers for hours or days" and their correctness is "not at all obvious until explained or fathomed" [37]. Furthermore, it allows us to use existing tools based on superoptimization [4, 28] and sketching [32, 33] as a baseline for experimentally evaluating our work. We chose the 2 additional benchmarks to illustrate the need for SMT solvers.

Some programs in our benchmarks require us to also discover constants, such as `0xffff`, that may occur in the program. Our synthesis framework can be easily extended to discovering such constants by introducing a generic base component f_c that simply outputs an arbitrary constant c. Then, in the final synthesis constraint (Equation 4), we existentially quantify over c too.

Implementation and Experimental Setup We implemented our technique in a tool called `Brahma`. It uses Yices 1.0.21 [3] as the underlying SMT solver, which supports reasoning for quantifier-free bitvectors and rational linear arithmetic. For bitvector examples, we synthesized programs to work on bit-vectors of size 32 bits. We used the optimization of synthesizing for increasingly large bitvector lengths until verification succeeded on bitvectors of size 32 bits. We ran our experiments on 8x Intel(R) Xeon(R) CPU 1.86GHz with 4GB of RAM. `Brahma` was able to synthesize the desired programs for each of the benchmark examples. We now present various statistics below.

Benchmark Id	#lines	Brahma Iter.	Brahma runtime sec	Sketch runtime sec	ratio Sketch/ Brahma	AHA time(sec) [#cand]
1	2	3	4	5	6	7
P1	2	2	3.2	69.8	22	0.1[1]
P2	2	3	3.6	28.9	8	0.1[1]
P3	2	3	1.4	91.8	63	0.1[1]
P4	2	2	3.3	68.4	21	0.1[1]
P5	2	3	2.2	67.9	31	0.1[1]
P6	2	2	2.4	87.0	36	0.1[1]
P7	3	2	1.0	69.6	68	1.7[9]
P8	3	2	1.4	70.0	51	1.4[9]
P9	3	2	5.8	85.1	15	6.5[5]
P10	3	14	76.1	timeout	NA	10.4[1]
P11	3	7	57.1	timeout	NA	9.3[1]
P12	3	9	67.8	timeout	NA	9.5[1]
P13	4	4	6.2	193.7	31	timeout
P14	4	4	59.6	935.3	16	timeout
P15	4	8	118.9	726.5	6	timeout
P16	4	5	62.3	820.8	13	timeout
P17	4	6	78.1	626.1	8	108.6[9]
P18	6	5	45.9	117.2	2	timeout
P19	6	5	34.7	472.8	14	timeout
P20	7	6	108.4	timeout	NA	timeout
P21	8	5	28.3	timeout	NA	timeout
P22	8	8	279.0	timeout	NA	timeout
P23	10	8	1668.0	timeout	NA	timeout
P24	12	9	224.9	timeout	NA	timeout
P25	16	11	2778.7	timeout	NA	timeout
Q1	4	2	0.2	timeout	NA	–
Q2	7	4	295.8	timeout	NA	–

Table 1. Comparing our tool Brahma with Sketch and AHA. Timeout was 1 hour. NA denotes not applicable. The table shows the runtime for `Brahma` (Col. 4), Sketch (Col. 5) and AHA (Col. 7) on 25 benchmarks sorted by lines of code (Col. 2). We also report the number of iterations needed by `Brahma` (Col. 3), ratio of runtimes of `Brahma` and Sketch (Col. 6) and the number of candidate solutions found by AHA (within brackets in Col. 7).

7.1 Performance of Synthesis Algorithm

Table 1 reports some interesting statistics about the synthesis algorithm (presented in Fig. 3). The total time taken by the algorithm (col. 4) on the various examples varies between 1.0 to 2778.7 seconds. We also report the number of iterations taken by the loop (col. 3) inside our constraint solving algorithm in Fig. 2 while performing the refined counterexample guided iterative synthesis. The small number of these iterations (which varies between 2 to 14) illustrates the effectiveness of our technique in using counterexamples for iterative synthesis.

There has been a huge investment in building formal reasoning technology for performing verification of hardware or software systems. In this paper, we show that this verification technology can be lifted to perform synthesis. In that context, the number of iterations required by our technique points out the extra factor of computational resources required to go from verification to synthesis. The largest example in our experimental evaluation took over 45 minutes but it involved only 11 iterations. Hence, the largest SAT problem solved during synthesis is roughly 11 times the size of the SAT problem for verification. Any improvement in satisfiability solvers for verification would also directly increase the scalability of our technique.

No. of	Runtime		Ratio of Runtime	Normalized Constraint Size	
Comps.	Brahma	Sketch	Sketch/Brahma	Brahma	Sketch
1	2	3	4	5	6
2	0.11	0.27	2.45	1	1
3	0.14	0.83	5.93	1.52	5.00
4	0.20	2.09	10.45	1.91	19.85
5	0.25	6.78	27.12	2.36	48.01
6	0.36	19.69	54.70	3.18	129.26
7	0.33	164.80	499.39	3.76	242.04

Table 2. Comparing Brahma and Sketch on running example by increasing the number of components. Constraint size is normalized with respect to the size for 2 components.

7.2 Comparison with Sketch and AHA

We experimentally compared the implementation of our synthesis technique `Brahma` with two other existing tools for synthesis, namely Sketch and AHA.

Sketch The Sketch tool [32, 33] takes as input a *sketch* – a program with holes – and synthesizes programs by correctly filling the holes. Hence, for comparing Sketch with `Brahma`, we expressed the component based design problem as a sketch. There are many different ways of achieving this, and after consultation with the Sketch team, we picked one encoding that produces a sketch that has size linear in the size of the component based synthesis problem. We note that implicit in the sketch is an upper bound on how many times a component can be used. The Sketch tool can not solve the *unbounded* component based synthesis problem. We used version v1.3.0 of Sketch for comparison with our technique.

At a high level, the approach used in Sketch for synthesis is similar to the approach used by `Brahma` – both generate constraints in the first step and then use off-the-shelf solvers to solve these constraints in the second step. However, there are fundamental differences in the constraints generated by the two techniques as well as the algorithms used for solving the constraints.

Sketch internally generates Boolean constraints, which are solved using Boolean satisfiability solvers. `Brahma` generates formulas in a richer logic, which are solved using Satisfiability Modulo Theory (SMT) solvers. As a result, Sketch performs poorly when there is reasoning in a theory, such as linear arithmetic, involved. Sketch times out on the arithmetic examples, Q1 and Q2, whereas `Brahma` easily synthesizes them. (The description of Q1-Q2 may appear to involve nonlinear expressions, but nonlinear reasoning can be eliminated easily and SMT solvers can be used).

Since Sketch is a generic tool, and not tailored for component based synthesis, it uses a suitably general translation of the synthesis problem into a Boolean constraint. Consequently, while the size of the (SMT) constraints generated by `Brahma` is provably quadratic in the number of components, experimental evidence indicates that the size of the constraints generated by Sketch is either exponential or a high degree polynomial in the number of holes or components. To highlight this difference, we used `Brahma` and Sketch to synthesize the running example, but we successively increased the number of components in the library. The time taken by Sketch, reported in Col. 3 of Table 2, appears to scale exponentially, whereas the time taken by `Brahma`, reported in Col. 2 of Table 2, appears to scale non-exponentially as the number of components in the library increases from 2 to 7. The ratio of Sketch runtime to `Brahma` runtime, shown in col. 4 of Table 2, increases from 2 to nearly 500. In Table 2, we also show the normalized size of the constraints generated by both techniques against the number of components (Col. 5 and Col. 6 for `Brahma` and Sketch respectively). We normalize the size of the constraints with respect to the constraint size for 2 components. This ensures a fair comparison of the rate of increase in constraint size with increase in number of components for the two tools irrespective of the absolute size of the generated constraints which may depend on optimizations and preprocessing. Clearly, for both tools, the runtime is correlated with the size of the constraint, but constraints generated by `Brahma` are much more succinct and scale better than Sketch.

Benchmark	Verification Runtime(ms)		
	Brahma	Sketch	Ratio
P1	35	18	1.94
P2	11	16	0.69
P3	98	57	1.72
P4	58	31	1.87
P5	59	45	1.31
P6	78	32	2.43
P7	03	11	0.27
P8	78	66	1.18
P9	14	08	1.75
P10	48	NA	NA
P11	29	NA	NA
P12	29	NA	NA
P13	12	16	0.75
P14	69	38	1.82
P15	108	56	1.93
P16	77	41	1.88
P17	109	78	1.40
P18	72	47	1.53
P19	64	52	1.23
P20	96	NA	NA
P21	42	NA	NA
P22	127	NA	NA
P23	103	NA	NA
P24	62	NA	NA
P25	184	NA	NA

Table 3. Comparing the verification times of Brahma and Sketch. Timeout was 1 hour. For the *similar* verification step, Sketch is slower only by an average factor of 1.4 (maximum factor is 2.43) on all examples. NA denotes that Sketch timeouts on that example and hence there is no verification time. For the algorithmically-different synthesis step, as shown in Table 1, Sketch was slower by a factor of 20 on examples on which it terminates – so, even if we normalize for use of different constraint solvers, sketch continues to be an order-of-magnitude slower.

The total runtime of Sketch on the bitvector examples is presented in col. 5 in Table 1. Sketch times out on 6 examples and is slower by an average factor of over 20 on other examples (col. 6). One might speculate that the runtime gains of `Brahma` over Sketch arise because `Brahma` uses a different SMT solver. However, this is not true, since `Brahma` and Sketch are experimentally observed to take comparable time for performing the verification step; see Table 3. It follows that the differences are entirely due to the algorithmic improvements in `Brahma`.

AHA The AHA tool [4] is a superoptimizer, endorsed by our benchmark book [37] as *A Hacker's Assistant*. It is based on an idea by Henry Massalin [28], and was made widely available by Granlund and Kenner as the GNU superoptimizer [11]. For experimental comparison, we provided the set of base components as a set of library functions. AHA enumerates all possible composition of these functions to generate candidate programs (in a way described in Figure 1), but it tests the correctness of the candidate programs only on some inputs, and often outputs a number of po-

tential solutions. The solutions produced by AHA must be verified in order to select the right solution. Table 1 lists the total number of solutions generated by AHA (col. 7 within [brackets]) and the total time (col. 7) taken for generation and verification of these solutions. AHA times out on 12 examples. The better performance of `Brahma` is explained by the fact that `Brahma` does not perform an exhaustive enumeration of the exponential state space, but relies on a non-trivial strategy of candidate selection and elimination though SMT solving. Thus, we exploit the engineering advances in the underlying SMT solving technology for an efficient search.

7.3 Choice of Multi-set of Base Components

We now discuss the strategy that we used for choosing the multi-set of base components for our benchmark examples. Picking the multi-set of base components is the only step in our approach that currently requires human guidance.

In our experiments, we started with a common multi-set of base components, referred to as *the standard library*, for all benchmarks. The standard library included 12 components, one each for performing standard operations, such as bitwise-and, bitwise-or, bitwise-not, add-one, bitwise-xor, shift-right, comparison, add, and subtract operations. The standard library was sufficient for synthesizing the first 17 benchmark examples. For other examples, the library was augmented with a set of new components suggested by the user. We call this set *the extended library*. (Giving user the option to extend libraries can facilitate hierarchical synthesis – the user can specify a synthesized program as a new component in the extended library.)

For the above-mentioned incremental design technique to be successful, it is pertinent that the synthesis engine not only synthesize correct designs quickly but also report infeasibility of the synthesis problem quickly. In our experiments, we noted that `Brahma` reports infeasibility of design rather quickly. More specifically, when the standard library was insufficient, `Brahma` terminated in less than 100 seconds on almost all examples. Hence, *reliance on human guidance can be reduced using a strategy where components are successively added to the library until synthesis is successful.*

Regarding the issue of synthesis of optimal designs – designs that use the minimal number of components – we observed that in experiments, we always got minimal designs. However, this is not a guarantee. Minimality can, however, be ensured by iteratively removing each component as long as a design exists.

8. Related Work

The component-based synthesis problem was recently independently formulated for the case when the components were finite-state machines with outputs (transducers) [25]. Our formulation allows for high-level components whose specifications are given using logical formulas in rich theories. More recently, linear-time programs were synthesized from specification automata recognizing the input/output relation [15].

Counterexample Guided Inductive Synthesis Inductive synthesis refers to the process of generating a system from input-output examples. This process involves using each new input-output example to refine the hypothesis about the system until convergence is reached. Inductive synthesis had its origin in the pioneering work by Gold on language learning [10] and by Shapiro on algorithmic debugging and its application to automated program construction [30]. The inductive approach [29, 9] for synthesizing a program involves *debugging* the program with respect to positive and negative examples until the correct program is synthesized. The negative examples can be counterexamples discovered while trying to prove a program's correctness. Counterexamples have been used in incremental synthesis of programs [33, 17] and switching logic for hybrid systems [20].

We have recently extended the ideas described in this paper to provide an alternative to writing formal specifications for synthesis [19], wherein logical specification of the desired program is replaced by an input-output oracle. This is especially important for the application of software deobfuscation. Quite interestingly, this also obviates the need for having a verifier, albeit at the cost of introducing potential unsoundness in the process. In contrast, the approach in this paper makes use of a verifier and we show how to transform a verifier into a synthesizer. This theme of transforming a verifier into a synthesizer is also present in recent work [36, 34], where the focus is on synthesizing code fragements along with inductive invariants as first-order instantiations of user-provided templates. In contrast, our focus in this paper is restricted to straight-line code fragments, but with the benefit of requiring the user to only provide a multi-set of required components.

Automated API Composition The Jungloid mining tool [26] synthesizes code-fragments (over a given set of API methods annotated with their type signatures) given a simple query that describes the desired code in terms of input and output types. We push this work forward to synthesizing code-fragments that meet a functional specification as opposed to simply type specifications. Typing constraints can also be easily incorporated in our synthesis constraints.

`DIPACS` [21] compiler incorporates an AI planner to replace a call of a programmer-defined abstract algorithm with a sequence of library calls. It uses programmer-compiler interaction to prune undesirable compositions. `DIPACS` requires the library (or application) programmer to specify behavior of the library procedures (or, desired effect of the abstract algorithm) using high-level *abstractions*, such as predicates *sorted* and *permutation*. Furthermore, it then needs axioms for these predicates. This is similar to some early work on automatic program synthesis [27, 35], where a theorem prover was used instead of an AI planner. This early work was later extended [8] to perform schema-guided deductive synthesis. Our approach does not use abstract predicates and axioms and relies on the predicates provided by the SMT solver. Our approach can only solve synthesis problems whose formalization can be done in SMT-supported theories, whereas the work on deductive synthesis uses more general-purpose deductive engines, but at the price of requiring axiomatization and performing incomplete reasoning. In our approach, the SMT solver reasons about the implicit theories using decision procedures.

Sketching *Sketching* [32, 33, 31] relies on the developer to come up with the algorithmic insight and uses the sketch compiler to fill in missing details using counterexample guided inductive synthesis. In contrast, our tool seeks to discover algorithmic insights, albeit at cost of being more suited for a special class of programs. We chose bitvector programs as our main application domain since coming up with algorithmic insight is the hard part here.

Super-optimizers Superoptimization is the task of finding an optimal code sequence for a straight-line target sequence of instructions, and it is used in optimizing performance-critical inner loops. One approach to superoptimization has been to simply enumerate sequences of increasing length or cost, testing each for equality with the target specification [28]. Another approach has been to constrain the search space to a set of equality-preserving transformations expressed by the system designer [22] and then select the one with the lowest cost. Recent work has used superoptimization [5, 6] to automatically generate general purpose peephole optimizers by optimizing a small set of instructions in the code. In these approaches, the exhaustive state space search is quite expensive making them amenable to only discovering optimal instructions of length four or less in reasonable amount of time.

Use of satisfiability solving for synthesis SAT solvers have been used for synthesis previously: Massalin [28] used them for verification of candidate synthesized programs and Sketching [33] used them to implement the inductive program synthesis technique. We use SMT solving to implement our algorithm for solving synthesis constraints. This makes our synthesis approach more efficient as well as more general.

9. Conclusion and Future Work

Program synthesis has the potential to revolutionize the process of system development. Up until recently, automating synthesis of non-trivial programs has mostly been impractical. However, huge engineering advances in logical reasoning have significantly changed the landscape. These advances have enabled verification of large systems, and in this paper, we show that they can also be used to synthesize 10-20 line programs with some help from the user and using computational resources that are within one order of magnitude of the resources required for verification.

We have applied our component-based synthesis methodology to synthesis of bit-vector circuits. However, our solution applies more generally to other naturally resource-constrained domains, such as loop-free assembly of physical components (e.g., FPGA circuits, or even some physical/chemical/biological systems). In such situations, the user naturally starts out with resource constraints that provide a good over-approximation of the multi-set of available components.

We also foresee generalizations of our formulation of the component-based synthesis problem. This includes synthesizing programs with richer control structure, such as loops and recursion, and synthesizing from partial specifications. There is also potential for using richer theories, and limited first-order reasoning that is supported by modern SMT solvers, to synthesize from components whose specifications use richer logical formulas. A followup problem worth investigating is to approximate some functionality using a given multi-set of components. We believe that our paper lays down the foundation for investigating such a line of work.

Acknowledgment We thank Viktor Kuncak, Rishabh Singh and other members of the Sketch team, and the anonymous reviewers for their help and valuable feedback.

References

[1] Satisfiability modulo theories competition (smt-comp). `http://www.smtcomp.org/2009/index.shtml`.

[2] SMTLIB: Satisfiability modulo theories lib. `http://smtlib.org`.

[3] Yices: An SMT solver. `http://yices.csl.sri.com`.

[4] The AHA! (A Hacker's Assistant) Superoptimizer, 2008. Download: `http://www.hackersdelight.org/aha.zip`, Documentation: `http://www.hackersdelight.org/aha/aha.pdf`.

[5] S. Bansal and A. Aiken. Automatic generation of peephole superoptimizers. In *ASPLOS*, 2006.

[6] S. Bansal and A. Aiken. Binary translation using peephole superoptimizers. In *OSDI*, 2008.

[7] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *POPL*, 1989.

[8] B. Fischer and J. Schumann. Autobayes: A system for generating data analysis programs from statistical models. *J. Funct. Program.*, 13(3):483–508, 2003.

[9] P. Flener and L. Popelmnsky. On the use of inductive reasoning in program synthesis: Prejudice and prospects. In *LOBSTR*. 1994.

[10] E. M. Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[11] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and In *PLDI*, 1992.

[12] S. Gulwani. Dimensions in program synthesis. In *PPDP*, pages 13–24. ACM, 2010.

[13] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[14] S. Gulwani, V. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[15] J. Hamza, B. Jobstmann, and V. Kuncak. Synthesis for regular specifications over unbounded domains. In *FMCAD*, 2010.

[16] B. Harris and S. Gulwani. Spreadsheet table transformations from examples. In *PLDI*, 2011.

[17] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. In *OOPSLA*, pages 36–46, 2010.

[18] R. N. Jackiw and W. F. Finzer. The geometer's sketchpad: programming by geometry. In *Watch what I do: programming by demonstration*, pages 293–307. MIT Press, 1993.

[19] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, pages 215–224, 2010.

[20] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Synthesizing switching logic for safety and dwell-time requirements. In *Proc. 1st ACM/IEEE Intl. Conf. on Cyber-Physical Systems, ICCPS*, pages 22–31, 2010.

[21] T. A. Johnson and R. Eigenmann. Context-sensitive domain-independent algorithm composition and selection. In *PLDI*, 2006.

[22] R. Joshi, G. Nelson, and K. H. Randall. Denali: A goal-directed superoptimizer. In *PLDI*, 2002.

[23] D. E. Knuth. The art of computer programming. `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`.

[24] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, 2000.

[25] Y. Lustig and M. Vardi. Synthesis from component libraries. In *Proc. FoSSaCS*, pages 395–409, 2009.

[26] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *PLDI*, pages 48–61, 2005.

[27] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM TOPLAS*, 2(1):90–121, 1980.

[28] H. Massalin. Superoptimizer - a look at the smallest program. In *ASPLOS*, pages 122–126, 1987.

[29] S. Muggleton, editor. *Inductive Logic Programming*, volume 38 of *The APIC Series*. Academic Press, 1992.

[30] E. Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[31] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodík, V. A. Saraswat, and S. A. Seshia. Sketching stencils. In *PLDI*, pages 167–178, 2007.

[32] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In *PLDI*, 2005.

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

[34] S. Srivastava, S. Gulwani, and J. S. Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[35] M. Stickel, R. Waldinger, M. Lowry, T. Pressburger, and I. Underwood. Deductive composition of astro. software from subroutine libraries. In *CADE*, '94.

[36] A. Taly, S. Gulwani, and A. Tiwari. Synthesizing switching logic using constraint solving. In *VMCAI*, pages 305–319. Springer, 2009.

[37] H. S. Warren. *Hacker's Delight*. Addison-Wesley, '02.

[38] I. H. Witten and D. Mo. TELS: learning text editing tasks from examples. In *Watch what I do: programming by demonstration*, pages 293–307. MIT Press, Cambridge, MA, USA, 1993.