Component-based Synthesis Applied to Bitvector Programs

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Susmit Jha
UC Berkeley
jha@eecs.berkeley.edu

Ashish Tiwari
SRI International
tiwari@csl.sri.com

Ramarathnam Venkatesan
Microsoft Research
venkie@microsoft.com

ABSTRACT

We define *component-based synthesis* to be the problem of synthesis of (straight-line) programs from appropriate composition of base components from a specified library of software components. The functional specification of the desired program and the library components is provided in the form of logical formulas that relate the respective input and output variables. This has applications in design of intricate circuits or algorithms, superoptimization, and API mining. Furthermore, automated synthesis provides the promise of correctness by construction, generation of efficient systems, and improvement in developer's productivity.

We solve the 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 logic formula whose size is quadratic in the number of components, but has quantifier alternation. 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 on synthesizing a variety of bitvector algorithms that involve unintuitive composition of standard bitvector operations and are difficult to synthesize manually. We also compare our technique with existing synthesis approaches based on sketching and superoptimization. Our tool `Brahma` can efficiently synthesize highly nontrivial 10-20 line loop-free programs. These programs represent a state space of approximately 20^{10} programs, and are beyond the reach of the other tools.

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 two applications of composition to another dimension, that of automated synthesis of systems from simply a specification of the desired system and specifications of base components. For technical reasons, we restrict ourselves to a circuit-style composition of these components, or equivalently, a straight-line program built out of these components. This is already a large and useful class of systems. Note that loop-free control-flow can be encoded by providing the ite (if-then-else) operator as a base component. As part of future work, we plan to study even loopy composition of these components.

Automated 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 correct components and the correct composition manually can be a daunting task, especially when the base component library is huge.

While we foresee several applications of component-based program synthesis, in this paper we consider a specific application – 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 [15]. In this paper, we demonstrate how to automate the discovery of small, but intricate, bitvector programs using the currently available formal verification technology.

The ability to automatically synthesize correct programs that accomplish a certain task can be used 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 the translation could be reducing 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 automatically replace vulnerable snippets of code by the automatically 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 [20, 11, 14] and sketching [25, 26] 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 to deal with problem instances that arise in practice, which are usually not hard, and hence end up not requiring exponential reasoning.

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, 22] 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. This method works great for certain classes of constraints, but not for our synthesis constraint.

Our refinement to the iterative synthesis strategy involves working with two representations of the synthesis constraint.

- The original $\exists\forall$ representation.
  This is used to check whether a solution to the existentially quantified variables found using only a few test cases (for the universally quantified variables) is, in fact, a correct solution.
- An alternate $\exists\forall\exists$ representation, where the universal quantification is only over input variables of the system.
  This is used to find a solution to the existentially quantified variables that works for a set of test cases.

At a high level, it is quite counter-intuitive how such an alternate representation helps since an extra level of quantifier alternation makes it even more difficult to reason about the constraint. However, it helps by reducing the universal quantification to be only over the "true inputs" of the system. This enables us to solve for the existentially quantified variables by using a few choices for only the true inputs (and making copies of the inner existentially quantified variables for each such choice). We experimentally validate this strategy for our application domain of bitvector program synthesis. Furthermore, the classic result on learning AC^0 circuits from a few test inputs [16] provides an excellent theoretical justification for the effectiveness of this strategy. The result relies on a theorem that states that AC^0 circuits can be approximated well by low-degree polynomials, which in turn are known to be identifiable by their behavior on few inputs.

We have implemented our constraint generation and constraint solving technique in a tool called `Brahma`. We have applied `Brahma` to 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 [28] in time ranging from 1.0 to 2778.7 seconds. In contrast, the Sketch and AHA tools, based respectively on sketching and super-optimization, timeout 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.

[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].

- 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 synthesis of bitvector programs using standard bitvector operators (Section 7). We also experimentally compare our technique with other existing techniques, namely sketching and superoptimization, that can be used to synthesize bitvector programs (Section 8). 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:

$$I \mathbin{\&} (I - 1)$$

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 [20] 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 used. The integer variable corresponding to an input of some component denotes the line number from 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 over 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 $\{\langle \vec{I_i}, O_i, \phi_i(\vec{I_i}, O_i)\rangle \mid i = 1, \ldots, N\}$,

$f_{\phi_{\tt impl}}(I)$: $O_2 := f_2(I,I)$; $O_1 := f_1(O_2)$; `return` O_1;	$f_{\phi_{\tt impl}}(I)$: $O_2 := f_2(I,I)$; $O_1 := f_1(I)$; `return` O_1;	$f_{\phi_{\tt impl}}(I)$: $O_1 := f_1(I)$; $O_2 := f_2(I,I)$; `return` O_2;	$f_{\phi_{\tt impl}}(I)$: $O_1 := f_1(I)$; $O_2 := f_2(O_1,O_1)$; `return` O_2;	$f_{\phi_{\tt impl}}(I)$: $O_1 := f_1(I)$; $O_2 := f_2(O_1,I)$; `return` O_2;	$f_{\phi_{\tt impl}}(I)$: $O_1 := f_1(I)$; $O_2 := f_2(I,O_1)$; `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 $f_{\phi_{\tt impl}}$ with one input I. Second row shows an integer encoding of the corresponding program using *location variables*.

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 $\phi_{\tt spec}$ using only the components provided in the library[17]. 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 $O_{\pi_N}$;
```

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:

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

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 8.3). 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 $\phi_{\tt spec}$ 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.*

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

- *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 $\phi_{\tt spec}(\vec{I}, O_{\pi_N})$ in the consequent by $\phi_{\tt spec}(\vec{I}, O)$ provided we add $O = O_{\pi_N}$ in the antecedent. Hence, the verification constraint can be rewritten as:

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

We now split the antecedent in the above formula into two parts $\phi_{\tt lib}$ and $\phi_{\tt conn}$. 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:

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

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

P and **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 $\phi_{\mathtt{lib}}$ represents the specifications of the base components, and ϕ_{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 ϕ_{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 ϕ_{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.*

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

$$\begin{array}{llll} \textit{where} & \phi_{\mathtt{lib}} & := & \phi_1(I_1, O_1) \;\wedge\; \phi_2(I_2, I_2', O_2) \\ \textit{and} & \phi_{\mathsf{conn}} & := & I_1 = I \;\wedge\; I_2 = O_1 \;\wedge\; I_2' = I \;\wedge\; O = O_2 \end{array}$$

and $\phi_1, \phi_2, \phi_{\mathsf{spec}}$ 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 ϕ_{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 ϕ_{spec} and $\phi_{\mathtt{lib}}$ as part of the synthesis problem. However, we do not know the interconnections ϕ_{conn} between the inputs and outputs of the base components. Hence, the synthesis problem is equivalent to solving the following constraint:

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

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 used.
- 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 used 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-formedness of Programs: $\psi_{\mathtt{wfp}}$

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 used. Hence for different i, l_{O_i} should be different. Thus we get the following *consistency constraint*.

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

Acyclicity Constraint : In a well-formed program, every variable is initialized *before* it is used. In our encoding, component f_i is used 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 used 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.

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

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 [8], 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 $\texttt{Lval2Prog}(L)$, we have the assignment $O_j := f_j(O_{\sigma(1)},..,O_{\sigma(t)})$ if $l_{O_j} = i$, $l_{I_j^k} = l_{\sigma(k)}$ for $k = 1,..,t$, where t is the arity of component f_j, and $(I_j^1,..,I_j^t)$ is the tuple of input variables $\vec{I}_j$ of f_j.

It isn't difficult to prove following property about our encoding.

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:*

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

$$\textit{where} \quad \psi_{\texttt{cons}} \ := \ (l_{O_1} \neq l_{O_2})$$

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

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

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

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 $\texttt{Lval2Prog}(L)$, 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.

We describe our procedure for solving the synthesis constraint, which has a quantifier alternation of the form ∃∀, in two steps. First, in Section 6.1, we present a generic solver for ∃∀ formulas. This solver can be built modularly over any existing satisfiability solver. It is based on the standard counterexample-guided iterative refinement paradigm [10, 22]. The generic solver is not limited to solving only the synthesis constraint. However, because of its generality, it turns out to be inefficient for our purpose. Then, in Section 6.2, we refine the generic procedure to efficiently solve the synthesis constraint.

6.1 Standard Counterexample-Guided Solver

The pseudocode for the generic procedure `StandardExAllSolver` is presented in Figure 2. The input to the procedure is a formula of the form $\exists L \forall \vec{I} : \phi(L, \vec{I})$. This procedure is iterative and it needs a seed to start. This seed is an arbitrarily chosen value $\vec{I}_0$ for the universally quantified variables $\vec{I}$. The program variable $\mathcal{S}$ is initialized to $\{\vec{I}_0\}$ (in Line 3). The procedure then iteratively performs the following steps:

Finite Synthesis (Lines 5-7): In this step, the procedure finds a value for the existential variables L that work for (only) finitely many choices $\mathcal{S}$ for the universal variables $\vec{I}$. (Line 5,6). If no such value for L is found, then we terminate and declare the formula as unsatisfiable (Line 7).

Verification (Lines 8-10): In this step, we verify if the value curr$L$ for existential variables L found in the previous step – that we know works for the values in $\mathcal{S}$ – also works for *all* possible values of the universal variables. If so, we return "Satisfiable" (Line 10) If not, we find a value $\vec{I}_1$ on which it does not work and add $\vec{I}_1$ to $\mathcal{S}$ (Line 9).

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 the above approach for solving ∃∀ formulas is correct. It is easily seen to be sound: if the procedure `StandardExAllSolver` terminates, then it terminates with the correct answer. It is also easy to show that the procedure makes progress in every iteration. Specifically, in every iteration, at least one choice of values for L is forever eliminated. If the domain of L is bounded, then this observation also proves termination of the above method. Of course, all these results hold only under the assumption that the base satisfiability procedure (used in Line 5 and Line 8) is sound, complete and terminating.

```
StandardExAllSolver($\exists L \forall \vec{I} : \phi(L, \vec{I})$):
1    // Input $\exists L \forall \vec{I} : \phi$ is an exists-forall formula
2    // Output:  unsatisfiable or satisfiable
3    $\mathcal{S} := \{\vec{I}_0\}$ // $\vec{I}_0$ is an arbitrary value for $\vec{I}$
4    while (1) {
5       model := T-SAT($\exists L : \bigwedge_{\vec{I}_0 \in \mathcal{S}} \phi(L, \vec{I}_0)$);
6       if (model $\neq \bot$) {curr$L$ := model$|_L$}
7       else {return("unsatisfiable")};
8       model := T-SAT($\exists \vec{I} : \neg\phi(\text{curr}L, \vec{I})$);
9       if (model $\neq \bot$) { $\vec{I}_1$ := model$|_{\vec{I}}$; $\mathcal{S}$ := $\mathcal{S} \cup \{\vec{I}_1\}$}
10      else {return("satisfiable")};
11   }
```

Figure 2: Standard counterexample guided ∃∀ solver built using an ∃ satisfiability solver.

```
RefinedExAllSolver($\psi_{\texttt{wfp}}$, $\phi_{\texttt{lib}}$, $\psi_{\texttt{conn}}$, $\phi_{\texttt{spec}}$):
1    // $\exists L \forall \vec{I}, O, T : \psi_{\texttt{wfp}} \wedge (\phi_{\texttt{lib}} \wedge \psi_{\texttt{conn}} \Rightarrow \phi_{\texttt{spec}})$
     //       is a synthesis constraint
2    // Output:  synthesis failed or values for $L$
3    $\mathcal{S} := \{\vec{I}_0\}$ // $\vec{I}_0$ is an arbitrary input
4    while (1) {
5       model := T-SAT($\exists L, O_1, \ldots, O_n, T_1, \ldots, T_n : \psi_{\texttt{wfp}}(L) \wedge$
                  $\bigwedge_{\vec{I}_i \in \mathcal{S}} (\phi_{\texttt{lib}}(T_i) \wedge \psi_{\texttt{conn}}(\vec{I}_i, O_i, T_i, L)$
                      $\wedge \phi_{\texttt{spec}}(\vec{I}_i, O_i)))$;
6       if (model $\neq \bot$) {curr$L$ := model$|_L$}
7       else {return("synthesis failed")};
8       model := T-SAT($\exists \vec{I}, O, T : \psi_{\texttt{conn}}(\vec{I}, O, T, \text{curr}L) \wedge$
                      $\phi_{\texttt{lib}}(T) \wedge \neg\phi_{\texttt{spec}}(\vec{I}, O)$);
9       if (model $\neq \bot$) {$\vec{I}_1$ := model$|_{\vec{I}}$; $\mathcal{S}$ := $\mathcal{S} \cup \{\vec{I}_1\}$;}
10      else {return(curr$L$)};
11   }
```

Figure 3: Refined counterexample guided ∃∀ 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).

The iteration between *finite synthesis* and *verification* steps is attractive because, 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.

6.2 Refined Counterexample-Guided Solver

The generic procedure `StandardExAllSolver`, when given the synthesis constraint in Eq. 4, performs no better than a naive exhaustive enumeration (as we also found experimentally for our benchmark examples). We first explain why this is the case, and then we refine the generic procedure to ensure that it works effectively on synthesis constraints.

First, let us observe what happens when we use Procedure `StandardExAllSolver` directly to solve the synthesis constraint (Eq. 4). Since the universal quantification in this formula is over $\vec{I}, O, T$, the variable $\mathcal{S}$ of Procedure `StandardExAllSolver` will need to keep tuples of the form $(\vec{I}_0, O_0, T_0)$. Performing two iterations of the procedure will convince the reader that the procedure will maintain a set $\{(\vec{I}_0, O_0, T_0), (\vec{I}_1, O_1, T_1), \ldots\}$ of tuples that correspond to “runs” of the implementations that have been tried so far. Since the implementations have not worked, $\phi_{\texttt{spec}}(\vec{I}_i, O_i)$ does not hold for each such tuple. In the verification phase, the procedure will add another such tuple to the above set. In the synthesis phase, the procedure will find a new implementation (new values for L) that is *inconsistent with all the above runs.* Hence, rather than finding implementations that work on more and more inputs, the procedure is finding implementations that are simply different from earlier ones.

It is easy to see that, as a result, the iterative loop of the procedure essentially performs “exhaustive enumeration”. What this means is that, since Procedure `StandardExAllSolver` is sound, we still get sound answers, but the number of iterations required to terminate (and the probability of nontermination) is greatly increased. This is clearly undesirable. Ideally, we want to keep values of only the inputs $\vec{I}$ in the set $\mathcal{S}$ and then synthesize designs that *work* for these finitely many inputs. We do not want to keep values for the temporary variables T since *they can change as the design changes.* We do not want to force them to remain unchanged.

The modified procedure, Procedure `RefinedExAllSolver`, is shown in Figure 3. The crucial difference is that the new procedure uses the following two different variants of the synthesis constraint in the two phases. The formula $(\texttt{F}_{\texttt{ver}})$ is same as the synthesis constraint, while the formula $(\texttt{F}_{\texttt{syn}})$ is a weaker version of the synthesis (Lemma 1 on Page).

$$
\begin{array}{ll}
(\texttt{F}_{\texttt{ver}}) & \exists L\ \forall \vec{I}, O, T : (\psi_{\texttt{wfp}} \wedge (\phi_{\texttt{lib}} \wedge \psi_{\texttt{conn}} \Rightarrow \phi_{\texttt{spec}})) \\
(\texttt{F}_{\texttt{syn}}) & \exists L\ \forall \vec{I}\ \exists O, T : (\psi_{\texttt{wfp}} \wedge (\phi_{\texttt{lib}} \wedge \psi_{\texttt{conn}} \wedge \phi_{\texttt{spec}}))
\end{array}
$$

The new procedure is similar to old one and works in 2 phases.

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 $(\texttt{F}_{\texttt{syn}})$, 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 $(\texttt{F}_{\texttt{ver}})$, which is the synthesis constraint.

Procedures `RefinedExAllSolver` and `StandardExAllSolver` perform similar operations on matching line numbers. However, by using Formula $(\texttt{F}_{\texttt{syn}})$in the synthesis phase, we guarantee that when we synthesize L, it “works” for the inputs in $\mathcal{S}$. In the verification phase, we continue to use the actual synthesis constraint.

We need to argue that Procedure `RefinedExAllSolver` 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 $(\texttt{F}_{\texttt{syn}})$ and $(\texttt{F}_{\texttt{ver}})$. Under the assumption that the implementations f_i's of the base components in the library are all *terminating*, we can prove that $(\texttt{F}_{\texttt{ver}})$ logically implies $(\texttt{F}_{\texttt{syn}})$.

```
CompositionSynthesis($\phi_{\texttt{spec}}$, $\{\phi_i \mid i = 1, \ldots, N\}$):
    // Input:  $\phi_{\texttt{spec}}$:  component specification
    //         $\{\phi_i \mid i = 1, \ldots, N\}$:  library specification
    // Output:  Failure/Program implementing $\phi_{\texttt{spec}}$
    Let $\exists L \forall \vec{I}, O, \mathbf{P}, \mathbf{R} : \psi_{\texttt{wfp}} \wedge (\phi_{\texttt{lib}} \wedge \psi_{\texttt{conn}} \Rightarrow \phi_{\texttt{spec}})$
        be the synthesis constraint.
    L := RefinedExAllSolver($\psi_{\texttt{wfp}}, \phi_{\texttt{lib}}, \psi_{\texttt{conn}}, \phi_{\texttt{spec}}$);
    if (L $\neq$ "synthesis failed") {return(Lval2Prog(L))}
    else {return("synthesis failed")};
```

Figure 4: Algorithm for the component-based synthesis problem.

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

Proof. Suppose $(\texttt{F}_{\texttt{ver}})$ holds. Let L_0 be the values of L that show validity of $(\texttt{F}_{\texttt{ver}})$. We need to prove that $(\texttt{F}_{\texttt{syn}})$ also holds. We will show that the values L_0 will also make the formula $(\texttt{F}_{\texttt{syn}})$ 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 $\phi_{\texttt{lib}}(\mathbf{P}, O) \wedge \psi_{\texttt{conn}}(\vec{I}, O, \mathbf{P}, \mathbf{R}, L_0)$ holds. Since $\psi_{\texttt{wfp}}(L_0)$ 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 $\phi_{\texttt{lib}}(\mathbf{P}, O) \wedge \psi_{\texttt{conn}}(\vec{I}, O, \mathbf{P}, \mathbf{R}, L_0)$ true. □

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

Theorem 3. *Suppose that Procedure* `RefinedExAllSolver` *is called with the input $\psi_{\texttt{wfp}}(L)$, $\phi_{\texttt{lib}}(T)$, $\psi_{\texttt{conn}}(\vec{I}, O, T, L)$, and $\phi_{\texttt{spec}}(\vec{I}, O)$, 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 $\psi_{\texttt{wfp}}(L)$. Second, the procedure returns `synthesis successful` only when the constraint $\exists \vec{I}, O, T : \phi_{\texttt{lib}} \wedge \psi_{\texttt{conn}} \wedge \neg \phi_{\texttt{spec}}$ is unsatisfiable. This means that the verification constraint, $\forall \vec{I}, O, T : \phi_{\texttt{lib}} \wedge \psi_{\texttt{conn}} \Rightarrow \phi_{\texttt{spec}}$, is valid. This completes the proof of Part (a).

Proof of Part (b): The procedure returns `synthesis failed` only when the constraint $\exists L, O_1, \ldots, O_n, T_1, \ldots, T_n : \psi_{\texttt{wfp}}(L) \wedge \bigwedge_{\mathbf{P}_i \in \mathcal{S}} (\phi_{\texttt{lib}}(T_i) \wedge \psi_{\texttt{conn}}(\vec{I}_i, O_i, T_i, L) \wedge \phi_{\texttt{spec}}(\vec{I}_i, O_i))$ 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 3), and the mapping from values of L to programs (`Lval2Prog`) – to describe our overall approach. Our complete synthesis procedure is described in Figure 4, and its correctness follows from the correctness of the three steps, namely Theorem 1, Theorem 2 and Theorem 3.

7. APPLICATION TO BITVECTOR PROGRAMS

We chose the domain of bitvector programs for applying our theory of component-based synthesis. The running example described in Section 2 belongs to this domain. Synthesis of bitvector programs has two main applications. (a) Efficient bitvector code-fragments are of great significance for people who write optimizing compilers or high-performance code as these code-fragments can be used to speed up the inner loop of some integer or bit-fiddly computation. (b) These are also helpful for designing specialized hardware.

We chose this domain for the following two primary reasons.

- There is a need for automated tools for synthesizing bitvector manipulating algorithms since these are usually ingenious little programming tricks that can "sometimes stall programmers for hours or days if they really want to understand why things work". These algorithms "typically describe some plausible yet unusual operation on integers or bit strings that could easily be programmed using either a longish fixed sequence of machine instructions or a loop, but the same thing can be done much more cleverly using just four or three or two carefully chosen instructions whose interactions are not at all obvious until explained or fathomed" [28].
- There are two existing techniques that can also be used to synthesize bitvector programs: superoptimizers [4, 20] and sketching [25, 26]. This allows for experimental comparison of our technique with existing techniques, which work in a fundamentally different way.

An additional challenge that this domain offers is the presence of arbitrary constants in some programs. Our synthesis framework can be easily extended to discovering such constants. For this purpose, we introduce a generic base component f_c that simply outputs some arbitrary constant c. The component f_c takes no input and returns one output O and its functional specification is written as $O = c$. The only change to the framework is that since c is allowed to be arbitrary, we existentially quantify over c in the synthesis constraint in Equation 4.

8. EXPERIMENTAL RESULTS

In this section, we present an experimental evaluation of our synthesis technique as applied to bitvector program synthesis. We also experimentally compare our technique with other existing techniques that can be used for bitvector program synthesis.

Benchmarks. We selected 25 benchmark examples from the book *Hacker's Delight*, commonly referred to as the Bible of bit twiddling hacks [28].

These examples are described in Figure 5. The benchmarks are organized in increasing order of complexity reflected by the number of lines in the program. 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.

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 bitvector arithmetic. 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.

8.1 Performance of Synthesis Algorithm

Table 1 reports some interesting statistics about the synthesis algorithm (presented in Fig. 4) on the various benchmark examples. 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. 3 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 full verification of safety-critical systems or hardware circuits, and partial verification of general purpose software. In this paper, we show that the same formal reasoning technology for verification 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.

8.2 Comparison with Sketch and AHA

We experimentally compared the implementation of our synthesis technique `Brahma` with two other existing tools for synthesis - Sketch and AHA - on our benchmark suite of 25 examples.

Sketch. The tool Sketch is based on the sketching technique [25, 26] to synthesis. We used the most recent version of Sketch (v1.3.0) for comparison with our technique. For these 25 examples, we expressed the component based design problem as a sketch by defining functions for the base components and encoding the component-based synthesis problem using a variety of encodings, some of which even turned out to be exponential. After consultation with the Sketch team, we chose the best encoding, which is at best a high degree polynomial (as illustrated below). The total runtime of Sketch on the benchmark 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).

We now explain why `Brahma` performs much better than Sketch. At a higher level, the sketching technique is similar to our synthesis technique – 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. To illustrate these differences, we compare the scalability of the two techniques as we increase the number of components in the user-specified library for synthesizing the running example in Table 2. The time taken by Sketch (Col. 3 of Table 2) appears to scale exponentially, while the time taken by `Brahma` (Col. 2 of Table 2) appears to scale non-exponentially as the number of components increases

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

```
$o_1$:=bvsub (x,1)
res:=bvand (x,$o_1$)
```

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

```
$o_1$:=bvadd (x,1)
res:=bvand (x,$o_1$)
```

P3(x) : Isolate the rightmost 1-bit

```
$o_1$:=bvneg (x)
res:=bvand (x,$o_1$)
```

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

```
$o_1$:=bvsub (x,1)
res:=bvxor (x,$o_1$)
```

P5(x) : Right propagate rightmost 1-bit

```
$o_1$:=bvsub (x,1)
res:=bvor (x,$o_1$)
```

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

```
$o_1$:=bvadd (x,1)
res:=bvor (x,$o_1$)
```

P7(x) : Isolate the rightmost 0-bit

```
$o_1$:=bvnot (x)
$o_2$:=bvadd (x,1)
res:=bvand ($o_1$,$o_2$)
```

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

```
$o_1$:=bvsub (x,1)
$o_2$:=bvnot (x)
res:=bvand ($o_1$,$o_2$)
```

P9(x) : Absolute Value Function

```
$o_1$:=bvshr (x,31)
$o_2$:=bvxor (x,$o_1$)
res:=bvsub ($o_2$,$o_1$)
```

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

```
$o_1$:=bvand (x,y)
$o_2$:=bvxor (x,y)
res:=bvule ($o_2$,$o_1$)
```

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

```
$o_1$:=bvnot (y)
$o_2$:=bvand (x,$o_1$)
res:=bvugt ($o_2$,y)
```

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

```
$o_1$:=bvnot (y)
$o_2$:=bvand (x,$o_1$)
res:=bvule ($o_2$,y)
```

P13(x) : Sign Function

```
$o_1$:=bvshr (x,31)
$o_2$:=bvneg (x)
$o_3$:=bvshr ($o_2$,31)
res:=bvor ($o_1$,$o_3$)
```

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

```
$o_1$:=bvand (x,y)
$o_2$:=bvxor (x,y)
$o_3$:=bvshr ($o_2$,1)
res:=bvadd ($o_1$,$o_3$)
```

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

```
$o_1$:=bvor (x,y)
$o_2$:=bvxor (x,y)
$o_3$:=bvshr ($o_2$,1)
res:=bvsub ($o_1$,$o_3$)
```

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

```
$o_1$:=bvxor (x,y)
$o_2$:=bvneg (bvuge (x,y))
$o_3$:=bvand ($o_1$,$o_2$)
res:=bvxor ($o_3$,y)
```

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

```
$o_1$:=bvsub (x,1)
$o_2$:=bvor (x,$o_1$)
$o_3$:=bvadd ($o_2$,1)
res:=bvand ($o_3$,x)
```

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

```
$o_1$:=bvsub (x,1)
$o_2$:=bvand ($o_1$,x)
$o_3$:=bvredor (x)
$o_4$:=bvredor ($o_2$)
$o_5$:=!($o_4$)
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

```
$o_1$:=bvshr (x,k)
$o_2$:=bvxor (x,$o_1$)
$o_3$:=bvand ($o_2$,m)
$o_4$:=bvshl ($o_3$,k)
$o_5$:=bvxor ($o_4$,$o_3$)
res:=bvxor ($o_5$,x)
```

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

```
$o_1$:=bvneg (x)
$o_2$:=bvand (x,$o_1$)
$o_3$:=bvadd (x,$o_2$)
$o_4$:=bvxor (x,$o_2$)
$o_5$:=bvshr ($o_4$,2)
$o_6$:=bvdiv ($o_5$,$o_2$)
res:=bvor ($o_6$,$o_3$)
```

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

```
$o_1$:=bvneg (bveq (x,c))
$o_2$:=bvxor (a,c)
$o_3$:=bvneg (bveq (x,a))
$o_4$:=bvxor (b,c)
$o_5$:=bvand ($o_1$,$o_2$)
$o_6$:=bvand ($o_3$,$o_4$)
$o_7$:=bvxor ($o_5$,$o_6$)
res:=bvxor ($o_7$,c)
```

P22(x) : Compute Parity

```
$o_1$:=bvshr (x,1)
$o_2$:=bvxor ($o_1$,x)
$o_3$:=bvshr ($o_2$,2)
$o_4$:=bvxor ($o_2$,$o_3$)
$o_5$:=bvand ($o_4$,0x11111111)
$o_6$:=bvmul ($o_5$,0x11111111)
$o_7$:=bvshr ($o_6$,28)
res:=bvand ($o_7$,0x1)
```

P23(x) : Counting number of bits

```
$o_1$:=bvshr (x,1)
$o_2$:=bvand ($o_1$,0x55555555)
$o_3$:=bvsub (x,$o_2$)
$o_4$:=bvand ($o_3$,0x33333333)
$o_5$:=bvshr ($o_3$,2)
$o_6$:=bvand ($o_3$,0x33333333)
$o_7$:=bvadd ($o_4$,$o_6$)
$o_8$:=bvshr ($o_7$,4)
$o_9$:=bvadd ($o_8$,$o_7$)
res:=bvand ($o_9$,0x0F0F0F0F)
```

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

```
$o_1$:=bvsub (x,1)
$o_2$:=bvshr ($o_1$,1)
$o_3$:=bvor ($o_1$,$o_2$)
$o_4$:=bvshr ($o_3$,2)
$o_5$:=bvor ($o_3$,$o_4$)
$o_6$:=bvshr ($o_5$,4)
$o_7$:=bvor ($o_5$,$o_6$)
$o_8$:=bvshr ($o_7$,8)
$o_9$:=bvor ($o_7$,$o_8$)
$o_{10}$:=bvshr ($o_9$,16)
$o_{11}$:=bvor ($o_9$,$o_{10}$)
res:=bvadd ($o_{10}$,1)
```

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

```
$o_1$:=bvand (x,0xFFFF)
$o_2$:=bvshr (x,16)
$o_3$:=bvand (y,0xFFFF)
$o_4$:=bvshr (y,16)
$o_5$:=bvmul ($o_1$,$o_3$)
$o_6$:=bvmul ($o_2$,$o_3$)
$o_7$:=bvmul ($o_1$,$o_4$)
$o_8$:=bvmul ($o_2$,$o_4$)
$o_9$:=bvshr ($o_5$,16)
$o_{10}$:=bvadd ($o_6$,$o_9$)
$o_{11}$:=bvand ($o_{10}$,0xFFFF)
$o_{12}$:=bvshr ($o_{10}$,16)
$o_{13}$:=bvadd ($o_7$,$o_{11}$)
$o_{14}$:=bvshr ($o_{13}$,16)
$o_{15}$:=bvadd ($o_{14}$,$o_{12}$)
res:=bvadd ($o_{15}$,$o_8$)
```

Figure 5: Benchmark Examples. The functions used in the examples have the usual semantics defined in SMTLIB QF_BF logic [2].

Benchmark		Brahma		Sketch	ratio	AHA
Id	#lines	Iter.	runtime	runtime	Sketch/	time(sec)
			sec	sec	Brahma	[#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

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).

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.

While the size of the constraints generated by our technique is provably quadratic in the number of components, experimental evidence indicates that the size of the constraints generated by the sketching technique is either exponential or a high degree polynomial in the number of holes or components. This is also illustrated in Table 2 that shows 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. The succinctness of our constraint is because of our non-trivial encoding that exploits the modular specifications of the components, and it helps us relegate the inherent exponential reasoning to the underlying SMT solvers.

It may be tempting to 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`.

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.

AHA. The AHA tool [4] is a superoptimizer, endorsed by our benchmark book [28] as *A Hacker's Assistant*. It is based on an idea by Henry Massalin [20], 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 potential 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.

8.3 Choice of Set of Base Components

We now discuss the strategy that we used for choosing the set of base components for our benchmark examples. Picking the set of base components is the only step in our approach that currently requires human guidance, although even it is partly automated.

In our experiments, we started with a common set of base components, referred to as *the standard library*, for synthesizing programs for each of the benchmark examples. The standard library included 12 components 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*. This is similar to many library driven programming languages such as Java and Ocaml which have standard library functions. If a program requires functions outside the standard library, the user has to select the appropriate libraries to include. Similarly, in our technique, programmer specifies the extended library if the standard library is not sufficient for the synthesis of the program. This also facilitates the hierarchical design of programs where the user can specify a synthesized program as a new component in a new synthesis problem.

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.

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 to synthesize a desired specification, `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 added in an incremental way 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.

9. RELATED WORK

Counterexample Guided Inductive Synthesis. Inductive synthesis refers to 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 [22]. The inductive approach [21, 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 [26] and discrete event systems [7].

We have recently used the encoding presented in this paper to solve a different component-based synthesis problem wherein logical specification of the desired program is replaced by an input-output oracle [12]. The synthesis approaches in the two papers are significantly different – [12] uses only the encoding of the synthesis problem as an $\exists\forall$ formula that is presented in (and is the contribution of) this paper, but not the $\exists\forall$ solving strategy.

Automated API Composition. The Jungloid mining tool [18] 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` [13] 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 the work on automatic program synthesis [19, 27], where a theorem prover was used instead of an AI planner. Our approach does not use abstract predicates and axioms and relies on the predicates provided by the SMT solver. The SMT solver also reasons about the implicit axioms using decision procedures.

Sketching. System development requires both algorithmic insights as well as careful attention to details. *Sketching* [25] requires a developer to come up with the algorithmic insight and uses the sketch compiler to fill in missing details by using principle of counterexample guided inductive synthesis. This allows the sketch technique to be quite general, and applicable to discovering small unknown details in a variety of programs [26, 23, 24]. 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 application domain since key hardness in synthesis of these programs is to come up with the algorithmic insight.

Super-optimizers. Superoptimization is the task of finding an optimal code sequence for a straight-line target sequence of instructions, and has shown to be useful 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 [20]. Another approach has been to constrain the search space to a set of equality-preserving transformations expressed by the system designer [14] 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 [20] used them for verification of candidate synthesized programs. SAT solvers are also used in Sketching [26] 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. We only require that synthesis constraints generated by our technique be solvable by an SMT solver.

10. CONCLUSION AND FUTURE WORK

Automated synthesis has the potential to revolutionize system development process. Up until now, automating synthesis was beyond the realm of practicality. However, huge engineering advances in logical reasoning have significantly changed the landscape. It has enabled verification of large systems, and in this paper, we show that synthesis requires resources within one order of magnitude of the resources required for verification.

Most recent work on automated synthesis is based on the philosophy that synthesis can be automated only if it is partially aided by humans. We demonstrate that, using a combination of the modularity principle, SMT solver, nontrivial encoding of synthesis as constraint solving, and refined constraint-solving approaches, human intervention can be eliminated. This is especially true for specific domains, such as bitvector algorithms, where algorithms are not intuitive and human guidance is a hindrance, rather than a help, to automated synthesis.

Our formulation of the component-based synthesis problem and our solution are both more widely applicable, and this exploration is left for future work. There are possible generalizations to synthesizing programs with richer control structure, such as loops and recursion, and to 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 are given at higher levels of abstraction.

Acknowledgment. We thank members of the Sketch team (Ras Bodik, Armando Solar-Lezama, Lexin Shan, Nicholas Tung) and Rishabh Singh who helped us with the Sketch tool.

11. 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. www.hackersdelight.org/aha.zip,/aha.pdf, 2008.

[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] B. Brandin, R. Malik, and P. Malik. Incremental verification and synthesis of discrete-event systems guided by counterexamples. *IEEE CST*, 12(3), 2004.

[8] 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.

[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. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010 (to-appear).

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

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

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

[16] N. Linial, Y. Mansour, and N. Nisan. Constant depth circuits, fourier transform, learnability. In *FOCS*, '89.

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

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

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

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

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

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

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

[24] A. Solar-Lezama, C. G. Jones, and R. Bodík. Sketching concurrent data structures. In *PLDI*, 2008.

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

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

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

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