Toward Tool Support for Interactive Synthesis

Shaon Barman[1] Rastislav Bodik[1] Satish Chandra[2] Emina Torlak[3]
Arka Bhattacharya[1] David Culler[1]

[1]University of California, Berkeley [2]Samsung Research [3]University of Washington

Abstract

Syntax-guided synthesis searches for an implementation of a given specification by exploring large spaces of candidate programs. Sketches reduce these search spaces, making synthesis more tractable, by predefining the structure of the desired implementation. Typically, this structure is obtained through human insight—this paper introduces a method for interactive, tool-supported discovery of such structure. The key idea is to decompose the specification into subcomputations such that the decomposition dictates the sketch. We rely on a readily obtainable specification that is nothing more than a finite set of sample input-output pairs or execution traces of the desired program. We introduce two complementary decomposition operators and demonstrate them on case studies. We find that our interactive methodology to discover structure extends the reach of computer-aided programming to problems that cannot be solved with synthesis alone.

Categories and Subject Descriptors F.3.1 [*Specifying and Verifying and Reasoning about Programs*]: Specification techniques

General Terms Theory, Languages, Algorithms

Keywords Specifications, relational algebra, refinement, decomposition

1. Introduction

Background Program synthesis enables a high-level approach to programming—the programmer provides a specification of the desired implementation, and a synthesis tool automatically turns this specification into a correct implementation. The specification can take many forms, from a set of input-output examples [14] to a logical formula [2] to a reference implementation [34]. Given such a specification, a synthesizer produces an implementation that is verifiably correct, often by searching a space of candidate programs from the target implementation language (see, e.g., [2, 14, 15, 31]). Such unrestricted search frees the programmer from having to provide hints to the synthesizer, but it inherently limits the size of a program that can be generated (to a few tens of instructions in contemporary systems [15, 31]).

To illustrate, consider the toy problem of synthesizing the sign function, given the following reference program as the specification:

```
def sign_spec(x):
  return (x == 0) ? x : x / abs(x)
```

Our goal is to obtain an implementation that avoids the division and absolute-value operations, as specified below:

```
def sign_impl(x) :
  return intExpr[x] // desired program: (x == 0) ? 0 : (x < 0) ? −1 : 1

grammar intExpr[identifier ...]:
  intExpr = identifier ... | constant | boolExpr ? intExpr : intExpr
  constant = 0 | 1 | −1
  boolExpr = intExpr boolOp intExpr
  boolOp = <= | < | > | >= | ==
```

The desired implementation is a program with an abstract syntax tree of depth three, requiring three derivation steps from the intExpr grammar. To find this program, a synthesizer will need to search the space of all programs of depth three or less—which includes roughly 131 million candidates.

One approach to reducing the size of this search space is to supply the synthesizer with a *sketch* [4, 34], a partial implementation that outlines the structure but not the details of the desired computation. By spelling out the structure of the computation, sketches can exponentially reduce the search space and even decompose the problem into smaller independent problems. For example, the following sketch for our toy synthesis problem reduces the search space from 131×10^6 to 6.4×10^3 candidates:

```
def sign_sketch(x):
  return boolExpr[x]:
    return 0
  else if boolExpr[x]:
    return −1
  else:
   return 1
```

```
grammar boolExpr[identifier ...]:
  boolExpr = intExpr boolOp intExpr
  boolOp = <= | < | > | >= | ==
  intExpr = identifier ... | constant
  constant = 0 | 1 | −1
```

Thanks to its space-reducing power, sketching has enabled practical synthesis for many application domains, including dynamic programming algorithms [29], stencil computations [35], database programming [8], and automatic bug fixing of student programming assignments [33].

Problem But where do sketches like sign_sketch come from? Typically, the process of sketch construction is entirely manual—the user or the designer of a synthesis tool has an insight about the structure of the desired computation(s) and expresses that insight with a partial implementation. In this paper, we introduce a tool-assisted method for obtaining a sketch from a specification. Our method starts with a specification and performs programmer-guided discovery of the structure of the computation that is typically expressed in a sketch. Formally, the method decomposes the specification into its subcomputations; the structure of the decomposition reveals the structure of the sketch, while the subcomputations, in turn, define the specifications of the holes (i.e., the missing details) in the sketch.

Approach Our approach relies on *concrete specifications*, which describe desired program behaviors with a finite set of input-output pairs or execution traces. Such a set of values forms a finite relation, and we analyze a concrete specification by decomposing its relational representation. For example, the following relation is a concrete specification of the input-output behavior of sign_spec on all 3-bit inputs:

x	$sgn(x)$
-4	-1
-3	-1
-2	-1
-1	-1
0	0
1	1
2	1
3	1

Concrete specifications such as this one are easy to obtain from reference implementations, from logical specifications, or directly from the programmer.

We decompose concrete specifications with two complementary operators, which form the basis of our interactive methodology for deriving the structure of a sketch. One operator decomposes the entire relation into a cross product of smaller relations (fewer columns in each smaller relation). The other decomposes the relation into a union of smaller subrelations (fewer rows in each subrelation) such that each subrelation can itself be factored into a cross-product of smaller relations. For example, the concrete specification for sign_spec decomposes into three subrelations, each consisting of two independent subcomputations:

$$\begin{array}{|c|}\hline x \\ \hline -4 \\ -3 \\ -2 \\ -1 \\ \hline \end{array} \times \begin{array}{|c|}\hline sgn(x) \\ \hline -1 \\ \hline \end{array} \quad \cup \quad \begin{array}{|c|}\hline x \\ \hline 0 \\ \hline \end{array} \times \begin{array}{|c|}\hline sgn(x) \\ \hline 0 \\ \hline \end{array} \quad \cup \quad \begin{array}{|c|}\hline x \\ \hline 1 \\ 2 \\ 3 \\ \hline \end{array} \times \begin{array}{|c|}\hline sgn(x) \\ \hline 1 \\ \hline \end{array}$$

This decomposition translates directly into the structure of the sign_sketch sketch—the desired computation is a case analysis with three distinct cases, and each case is an independent (constant) function over a subset of the input values. Our interactive methodology for sketching exploits the ability of the two operators to uncover case structure and independent subcomputations from a specification.

Interactive methodology We describe a step-by-step process for defining concrete specifications, for analyzing them, and for translating the results of the analysis into a sketch, which describes the desired modularized synthesis problem. Some steps are fully automated, while others are interactive, requiring the user to pose a hypothesis, not unlike in debugging or other forms of interactive problem solving. We describe two variants of our interactive method, which are *duals* of each other.

The first method starts with a *precise* concrete specification that includes exactly the desired behaviors of the program. A precise concrete specification can be thought of as a full functional specification of the program. Following our first method, the user decomposes the specification, leading to a structured sketch which is then completed by an off-the-shelf synthesis tool. This method can also be used to understand a concrete specification produced by a black-box computation, even when the sketch itself is not desired.

The second method starts with a sketch and a *partial* concrete specification that defines all acceptable behaviors of a program. When some of these acceptable behaviors are more desirable than others (*e.g.,* some can be implemented with a deterministic program while others cannot), the second method helps the user refine the partial specification to obtain a maximal set of desirable behaviors—that is, a precise specification. An off-the-shelf synthesizer can then complete the sketch to satisfy only the desirable behaviors.

Contributions In summary, this paper makes the following contributions:

- The notion of concrete specifications, which unifies the various notions of value-based specifications (*e.g.,* example-based [14, 21] or trace-based [12, 13, 39]) that have been proposed in previous work.
- Two decomposition operators that reveal the structure of a computation expressed as a concrete specification. We support these operators with efficient algorithms, which scale to thousands of behaviors.
- Two interactive methods for using our decomposition operators to derive sketches from precise specifications and to refine partial specifications into precise ones.
- Three case studies that demonstrate our methodology and the scalability of the supporting algorithms. The problems

tackled in the studies are all difficult or impossible to complete without decomposition.

Outline The rest of the paper is organized as follows. We first present our interactive methodology and illustrate its application to synthesis-based program deobfuscation (Sec. 2). We then describe a theory of lossless decomposition which underlies our methodology (Sec. 3) and present the algorithms for executing these decomposition operators (Sec. 4). We illustrate our methodology on three case studies, applying it to deobfuscation, parsing-by-demonstration and angelic programming (Sec. 5). The paper concludes with a discussion of related work (Sec. 6) and a brief summary of contributions (Sec. 7).

2. Overview

To illustrate our interactive synthesis methodology, consider the problem of synthesizing a deobfuscated version of the toy program in Fig. 1a. That is, we wish to understand what the program is computing and synthesize a functionally equivalent but understandable implementation. While this program is artificial, one can imagine performing the same steps to translate legacy assembly code into a modern language or a minimized JavaScript code snippet to a program that elucidates the webpage functionality.

The program $\mathsf{toy_k}$ takes as input two signed k-bit values and produces a $2k$-bit output. Given only this program as a reference, we want to find a more readable program that performs the same computation. To accomplish this goal, we will develop a sketch of the readable program and use syntax-guided synthesis to fill in the missing expressions. We show how to develop such a sketch by decomposing a concrete specification for $\mathsf{toy_k}$ with our interactive approach.

2.1 Example: Sketching Programs for Deobfuscation

An easy way to deobfuscate $\mathsf{toy_k}$ is to create a sketch [34] of a simpler implementation and then use program synthesis to complete the sketch automatically. A sketch is a partially implemented program containing "holes" to be filled with expressions. A program synthesizer searches for expressions that fill the holes correctly—in our case, the completed sketch must be functionally equivalent to $\mathsf{toy_k}$.

The simplest sketch consists of a single hole, to be filled with an expression from a grammar of all possible programs, such the one shown in Fig. 1b. Using operators found in the original program, the sketch gives grammars for expressions, predicates and statements. It then goes on to define the desired program $\mathsf{T_k}$ to be a sequence of guarded statements. While this sketch is expressive enough to capture the computation, its generality poses two problems. First, it induces a search space that is too large for a synthesizer to explore efficiently. Second, it places insufficient constraints on the syntactic form of candidate programs—even if a solution were found, it may be as complex as the original program.

To make the search tractable, and the resulting program syntactically simple, the sketch needs to be sufficiently detailed. Ideally, it should include a breakdown of the deobfuscated implementation into procedures and an outline of each procedure's control structure. For example, Fig. 1c shows one such sketch, and Fig. 1d shows a completion of this sketch. But going from the original program $\mathsf{toy_k}$ to the sketch in Fig. 1c is nontrivial. We propose a way of taking a specification and understanding the inherent structure required to compute that specification. The programmer can then use the results of this analysis to write a sufficiently detailed sketch that a synthesizer can complete.

2.2 Concrete Specification

Most specification analyses (*e.g.,* [9, 10, 28]) work on logical specifications that describe acceptable program behaviors implicitly, as formulas. Such specifications are expressive, succinct, and amenable to algebraic reasoning and symbolic solving. But they are also rarely available in practice.

We focus on analyzing *concrete specifications*, which can easily be obtained in practice. A concrete specification describes the set of acceptable *concrete behaviors* of a program explicitly, as tuples in a database relation. These tuples consist of concrete values and represent, for example, traces of program states or legal input-output pairs. As such, they can be observed from a reference implementation, extracted from a test suite, provided directly by the programmer, or enumerated from a logical specification.

Unlike logical specifications, concrete specifications can only describe finite sets of acceptable behaviors. They are therefore rarely complete descriptions of a computation—unless the computation is a finite function, a concrete specification is an underapproximation of its full set of behaviors. These underapproximate descriptions, however, still capture useful properties that can help the programmer arrive at a desired implementation. For example, given a set of program traces, dynamic invariant detection [12] can discover likely invariants of the underlying computation by inferring properties over program variables that hold for every trace. The resulting properties can then be used to automatically repair errors [27].

Fig. 2 shows a concrete specification T for our example program $\mathsf{toy_k}$, obtained by recording the output of $\mathsf{toy_k}$ on all pairs of signed 2-bit values (*i.e.,* $x, y \in [-2, 1]$). Each behavior (row) in T consists of the bits comprising one input/output triplet: $t_3t_2t_1t_0 = \mathsf{toy_2}(x_1x_0, y_1y_0)$. This low-level representation of behaviors enables us to discover the subcomputations of $\mathsf{toy_k}$, if any, relating the individual bits of input and output.

2.3 Lossless Decomposition

We discover structure in concrete specifications with the help of two complementary *lossless decomposition* operators (Sec. 3). *Lossless product decomposition* (LPD) finds the best way to decompose a specification relation into a cross

```
def toy_k(x, y):
  t1 = k − 1
  t2 = x >> t1
  t3 = − x
  t4 = t3 >> t1
  t5 = − t4
  t6 = t2 | t5
  t7 = −1 << k
  t8 = ~t7
  t9 = t6 & t8
  t10 = y << k
  t11 = t9 | t10
  t12 = 1 << k
  t13 = t11 + t12
  t14 = t7 << k
  t15 = ~t14
  t = t15 & t13
  return t
```

(a)

```
// grammar of simple expressions
grammar expr[id]:
  expr = id | lit | expr op expr | uop expr
  lit = integer literal
  op = + | − | ∗ | << | >> | & | |
  uop = ~

// grammar of simple predicates
grammar pred[id]:
  pred = expr[id] op expr[id]
  op = < | <= | ==

// grammar for statements
grammar statement[id]:
  statement = expr | id = expr | return expr

// grammar for guarded statements
grammar statements[id]:
  statements = {if pred[id]: statement[id]}∗

// sketch of the desired program
def T_k(x, y):
  statements[x, y]
```

(b)

```
def H_k(y):
  return expr[y] << k

def L_k(x):
  return expr[x]

def T_k(x, y):
  // low order k-bit mask
  low = ~(−1 << k)
  // high order k-bit mask
  high = low << k
  // combine L and H
  return L_k(x) & low | H_k(y) & high
```

(c)

```
def H_k(y):
  return (y+1) << k

def L_k(x):
  if x < 0:
    return −1
  if x > 0:
    return 1
  return 0

def T_k(x, y):
  // low order k-bit mask
  low = ~(−1 << k)
  // high order k-bit mask
  high = low << k
  // combine L and H
  return L_k(x) & low | H_k(y) & high
```

(d)

Figure 1: An obfuscated program (a). To understand this program, we would like to synthesize a program from a sketch (b), which defines a space of possible programs. But to scale sketching to larger problems, we instead need to provide a structured sketch (c). Solving a refined version of the sketch (c) leads us to the desired program (d).

x_1	x_0	y_1	y_0	t_3	t_2	t_1	t_0
1	0	1	0	1	1	1	1
1	0	1	1	0	0	1	1
1	0	0	0	0	1	1	1
1	0	0	1	1	0	1	1
1	1	1	0	1	1	1	1
1	1	1	1	0	0	1	1
1	1	0	0	0	1	1	1
1	1	0	1	1	0	1	1
0	0	1	0	1	1	0	0
0	0	1	1	0	0	0	0
0	0	0	0	0	1	0	0
0	0	0	1	1	0	0	0
0	1	1	0	1	1	0	1
0	1	1	1	0	0	0	1
0	1	0	0	0	1	0	1
0	1	0	1	1	0	0	1

Figure 2: Concrete specification T for $\mathsf{toy_k}$ in Fig. 1a, obtained by applying $\mathsf{toy_2}$ to all pairs of 2-bit inputs and recording the output

$T =$

L			
x_1	x_0	t_1	t_0
1	0	1	1
1	1	1	1
0	0	0	0
0	1	0	1

$\times$

H			
y_1	y_0	t_3	t_2
1	0	1	1
1	1	0	0
0	0	0	1
0	1	1	0

Figure 3: An LPD for the concrete specification in Fig. 2. This decomposition inspired the structured sketch in Fig. 1c.

product of smaller relations, each of which represents a concrete specification of an independent subcomputation. If a relation has no independent subcomponents, our second operator, *lossless union-of-products decomposition* (LUPD), can be used to restructure it into a union of decomposable *refinements*—that is, subsets of the original specification. This union consists of all maximal specification refinements that can be expressed as products of smaller relations. Together, these two operators enable us to discover independence and case structure of a concrete specification (and its underlying computation), as shown next.

Independence Analysis with LPD Figure 3 shows the LPD decomposition of our example specification T (Fig. 2), which exposes two independent subcomputations in $\mathsf{toy_k}$ (Fig. 1a). In particular, LPD infers that T can be decomposed into two smaller relations, L and H, whose cross-product yields the behaviors of T. This decomposition proves that the computations described by L and H are fully separable—an arbitrary choice of a behavior from L, which computes the low-order bits of the output from x, can be combined with an arbitrary behavior from H, which computes the high-order bits of the output from y, to obtain a legal behavior in T. We can therefore implement L and H with two independent procedures and combine their results.

The sketch in Fig. 1c captures this independence structure. The bodies of the two procedures, $\mathsf{L_k}$ and $\mathsf{H_k}$, contain one hole each that, we hypothesize, can be filled with simple arithmetic expressions. An off-the-shelf synthesizer [36] validates this hypothesis for $\mathsf{H_k}$ and its specification H in seconds, replacing the hole with the expression $\mathsf{y + 1}$. Indeed, it is easy to see that H specifies addition-by-1 for signed 2-bit numbers: $t_3t_2 = y_1y_0 + 1$. But no simple expression is found for $\mathsf{L_k}$ and L, indicating that $\mathsf{L_k}$ needs further refinement.

Case Analysis with LUPD The computation described by L lacks independent subcomputations in that $\text{LPD}(L) = L$. The LUPD operator provides a way to decompose a specification like L into a union of (possibly overlapping)

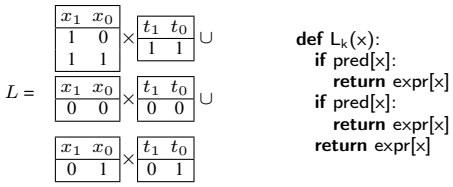

Figure 4: The LUPD of the concrete specification L from Fig. 3 and the refined sketch for the function $\mathsf{L_k}$ from Fig. 1.

refinements that satisfy a given independence hypothesis. These hypotheses state that certain parts of the specification should be decomposable from each other. For example, Fig. 4 shows the LUPD decomposition of L, a union of three maximal refinements in which the output bits are independent from the input bits. This decomposition reveals that we can compute L with a compact case-analysis on the value of x. In particular, the number of cases to be considered (three) is smaller than the number of all (four) possible values that x can take.

The sketch in Fig. 4 captures this case structure. We hypothesize that each case corresponds to a simple arithmetic expression. This time, the synthesizer completes the sketch in seconds, producing an implementation of the sign function (as shown in Fig. 1d).

Generalizing Analysis Results Given that we performed the analysis and sketching on a concrete specification for 2-bit inputs, it is natural to ask whether these results generalize to larger input spaces. We confirmed that they do. Repeating the analyses on concrete specifications of $\mathsf{toy_4}$, for example, yields analogous decompositions to those we have seen for $\mathsf{toy_2}$. Similarly, the implementation of $\mathsf{L_k}$ and $\mathsf{H_k}$, as well as the overall deobfuscated program, are general: $\mathsf{toy_k}(x, y) = \mathsf{T_k}(x, y)$ for all $k \leq 32$. We speculate that this generalizability effect—which also appears in our case studies (Sec. 5)—is due to the capacity of concrete specifications to capture the essential structure of the underlying computation, which remains constant as the problem scales.

Dual Problem In the $\mathsf{toy_k}$ example, we used our decomposition operators to interactively derive a structured sketch from a precise specification. We introduce a dual problem, in which the programmer starts with a structured sketch and a *partial* concrete specification that contains acceptable but undesirable behaviors. The goal in this problem is to strengthen the specification to a desirable subset of the original tuples, so that the holes can be synthesized according to this stronger speculation. In Sec. 5.3, we show how the programmer can use the LPD and LUPD operators to find this desired subset.

3. A Theory of Lossless Decomposition

In this section, we formalize the notion of concrete specifications and present our theory of specification decomposition. We start with the simpler of the two operations, lossless product decomposition (LPD), and show that, while a specification may have many LPDs, it has a unique *finest* LPD. As such, the finest LPD reveals the best inherent decomposition of a specification into mutually independent components. Intuitively, this step decomposes the original problem into smaller components that can be represented as separate holes in the sketch, and possibly be individually synthesized.

If this best decomposition is still too coarse for a given application, we show how to express the specification, with the help of lossless union-of-products decomposition (LUPD), as a union of stricter specifications (technically, *refinements* of the original specification), each of which exhibits the finest LPD of the desired granularity. The LUPD of a specification, like its finest LPD, is unique. This decomposition allows the programmer to learn about and validate the structure of the sketch. By adjusting the granularity of each refinement's LPD the programmer can trade off the complexity of each refinement with the total number of refinements, which corresponds to trading off complexity of each hole in the sketch with the complexity in the structure of the sketch.

3.1 Concrete Specifications

We represent specifications as database relations with set semantics (Def. 1). A relation is a set of tuples that map attributes to values. Each tuple represents a valid program *behavior*, and each attribute describes a distinct aspect of that behavior (*e.g.,* the value of a variable at a specific point in an execution). We display relations as tables, with rows representing tuples and column names representing attributes.

Definition 1 (Relation)**.** *A* relation R *is a finite set of* tuples*, defined over a finite set of* attributes*. A tuple is a function from the relation's attributes, denoted by* $attr(R)$*, to values of any type. An attribute is a name drawn from an infinite set of identifiers. We view tuples both as functions and as sets of attribute-value pairs.*

Despite its simplicity, our notion of concrete specifications is general enough to accommodate all forms of finite descriptions of program behaviors. In Sec. 2, we saw an example (Fig. 2) of a concrete specification whose behaviors represent valid input / output pairs for a program. But behaviors can also represent execution traces (Sec. 5.3) or even just program outputs (Sec. 5.2). As long as the descriptions of individual behaviors are finite, it is easy to represent them as a relation over the same set of attributes: we define each tuple to map irrelevant attributes (for which the represented behavior has no value) to a distinguished bottom value.

3.2 Lossless Product Decomposition (LPD)

Lossless product decomposition (LPD) breaks a specification into a set of relations that yield the original specification when combined with (relational) cross-product (Def. 2). A specification may have many LPDs. For example, the specification in Fig. 2 has two LPDs: the specification itself (*i.e.,* the trivial LPD) and the LPD shown in Fig. 3.

$$\{\{x_1, x_0, t_1, t_0\}, \{y_1, y_0, t_3, t_2\}\}$$

Figure 5: The LAP for the LPD in Fig. 3.

Definition 2 (Lossless Product Decomposition)**.** *A set of relations $P = \{P_1, \ldots, P_k\}$ is a* lossless product decomposition *(LPD) of a relation R iff $R = P_1 \times \ldots \times P_k$ and $attr(P_i) \cap attr(P_j) = \emptyset$ for all $i \neq j$. We define the cross-product of two relations in the usual way: $P_i \times P_j = \{t_i \cup t_j \mid t_i \in P_i \wedge t_j \in P_j\}$.*

For small specifications, such as the toy example in Fig. 1, it is easy to write down and examine an LPD, but for larger examples this becomes unwieldy. We therefore introduce a more compact formulation (Defs. 3-4) of the same concept, which we call *lossless attribute partition* (LAP). An LAP is a partition of a relation's attributes that corresponds to an LPD. Fig. 5 shows the LAP for the LPD in Fig. 3.

Definition 3 (Attribute Partition)**.** *A set of attribute sets $A = \{A_1, \ldots, A_k\}$ is an* attribute partition *for a relation R iff $attr(R) = A_1 \cup \ldots \cup A_k$ and $A_i \cap A_j = \emptyset$ for all $i \neq j$.*

Definition 4 (Lossless Attribute Partition)**.** *An attribute partition $A = \{A_1, \ldots, A_k\}$ for R is* lossless *iff $\{\Pi_{A_1} R, \ldots, \Pi_{A_k} R\}$ is an LPD of R. The operator Π stands for relational projection, where $\Pi_{A_i} R = \{\bigcup_{a \in A_i} \langle a, t(a) \rangle \mid t \in R\}$.*

LAPs and LPDs are equivalent formulations of the same concept in that one uniquely determines the other. We can obtain the LAP for an LPD $P = \{P_1, \ldots, P_k\}$ by applying the $attr$ function to each relation in P: $\{attr(P_1), \ldots, attr(P_k)\}$. Similarly, we can obtain the LPD from an LAP $A = \{A_1, \ldots, A_k\}$ of R by projecting R onto each set in A: $\{\Pi_{A_1} R, \ldots, \Pi_{A_k} R\}$. In the rest of this paper, we will use the two formulations interchangeably.

While a relation can have many LAPs—one for each LPD—it has a unique *finest* LAP (Def. 5, Thm. 1) and, correspondingly, a unique finest LPD. The finest LAP and LPD for our toy example are shown in Figs. 5 and 3. In general, the finest LAP for a specification is the finest-grained partition of a relation's attributes that is also an LAP. All other LAPs can be obtained from the finest LAP by combining its parts with set union ($\cup$) to form coarser attribute partitions. When ordered by the standard partition refinement relation $\sqsubseteq$ (Def. 5), the LAPs for a relation form a lattice, with the finest LAP as the bottom element. Our LPD decomposition operation therefore returns the finest LAP / LPD as the best (most informative) decomposition of a given relation.

Definition 5 (Finest LAP)**.** *An LAP A for a relation R is a* finest *LAP iff R has no LAP B such that $B \neq A$ and $B \sqsubseteq A$. We use the standard definition of partition refinement: $B \sqsubseteq A$ iff $\forall B_i \in B.\ \exists A_j \in A.\ B_i \subseteq A_j$.*

Theorem 1 (Uniqueness of the Finest LAP)**.** *Every relation R has a unique finest LAP.*

Proof. Suppose that a relation R has two distinct finest LAPs, A and B. If $A \sqsubseteq B$ or $B \sqsubseteq A$, we arrive at a contradiction. If $A \not\sqsubseteq B$ and $B \not\sqsubseteq A$, then there must be two parts $A_i \in A$ and $B_j \in B$ such that $A_i \neq B_j$ and $A_i \cap B_j \neq \emptyset$. Let $C = A_i \cap B_j$, $A'_i = A_i \setminus C$ and $B'_j = B_j \setminus C$. Because A is an LAP for R, it follows from Defs. 2-4 that $\{A_i, A \setminus A_i\}$ is also an LAP for R. Consequently, we have that $R = (\Pi_{A_i} R \times \Pi_{A \setminus A_i} R) = (\Pi_{A'_i \cup C} R \times \Pi_{A \setminus A_i} R)$. Given this equality and the fact that $A'_i \cup C$ and $A \setminus A_i$ are disjoint, we can use the definitions of projection and cross-product to derive the following: $\Pi_{B_j} R = \Pi_{B_j}(\Pi_{A'_i \cup C} R \times \Pi_{A \setminus A_i} R) = (\Pi_{B_j} \Pi_{A'_i \cup C} R) \times (\Pi_{B_j} \Pi_{A \setminus A_i} R) = \Pi_{B_j \cap (A'_i \cup C)} R \times \Pi_{B_j \cap (A \setminus A_i)} R = \Pi_C R \times \Pi_{B'_j} R$. This shows that B_j could be decomposed into two finer parts, so B could not have been a finest LAP. □

3.3 Lossless Union of Products Decomposition (LUPD)

Many concrete specification relations are only trivially decomposed by the finest LAP. An example of this is the relation L in Fig. 3, which has $\{attr(L)\}$ as its finest LAP. To obtain a better (finer) decomposition for a relation like L, we turn to lossless union-of-products decomposition (LUPD) (Defs. 6-8, Thm. 2).

LUPD enables the programmer to see all maximal subsets—or, *refinements*—of a relation R that have a specific, desirable attribute partition A as an LAP. The union of these refinements, which we call *maximal product components* (MPCs), is equal to R, and each is maximal in that it cannot be augmented with any more tuples from R while continuing to have A as an LAP. Together, the MPCs comprise all possible ways to refine R into stricter specifications that are themselves decomposable according to A.

When deobfuscating the toy example in Fig. 1, we used LUPD to find all refinements of L (Fig. 4) that decouple the input and output bits. We expressed this property of the desired refinements by applying the LUPD operation to L and the attribute partition $A = \{\{x_1, x_0\}, \{t_1, t_0\}\}$. All of the resulting refinements have A as an LAP, and, as such, they all specify functions in which the output bits are independent from the inputs. Because this set of refinements is exhaustive, we know that it fully captures the distinct "cases" in the computation—if all of the refinements are implemented separately and combined with a case statement, no behaviors will be lost.

Definition 6 (Product Component)**.** *A relation Q is a* product component *of a relation R w.r.t. an attribute partition A, denoted by $PC(Q, R, A)$, iff $Q \subseteq R$ and A is an LAP for Q.*

Definition 7 (Maximal Product Component)**.** *A product component Q of a relation R w.r.t. an attribute partition A is* maximal*, denoted by $MaxPC(Q, R, A)$, iff there is no relation S such that $PC(S, R, A)$ and $Q \subset S$.*

Definition 8 (Lossless Union-of-Products Decomposition)**.** *A set of relations $P = \{P_1, \ldots, P_k\}$ is the* lossless union-

```
COMPUTELAP(R)
1  A ← {}
2  rest ← attr(R)
3  while rest ≠ ∅ do
4    a ← choose(rest)
5    B ← PARTITION(a, Π_rest R)
6    rest ← rest \ B
7    A ← A ∪ {B}
8  return A

PARTITION(a, R)
1  B ← {a}
2  W ← WITNESS(B, R)
3  while W ≠ ∅ do
4    B ← B ∪ W
5    W ← WITNESS(B, R)
6  return B

WITNESS(B, R)
1  X ← Π_B R
2  Y ← Π_(attr(R)\B) R
3  if (X × Y) = R then
4    return {}
5  t ← choose((X × Y) \ R)
6  x ← π_B t
7  y ← π_(attr(R)\B) t
8  W ← (attr(R) \ B)
9  for y' ∈ Y s.t. x ∪ y' ∈ R do
10   W' ← {a | y(a) ≠ y'(a)}
11   if |W'| < |W| then
12     W ← W'
13 return W
```

Figure 6: Algorithm to find the finest LAP for a relation R.

of-products decomposition *(LUPD) of a relation R w.r.t. an attribute partition A iff* $\forall P_i \in P.\ MaxPC(P_i, R, A)$ *and* $\forall Q.\ MaxPC(Q, R, A) \implies Q \in P$.

Theorem 2 (Uniqueness and Completeness of the LUPD)**.** *Every relation R has a unique LUPD with respect to a given attribute partition A, and this LUPD is complete in that* $R = \bigcup_{P \in \text{LUPD}(R,A)} P$.

Proof. The proof follows directly from Def. 8. □

4. Computing Lossless Decompositions

To automate lossless decomposition of relations, we have designed two efficient algorithms for answering LAP and LUPD queries. The COMPUTELAP algorithm finds the finest lossless attribute partition, and the COMPUTELUPD algorithm enumerates all maximal product components of a relation induced by a given attribute partition. Both algorithms require the input relation to be finite. We describe the algorithms in detail in the rest of this section.

4.1 Computing the Finest LAP

We compute the finest LAP for a given relation R using the algorithm in Fig. 6. The top-level procedure, COMPUTELAP, is straightforward. Line 1 initializes the variable A, which holds the constituent parts of the decomposition, to the empty set; line 2 initializes $rest$, which holds the unpartitioned attributes, to the set of all attributes of R. The main loop then computes A by repeatedly choosing some attribute a that has not yet been assigned to a part (line 4); finding the the part B that contains a (line 5); and updating $rest$ to exclude, and A to include, B (lines 6-7).

The key step in the algorithm—finding the part B that contains a given attribute—is performed by the procedures PARTITION and WITNESS. Given an attribute a and a relation R such that $a \in attr(R)$, PARTITION computes the smallest such part for a, with respect to R, as follows. We first hypothesize that a is in a set B of its own (line 1). This hypothesis is then tested by invoking WITNESS on B and R (line 2). The WITNESS procedure, as explained below, returns the empty set if R can be expressed as a cross product of $\Pi_B R$ and $\Pi_{attr(R)\setminus B} R$. Otherwise, it returns some, but not necessarily all, attributes that belong in B. Because the set of attributes returned by WITNESS may be incomplete, the main loop of PARTITION (lines 3-5) keeps expanding B with WITNESS attributes until $\{B, attr(R) \setminus B\}$ comprises an LAP for R. We show below that WITNESS returns no extraneous attributes—only the attributes that *must* be in B are returned.

The WITNESS procedure works by first checking if $\{B, attr(R) \setminus B\}$ already comprises an LAP for R. Lines 1-3 implement this check as a straightforward application of Def. 3. If the check succeeds, the procedure returns the empty set (line 4). Otherwise, we choose (line 5) some tuple t *not* in R, and split it into x and y such that x is in the projection of R onto B and y is in the projection of R onto the complement of B. We use $\pi_B t$ to denote the projection of a single tuple t onto a set of attributes B. The chosen tuple is a witness that B must contain some attributes in B's complement. The rest of the procedure (lines 9-13) collects and returns these attributes, which comprise the *smallest* subset of B's complement that the witness $x \cup y$ maps differently than the *valid* tuple $x \cup y'$.

To see that any non-empty set returned by WITNESS contains no extraneous attributes, suppose that, at the end of the loop, w contains one or more redundant attributes. Denote these attributes with C. Then, by Def. 4, there is an LAP $\{A_1, A_2\}$ of $attr(R)$ such that $R = \Pi_{A_1} R \times \Pi_{A_2} R$, $B \subseteq A_1$ and $C \subseteq A_2 \subseteq (attr(R) \setminus B)$. This and line 5 imply that for any witness $x \cup y$, the following equalities must hold:

$$\begin{aligned} x \cup y &= (\pi_{A_1}(x \cup y)) \cup (\pi_{A_2}(x \cup y)) \\ &= (x \cup (\pi_{A_1} y)) \cup (\pi_{A_2} y). \end{aligned}$$

Since $A_2 \subseteq (attr(R) \setminus B)$ and y is chosen from $\Pi_{(attr(R)\setminus B)} R$, we know that $\pi_{A_2} y \in \Pi_{A_2} R$. As a result, there must be a tuple $e \in R$ such that $\pi_{A_2} e = \pi_{A_2} y$. Now, let e' be the tuple $x \cup y' \in R$ for which $w = \{a \mid y(a) \neq y'(a)\}$ on line 13. Because $e, e' \in R$ and $\{A_1, A_2\}$ comprises an LAP for R, there must be a tuple $e'' \in R$ such that

$$e'' = (\pi_{A_1} e') \cup (\pi_{A_2} e) = (x \cup (\pi_{A_1} y')) \cup (\pi_{A_2} y).$$

Rewriting e'' as $x \cup y''$, where $y'' = (\pi_{A_1} y') \cup (\pi_{A_2} y)$, gets

$$\{a \mid y(a) \neq y''(a)\} = (\{a \mid y(a) \neq y'(a)\} \cap A_1) = W \cap A_1.$$

Given that $C \subseteq W$ is both non-empty and contained in A_2, $W \cap A_1$ must be a strict subset of W, which means that

$|\{a \mid y(a) \neq y''(a)\}| < |W|$. This, however, contradicts the post-condition of the loop on lines 9-12, which guarantees that the cardinality of W is minimal.

Correctness of the algorithm as a whole follows easily from the correctness of WITNESS. The running time is at most cubic in the size of R: this cost can be seen from line 5 in procedure WITNESS. In the worst case, $|X \times Y|$ is $O(|R|^2)$, so computing $(X \times Y) \setminus R$ can take up to $O(|R|^3)$ comparisons. We take the number of tuples in R as the dominant cost since the number of attributes is negligible in comparison.

Example. Fig. 7 illustrates an execution of the COMPUTELAP algorithm on the concrete specification T from Fig. 1. Each column in Fig. 7 represents one iteration of the main loop of COMPUTELAP.

During the first iteration, the algorithm checks if x_1 comprises a partition on its own, by trying to find a witness to the contrary. We can find such a witness showing that x_1 cannot be separated from some of the remaining labels. One such witness is $x \cup y = \{x_1 \mapsto 0\} \cup \{x_0 \mapsto 0, y_1 \mapsto 1, y_0 \mapsto 0, t_3 \mapsto 1, t_2 \mapsto 1, t_1 \mapsto 1, t_0 \mapsto 1\}$. Given this witness, the algorithm chooses a minimal set of labels in y, $W = \{t_1, t_0\}$, such that the values of these labels in $x \cup y$ can be changed to get a tuple in $\Pi_{rest} R$ (*e.g.,* $\{x_0 \mapsto 0, y_1 \mapsto 1, y_0 \mapsto 0, t_3 \mapsto 1, t_2 \mapsto 1, t_1 \mapsto 0, t_0 \mapsto 0\}$).

At this point, the algorithm has found that $\{x_1, t_1, t_0\}$ belong in the same partition, but must repeat the loop in PARTITION to ensure that no other attribute has been left out. It finds that x_0 should also be added to the partition. During the next iteration, no witness can be found (*i.e.,* $(X \times Y) = R$ on line 3 of WITNESS), so we add $B = \{x_1, x_0, t_1, t_0\}$ to A.

The algorithm then repeats the main loop of COMPUTELAP, choosing a new attribute to form a part of the finest LAP. The execution terminates when there are no attributes remaining. The finest LAP for T is $\{\{x_1, x_0, t_1, t_0\}, \{y_1, y_0, t_1, t_0\}\}$, as noted in the previous section.

4.2 Computing the LUPD

Given a relation R and an attribute partition $A \sqsubseteq \{attr(R)\}$, we use the COMPUTELUPD algorithm in Fig. 8 to enumerate all maximal product components of R with respect to A. If A has a single part, then its LUPD is simply $\{R\}$ (lines 2-3). If A consists of two or more parts, then the problem of computing the MPCs reduces to the problem of enumerating all maximal bicliques [1, 23] in the bipartite graph representation of R. In particular, given a partition $\{A_1, A_2\} \sqsubseteq \{attr(R)\}$, a specification R can be encoded directly as a bipartite graph $(V_1 \cup V_2, E)$ using the procedure $\mathcal{G}$ in Fig. 8. A maximal biclique in this graph is a maximal subgraph of the form $(V \cup V', V \times V')$, where the subgraph relation is defined in the usual way: *i.e.,* $V \subseteq V_1$, $V' \subseteq V_2$ and $V \times V' \subseteq E$. It is easy to see that the subrelation corresponding to a maximal biclique in $\mathcal{G}(T, \{A_1, A_2\})$ satisfies the definition of a maximal product component (Def. 7). Hence, line 6 correctly enumerates all maximal product components of R with respect to $\{A_1, A_2\}$. The correctness of the algorithm in the case of an A with more than two parts (lines 7-10) follows by induction from the base cases.

Since there may be exponentially many maximal bicliques in a given graph, the worst case running time of the COMPUTELUPD algorithm is exponential. In practice, however, graphs that correspond to concrete specifications have a small number of bicliques for any given decomposition. Our current implementation enumerates them quickly using a SAT-based constraint solver [37].

Example. Revisiting the example in Fig. 1, COMPUTELUPD enumerates all maximal components of L with respect to $A = \{\{x_1, x_0\}, \{t_1, t_0\}\}$ as follows.

Since the maximal biclique subroutine works on two partitions at a time, the algorithm creates a graph using L and A and enumerates all maximal bicliques. There are three such bicliques, $(V_i \cup V_i', V_i \times V_i')$. There is no need for a recursive call and the algorithm returns the three maximal components shown in Section 1. Fig. 9 illustrates the graph created during the call to COMPUTELUPD.

5. Case Studies

We evaluated our interactive synthesis methodology by applying it to three case studies. In the first two studies, we employ the methodology given in the overview, using a precise concrete specification to understand the inherent structure of the computation. In the remaining study, we solve the dual problem, using a sketch and an imprecise concrete specification to find a precise specification that satisfies the intuition of the programmer.

The evaluation was designed to assess the applicability of the decomposition analysis to writing structured sketches. With our approach, we find that we can tackle problems that are otherwise hard or impossible to solve through general syntax-guided synthesis. We show how to use our operators to discover recurrent structure in a computation (Sec. 5.1); cluster similar behaviors in a noisy specification (Sec. 5.2); and choose the best refinement of a non-deterministic specification that leads to a deterministic implementation (Sec. 5.3).

Two of the problems we study involve specifications with thousands of behaviors, which our algorithms decompose in seconds. All three problems are hard (or impossible) to solve without decomposition, either manually or using the best available automatic techniques. While our evaluation is limited to three application domains, we believe that the results presented here generalize to other domains as well.

5.1 Synthesis-Aided Deobfuscation of Programs with Loops

Our first application scenario is familiar: as in Sec. 2, we want to synthesize an easy-to-understand implementation of an obfuscated function. Unlike our toy example, the computations we consider next involve loops and recursion.

COMPUTELAP(R)			COMPUTELAP(R)		
$A = \emptyset$			$A = \{\{x_1, x_0, t_1, t_0\}\}$		
$rest = \{x_1, x_0, y_1, y_0, t_3, t_2, t_1, t_0\}$			$rest = \{y_1, y_0, t_3, t_2\}$		
PARTITION(x_1, R)			PARTITION($y_1, \Pi_{rest} R$)		
$B = \{x_1\}$	$B = \{x_1, t_1, t_0\}$	$B = \{x_1, x_0, t_1, t_0\}$	$B = \{y_1\}$	$B = \{y_1, t_3\}$	$B = \{y_1, y_0, t_3, t_2\}$
WITNESS(B, R)	WITNESS(B, R)	WITNESS(B, R)	WITNESS($B, \Pi_{rest} R$)	WITNESS($B, \Pi_{rest} R$)	WITNESS($B, \Pi_{rest} R$)
$X = \{0, 1\}$	$X = \{111, 000, 001\}$	$X = \{1011, 1111,$	$X = \{0, 1\}$	$X = \{11, 10, 00, 01\}$	$X = \{1011, 1111,$
$Y = \{0101111, \ldots\}$	$Y = \{01011, \ldots\}$	$0000, 0101\}$	$Y = \{011, \ldots\}$	$Y = \{01, 10\}$	$0000, 0101\}$
$\lvert Y\rvert = 16$	$\lvert Y\rvert = 8$	$Y = \{1011, 1100,$	$\lvert Y\rvert = 4$	$\lvert Y\rvert = 2$	$Y = \emptyset$
$x \cup y = 0 \cup 0101111$	$x \cup y = 000 \cup 11011$	$0001, 0110\}$	$x \cup y = 1 \cup 001$	$x \cup y = 11 \cup 10$	$\lvert Y\rvert = 0$
$W = \{t_1, t_0\}$	$W = \{x_0\}$	$\lvert Y\rvert = 4$	$W = \{t_3\}$	$W = \{y_0, t_2\}$	

Figure 7: Trace of COMPUTELAP as applied to the relation in Fig. 1. The finest LAP is $\{\{x_1, x_0, t_1, t_0\}, \{y_1, y_0, t_3, t_2\}\}$.

COMPUTELUPD(A, R)

switch A
  case $\{A_1\}$:
    return $\{R\}$
  case $\{A_1, A_2\}$:
    $B \leftarrow$ MAXBICLIQUES($\mathcal{G}(R, \{A_1, A_2\})$)
    return $\bigcup_{(V \cup V', V \times V') \in B} \{V \times V'\}$
  case $\{A_1, A_2, \ldots, A_n\}$:
    $A' \leftarrow A \setminus \{A_1\}$
    $G \leftarrow$ MAXBICLIQUES($\mathcal{G}(R, \{A_1, \bigcup_{2 \le i \le n} A_i\})$)
    return $\bigcup_{(V \cup V', V \times V') \in G} \bigcup_{M \in \text{COMPUTELUPD}(A', V')} \{V \times M\}$

$\mathcal{G}(R, \{A_1, A_2\})$

$V_1 \leftarrow \Pi_{A_1} R$
$V_2 \leftarrow \Pi_{A_2} R$
$E \leftarrow \{\langle v_1, v_2 \rangle \mid v_1 \in V_1 \wedge v_2 \in V_2 \wedge v_1 \cup v_2 \in R\}$
return $(V_1 \cup V_2, E)$

Figure 8: Algorithm for enumerating all MPCs for a relation R with respect to an attribute partition $A \sqsubseteq attr(R)$.

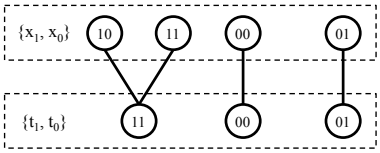

Figure 9: Graph of $\mathcal{G}(L, \{\{x_1, x_0\}\{t_1, t_0, \}\})$.

As such, they cannot be deobfuscated by existing synthesis-based techniques [15, 18], which can synthesize only loop-free code. We instead deobfuscate each by employing our interactive methodology to develop a structured sketch, which can then be passed off to a synthesizer.

We consider two functions, Z and H, that operate on finite precision integers. Each takes as input two k-bit integers and produces a $2k$-bit integer. Figures 10 and 14 show a sample concrete specification for each function, obtained by recording its output on every pair of k-bit inputs. Figure 11 illustrates the obfuscated (in reality, optimized) code for Z; the implementation for H is similarly complex.

x_1	x_0	y_1	y_0	z_3	z_2	z_1	z_0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	1	0
0	0	1	0	1	0	0	0
0	0	1	1	1	0	1	0
0	1	0	0	0	0	0	1
0	1	0	1	0	0	1	1
0	1	1	0	1	0	0	1
0	1	1	1	1	0	1	1
1	0	0	0	0	1	0	0
1	0	0	1	0	1	1	0
1	0	1	0	1	1	0	0
1	0	1	1	1	1	1	0
1	1	0	0	0	1	0	1
1	1	0	1	0	1	1	1
1	1	1	0	1	1	0	1
1	1	1	1	1	1	1	1

Figure 10: A concrete specification, $spec(Z)$, of Z for $k = 2$.

```
a, b, c = [...], [...], [...] // 3 arrays containing 256 constants each

def Z(x, y, z)
  r = 0;
  r = c[(z>>16) & 0xFF ] | b[(y>>16) & 0xFF ] | a[(x>>16) & 0xFF ]
  r = r<<48 | c[(z>>8) & 0xFF ] |
b[(y>>8) & 0xFF ] | a[(x>>8) & 0xFF ]
  r = r<<24 | c[(z) & 0xFF ] | b[(y) & 0xFF ] | a[(x) & 0xFF ]
  return r
```

Figure 11: An obfuscated implementation of Z.

$$spec(Z) = \begin{array}{|cc|}\hline y_1 & z_3 \\ 0 & 0 \\ 1 & 1 \\ \hline \end{array} \times \begin{array}{|cc|}\hline x_1 & z_2 \\ 0 & 0 \\ 1 & 1 \\ \hline \end{array} \times \begin{array}{|cc|}\hline y_0 & z_1 \\ 0 & 0 \\ 1 & 1 \\ \hline \end{array} \times \begin{array}{|cc|}\hline x_0 & z_0 \\ 0 & 0 \\ 1 & 1 \\ \hline \end{array}$$

Figure 12: The finest LPD for $spec(Z)$ (Fig. 10).

5.1.1 Deobfuscating Z

Figure 12 shows the finest LPD for Z. According to this LPD, we can decompose Z's specification into four simpler ones that relate each output bit to a single input bit. In particular, Z interleaves the input bits so that $z_{2*i} = x_i$ and $z_{2*i+1} = y_i$. Fig. 13 captures this insight in a sketch, which is completed by our synthesizer [36] in just a few seconds.[1]

5.1.2 Deobfuscating H

Our second function, H, can neither be synthesized by a simple sketch nor broken down further by the finest LPD.

[1] The function Z computes points on the G. M. Morton's Z-order curve[26].

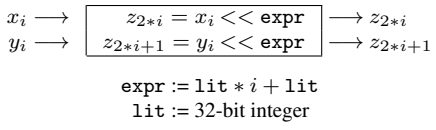

Figure 13: An abstract sketch for Z based on the LPD shown in Fig. 12. The sketch relates the i^{th} bits of input to the corresponding bits of output, using two holes constrained by the `expr` grammar.

x_2	x_1	x_0	y_2	y_1	y_0	h_5	h_4	h_3	h_2	h_1	h_0
0	0	0	0	0	0	0	0	0	0	0	0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
1	1	1	1	1	1	1	0	1	0	1	0

Figure 14: A concrete specification, $spec(H)$, of H for $k = 3$.

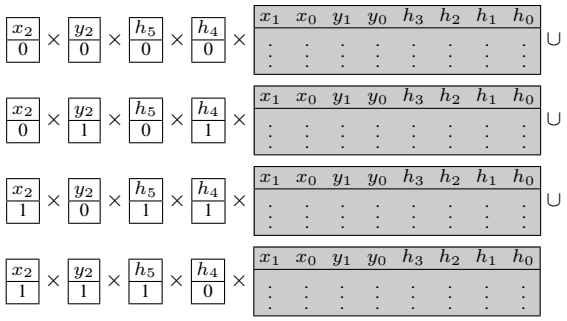

Figure 15: The LUPD of $spec(H)$ (Fig. 14) with respect to the attribute partition $A_2 = \{\{x_2, y_2\}, \{x_1, x_0, y_1, y_0, h_5, \ldots, h_0\}\}$.

Instead, we use the LUPD operator to validate whether the specification fits a particular sketch structure. We form the following hypothesis about its case structure: the highest-order bits of input, x_2 and y_2, jointly affect the behaviors of H in a way that does not depend on any lower order bits. Where did this hypothesis come from? Because the finest LPD proves that none of the input bits affect any output bits independently of others, our next guess is that the function takes a few bits from each input—perhaps, the bits in corresponding positions—and combines them to obtain one or more bits of output.

Expressing our initial hypothesis as the partition $A_2 = \{\{x_2, y_2\}, \{x_1, x_0, y_1, y_0, h_5, \ldots, h_0\}\}$, we obtain the LUPD in Fig. 15. This reveals an interesting correlation: the finest LAP of each maximal refinement of $spec(H)$ is $\{\{x_2\}, \{y_2\}, \{h_5\}, \{h_4\}, \{x_1, x_0, y_1, y_0, h_3, \ldots, h_0\}\}$, which means that the two highest-order bits of input uniquely determine the two highest-order bits of output, validating our hypothesis. Since the LUPD is showing *all* possible refinements that satisfy A_2, we can see that each pair of high-order input bits is combined with one specific pair of high-order output bits.

Our next hypothesis is that the pattern we observed for the high-order bits generalizes to the remaining bits, as

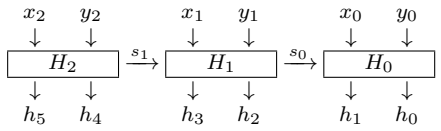

Figure 16: The structure of the computation performed by H. The output bits h_{2*i} and h_{2*i+1} are obtained from the input bits $x_k \ldots x_i$ and $y_k \ldots y_i$, where the value s_i stands for $x_k \ldots x_{i+1} y_k \ldots y_{i+1}$.

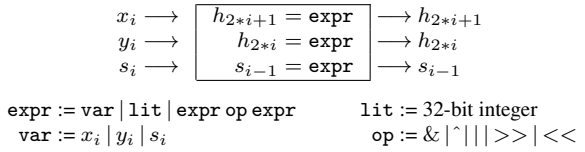

Figure 17: An abstract sketch for H based on the LUPD analysis. The sketch computes the output bits h_{2*i+1} and h_{2*i} from the input bits x_i and y_i, as well as n-bit summary s_i of the previous state of the computation. The grammar for `expr` uses standard bitvector operations.

illustrated in Fig. 16. The figure expresses the hypothesis that H computes its output bits from left to right, so that h_{2*i+1} and h_{2*i} are a function of $x_k \ldots x_i$ and $y_k \ldots y_i$. Note that we already know from Fig. 15 that x_i and y_i do not determine h_{2*i+1} and h_{2*i} by themselves—if they did, then x_1 and y_1 would determine h_3 and h_2, causing each maximal refinement in Fig. 15 to have an LAP of the form $\{\ldots, \{x_1, y_1, h_3, h_2\}, \ldots\}$.

To check our new hypothesis, we obtain the LUPDs of the highlighted (gray) specifications in Fig. 15 with respect to the partition $\{\{x_1, y_1, h_3, h_2\}, \{x_0, y_0, h_1, h_0\}\}$.[2] The resulting LUPDs validate the hypothesis: each has 4 refinements, and every refinement has the finest LAP of the form $\{\{x_1\}, \{y_1\}, \{h_3\}, \{h_2\}, \{x_0, y_0, h_1, h_0\}\}$.

We can now either encode the structure from Fig. 16 directly into a sketch or try the stronger sketch illustrated in Fig. 17. Our stronger sketch expresses the hypothesis that H computes its output incrementally, using a recurrence (*i.e.,* $H_0 = H_1 = H_2$). In particular, we guess that h_{2*i+1} and h_{2*i} are computed from x_i, y_i, and s_i, which is an n-bit summary of the previous state of the computation. Our synthesizer completes the sketch in 5 minutes, finding that $n = 2$ bits of summary are sufficient to implement H.[3]

5.2 Example-Based Clustering of Noisy Data

In our second case study, we develop a technique to tokenize a large set of strings by clustering together those with similar tokenizations. The original problem comes from the building science community, where sensors are used

[2] We could also obtain the LUPD of the original specification with respect to the partition $\{\{x_2, y_2, h_5, h_4, x_1, y_1, h_3, h_2\}, \{x_0, y_0, h_1, h_0\}\}$.

[3] The function H computes points on the Hilbert curve[17].

```
1  grammar tokenization:
2    tokenization = {split}*
3    split = integer literal
4
5  grammar pred[id]:
6    pred = id[int:int] == lit
7    int = integer literal
8    lit = string literal
9
10 grammar statements[id]:
11   statements = {if pred[id]: return tokenization}*
12
13 def tokenize(name):
14   statements[name]
```

Figure 18: Sketch of a program that given a sensor name, returns a tokenization. The program tokenizes a name certain way based on whether the name matches a set of conditions.

```
1  def tokenize(names) {
2    output = {}
3    while (names != ∅) {
4      n = choose(names)
5      t = userTokenize(n)
6      p = expressAsPartitionOfStringIndices(t)
7      cs = computeLupd(p, names)
8      fs = cs.filter(c → return n in c}
9
10     for (f ← fs) {
11       if (verify(computeLPD(f))) {
12         for (name ← f) {
13           output[name] = t
14           names.remove(f)
15   } } } }
16   return output
17 }
```

Figure 19: Algorithm for tokenizing a set of sensor names.

to collect real-time information. Each sensor name is composed of tokens [5, 32] that give information about the sensor (such as the type or location) which assist in writing building-applications. A commercial vendor often assigns sensor names manually, in an ad-hoc manner, making fully automated parsing impossible.

At first, this problem seems unrelated to our interactive method. But imagine that there exists a program which automatically tokenized a name. A sketch of such a program is given in Fig. 18. One approach to tokenize the names is to first group together names that are handled by the same predicate. An expert could then manually label each set with a tokenization. This decomposition closely resembles the output of the LUPD operator, except it cannot be applied since the tokenization is unknown. But we hypothesize that since the names handled by the same predicate have the same tokenization, they can be represented as the cross-product of tokens, *i.e.,* an LPD with an LAP derived from the tokenization. We can therefore exploit this structure in the names to find a cluster of names that share a tokenization. Note that we do not need to synthesize concrete predicates since the set of names is fixed.

Our technique tokenizes these names by asking an expert to provide tokens for one name, and then uses the LUPD analysis to discover a cluster of names with the same tokenization. We applied the technique to a set of 1532 sensor names taken from a real dataset. On average, our tool correctly tokenizes 1464 names (96%) by asking the expert to provide tokenizations for 91 of those names. This level of precision is competitive with the best existing example-based approach, with the additional benefit that our approach makes it much easier for the expert to verify that the resulting tokenizations are correct.

5.2.1 Building Sensor Data

All sensor names in our sample dataset consist of 14 characters. While a building manager can parse these 14 characters into tokens, there is no standard structure for an automated system to use. For example, the manager tokenizes the name 'SODA1C600A_ART' as $\boxed{\text{SOD}}\boxed{\text{A}}\boxed{\text{1}}\boxed{\text{C}}\boxed{\text{600A\_}}\boxed{\text{ART}}$. When expressed in terms of string indices, this tokenization[4] applies to several other names in the dataset, but the rest are tokenized differently. The name 'SODA2S14SASA_M', for example, is tokenized as $\boxed{\text{SOD}}\boxed{\text{A}}\boxed{\text{2}}\boxed{\text{S}}\boxed{\text{14}}\boxed{\text{SASA\_M}}$.

5.2.2 Our Approach

We assign tokenizations to sensor names using the algorithm in Fig. 19. We treat the names dataset as a concrete specification. Each name in the set represents a behavior that maps attributes $[0..13]$ to characters. The algorithm starts by asking an expert to provide the tokenization t for a randomly chosen name n (lines 4-5). It then finds a cluster of names with a similar structure, by obtaining the LUPD of names with respect to the attribute partition corresponding to t (line 7). We only keep the maximal refinements that include the name n (line 8), as these are heuristically most likely to contain names that should, in fact, satisfy the same predicate at n and therefore be tokenized like n. The expert verifies this heuristic guess (line 11) by examining the finest LPD of each such refinement (see Fig. 20 for an example). If the guess is correct, all names in the cluster are tokenized according to t and dropped from the dataset. These steps are repeated until all names are tokenized.

5.2.3 Alternative Approaches

Two alternate techniques exist for this problem. One approach uses a hand-written Python script, which was difficult to write and is difficult to maintain when new sensors are added. The other approach, RegEx, synthesizes a regular expression tokenizer from examples. Like our approach, the RegEx algorithm alternates between asking an expert to tokenize a single string n and parsing strings similar to n. Fig. 21 shows a few regular expressions generated on our sample dataset. We evaluate our algorithm against RegEx below.

[4] $\{\{0,1,2\},\{3\},\{4\},\{5\},\{6,7,8,9,10\},\{11,12,13\}\}$

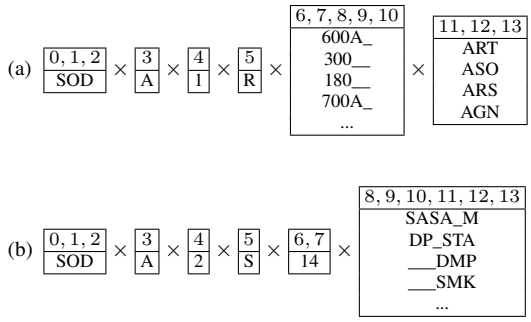

Figure 20: Refinements generated by the tokenizing of (a) 'SODA1C600A_ART and (b) 'SODA2S14SASA_M'.

```
^(SOD)(A)(.+?)(E)(.+?)_+?(RAT)$
^(SOD)(C)(.+?)(C)(.+?)(P)(.+?)_+?(STA)$
^(SOD)(A)(.+?)(R)(.+?)(RVAV)$
^(SOD)31NET___(TMR)$
^(SOD)_+?(BLD)_PR(ALM)$
^(SOD)(A)(.+?)(S)(.+?)_+?(P__VR)$
^(SOD)(C)(.+?)(P)(.+?)(DP_STA)$
^(SOD)(A)(.+?)(S)(.+?)_+?(DMP)$
^(SOD)34(BLD)_C_(SAS)$
^(SODA)_+?(CH)(.+?)_+?(CHWST)$
```

Figure 21: A subset of regular expressions generated by the RegEx algorithm on our sample dataset.

	# Correct	% Correct	# Examples
LUPD	1464	95.6%	91
RegEx	1489	97.2%	190

Table 1: Comparison of tokenization algorithms.

5.2.4 Evaluation

We compare our approach to RegEx by using the output of the Python script as the ground truth. The script was written in consultation with an expert, and we assume that it provides the most accurate tokenization of our dataset. Since the results of both our algorithm and RegEx depend on the order of randomly chosen names, we executed each 10 times, using the ground truth to answer queries posed to the expert. Table 1 presents the average precision and the number of expert queries across all runs.

We found that our algorithm matches RegEx in precision, while using half as many expert queries. Neither algorithm achieves 100% precision, due to the inherent ambiguity in our dataset (*i.e.,* a few names can be tokenized in multiple ways). But the results of our algorithm are much easier to verify. An expert can do so visually, by inspecting a decomposition of the kind shown in Fig. 20. RegEx, on the other hand, produces a long list of regular expressions, which can only be verified by applying them, in turn, to every name in the dataset. In summary, our approach provides comparable precision to RegEx, while requiring fewer expert queries and easing the process of verifying the results.

5.3 Developing Algorithms with Angelic Programming

For our third case study, we use our interactive methodology and angelic programming [6] to develop the Deutsch-Schorr-Waite (DSW) algorithm for marking reachable nodes in a directed graph. Angelic programming is similar to sketching: a developer writes a program replacing tricky-to-implement expressions with holes. These holes represent non-deterministic choice. But instead of producing an expression for each hole, a solver, playing the part of an *angelic oracle*, dynamically replaces each hole with a value such that the program terminates successfully. The resulting sequence of angelically chosen values forms a *trace*; in general, there are many valid traces for a given input. The programmer observes these traces and tries to generalize them to a deterministic implementation. A key challenge in this process is to identify the subset of traces that could be produced by a deterministic algorithm.

In this case study, we tackle the dual problem. The set of angelic traces provides a partial concrete specification. The desired concrete specification is the subset of traces that correspond to an easily implementable algorithm (such a constraint is difficult to encode in assertions, which leads to undesired traces). We use the structure of the angelic program and our decomposition operators to refine the partial specification to develop a determinstic DSW algorithm.

5.3.1 Angelic DSW

Unlike graph marking with an explicit stack, the DSW algorithm uses constant memory by cleverly reversing pointers in the graph. Bodik *et al.* [6] developed an implementation of DSW that hides the tricky pointer manipulations in a *parasitic stack*—a data structure that behaves like a stack but borrows storage from the host graph. Thanks to this formulation, DSW can be written as a standard depth-first traversal (not shown for brevity). The stack itself, however, is hard to implement and was developed using angelic programming.

Fig. 22 shows the angelic implementation of a parasitic stack. The choose(list) expressions denote non-deterministic choice; the runtime angelically selects a value from the provided list. The stack keeps just a single memory location (line 2). Its push and pop methods work by borrowing and restoring additional locations from the host graph.

In the original development of the parasitic stack, Bodik applied DSW to the example tree in Fig. 23, obtaining 8040 traces. Their test harness constrained the angelic runtime (via assertions) to look for executions that restore the tree to its original state and use the same number of pushes and pops. The resulting traces were examined manually to find a few that can be implemented with deterministic expressions. We now show how to find these desirable traces with just two refinement steps, guided by our decomposition analyses.

```
1  ParasiticStack {
2      e = choose(nodes in g) // initialize one extra storage location
3
4      // 'nodes' is list of nodes we can borrow from
5      def push(x : Node, nodes : List[Node]) {
6          // borrow memory location n.children[c]
7          n = choose(nodes)
8          c = choose(0 until n.children.length)
9
10         // value in borrowed location will need to be restored
11         v = n.children[c]
12
13         // select which 2 values to remember and where
14         e, n.children[c] = o.angelicallyPermute(x, n, v, e)
15     }
16     // 'values' is a list of nodes that may be useful
17     def pop(values : List[Node]) {
18         // choose the location we borrowed in push()
19         n = choose({e} ∪ values)
20         c = choose(0 until n.children.length)
21
22         // v is the value stored in the borrowed location
23         v = n.children[c]
24
25         // select return value, restore the borrowed location, and update e
26         r, n.children[c], e = angelicallyPermute(n,v,e,values)
27         return r
28     }
29 }
```

Figure 22: Angelic implementation of the parasitic stack.

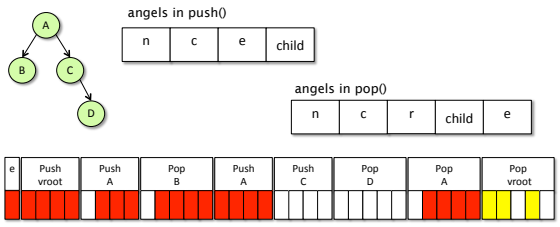

Figure 23: Specification decomposition for a small input. Colors show the finest LAP on the set of behaviors.

Analysis	E[# values]	E[# values] / length(trace)
Naive	1480	40
LPD/LUPD	276	7.5

Table 2: Comparison of the expected number of angelically chosen values a programmer must examine to find a correct trace. We also normalize this value by trace length (37) to give a sense of the number of traces the programmer examines.

The key idea in the refinement step is to use the LUPD operator with a partitioning inspired by the structure of the sketch. Doing so creates components such that holes in each partition are independent from each other. Since each partition corresponds to a piece of the sketch, the programmer can then reason about each piece independently. Each component also represents a different interface between the pieces of the sketch. By examining each interface, the programmer can choose which ones seem likely to be implementable and refine the concrete specification to this set.

5.3.2 Decomposition Analysis

Our concrete specification of the parasitic stack consists of the 8040 traces obtained by applying DSW to the example tree in Fig. 23. Each trace represents a single behavior, which maps dynamic invocations of choose expressions to their angelically selected values.[5]

Fig. 23 shows, by means of colors, the finest LAP of our specification. The angelic choices are visualized by the push or pop operation in which they occur. Each push makes four choices and each pop makes five; they select values for local variables as shown in the figure. Choices with the same color (red or yellow) all belong to the same part in the finest LAP. Uncolored choices are independent of all others—they each form their own singleton part.

The LAP reveals that the very first choose in the program, which initializes the extra location e, cannot be decomposed from other choices in the red part. This violates the intuition that the initialization value should be computed independently of all other values. To find the behaviors matching this intuition, we obtain the LUPD of the specification with respect to the attribute partition $\{\{\mathsf{e}\}, \{\ldots\}\}$.

The resulting LUPD contains four maximal refinements of the original specification, with three refinements containing roughly 2,000 behaviors each and one refinement with 6,000 behaviors. In the smaller refinements, all behaviors map e to the same node from the example tree—*i.e.,* their finest LPDs take the form $\boxed{\begin{matrix} e \\ n \end{matrix}} \times \ldots \times \boxed{\begin{matrix} \ldots \\ \ldots \end{matrix}}$, where n is a node in the example tree. In the larger refinement, however, e can be bound to any node. The behaviors in this refinement overwrite the location e before reading it, which matches a stronger hypothesis: the initialization value is not only independent of other choices in the program, but it can be any value. We continue the analysis with the largest refinement.

Our next hypothesis is that the choose expressions in the push method can be implemented independently of the choose expressions in pop. We therefore use LUPD to decompose the largest refinement into specifications that keep the push and pop choices independent from each other. The resulting decomposition consists of seven specifications. Examining their finest LPDs, we find that one specification has similar behaviors for all invocations of pop and all invocations of push. This specification allows the angelic choices for pop to make identical decisions, except for the choice of which child location to borrow (line 20). It also allows the angelic choices for push to make identical decisions except for which child location to restore (line 8). This specification turns out to contain precisely the traces that were previously [6] deemed to demonstrate the algorithm.

[5] We use execution indexing [40] to label the dynamic invocations of choose so that the same invocation has the same label in every trace.

5.3.3 Quantitative Evaluation

How much effort does the programmer save by using our decomposition operators versus manually inspecting each trace? As a proxy for measuring this effort, we count the *expected* number of angelically selected values that the programmer must examine. Table 2 summarizes our findings.

Without the decomposition operators, the programmer will randomly select and examine traces until he finds a demonstration of the algorithm. The original set has 8040 traces, each of length 37. Within this set, 200 traces correspond to a correct algorithm. The programmer is expected to examine 40 traces until a correct trace is found, assuming random selection without replacement. This leads to an expected cost of 1480 values.

To analyze the cost of using the decomposition operators, we consider each decomposition step in turn. The first step separates the extra location choose from the rest of the trace, leading to four maximal refinements. Each refinement has one value for the location except for one which has four values. Since the programmer only inspects the values of the extra location choose, the cost is 7 values.

The next application of LUPD decomposes the specification along function boundaries, leading to seven specifications. One specification contains precisely the 200 correct traces. We assume the programmer will randomly draw and examine values from each specification until he finds one correct trace. Each specification is relatively small, ranging from 54 to 80 values, giving an expected cost of 269 values.

In summary, our decomposition operators lead to a 5.3× reduction in the expected number of values examined. Most of the savings is a direct result of the compact representation as a cross product of independent specifications. Another source is the elimination of traces which do not fit the programmer's hypotheses.

6. Related Work

Concrete Specifications The notion of concrete specifications can be viewed as a generalization of other finite descriptions of program behaviors, such as input/output (IO) pairs or traces. IO pairs are widely used in program synthesis (*e.g.,* [14]) and testing (*e.g.,* [21]). Previous uses of traces include invariant detection [12], trace based optimization [13], and concurrency testing [39]. Concrete specifications capture both notions, providing a unifying view of finite program descriptions.

Decomposition A technique related to LPD is the lossless-join decomposition (LJD) from standard relational algebra. The goal of LJD is to remove redundancy by splitting a relation R into relations R_1 and R_2 such that $R_1 \bowtie R_2 = R$. This can be done if the functional dependencies of the relation satisfy Heath's Theorem [16]. Others have precisely defined the notion of independence in relation algebra context [30]. Unlike these approaches, there is no corollary to Heath's Theorem that corresponds to the notion of the LUPD. Consequently, relational algebra provides no way to extend the notion of the LJD to an arbitrary partition of attributes.

There has also been work in collecting, describing, and composing/decomposing specifications, although these specifications generally take a different form than ours, such as model transition systems[20, 38].

Others have proposed statistical methods that find properties inherent to a computation. Specification mining [3], for example, uses program executions and machine learning to create a state machine that represents implicit dependencies and, therefore, implicit modularity in a program. But statistical approaches could not compute the lossless decompositions produced by the (finest) LPD and LUPD.

Much work has also been done in trace analysis and clustering. Traces are generated by debug statements of a program, providing insight into its runtime state. Trace analysis can detect anomalies, eliminate redundant traces [11], or cluster similar traces together [24]. Our decomposition analysis may be applicable in these settings as well.

The finest LPD analysis is closely related to Boolean function decomposition (*e.g.,* [7, 19, 25]). Boolean truth tables are a special kind of concrete specification, where all attributes of a relation take on Boolean values. Boolean function decomposition breaks a complex function $f(X)$ into simpler subfunctions $h, g_1, \ldots, g_n$, such that $f(X) = h(g_1(X), \ldots, g_n(X))$. When applied to a truth table, the finest LPD produces a simple break down of f into a conjunction of formulas with disjoint variables. Unlike the finest LPD, which can be computed in polynomial time, general Boolean decompositions are more expensive. They are usually computed using BDDs or SAT solvers, but techniques involving relational algebra have been used as well [22]. In contrast to Boolean decomposition, our analysis is applicable to arbitrary relations, not just functions over Booleans.

7. Conclusion

We introduced a new method for interactive, tool-supported discovery of structure in a computation, and we showed how to use the resulting structure to arrive at tractable sketches. Our approach is based on the simple idea of decomposing concrete specifications, which are relations that explicitly enumerate the set of legal behaviors of a computation. We designed two automated decomposition operators that help discover independent subcomputations and case structure in a computation. We demonstrated our operators on three case studies, solving hard problems that cannot be solved with synthesis alone.

References

[1] G. Alexe, S. Alexe, Y. Crama, S. Foldes, P. L. Hammer, and B. Simeone. Consensus algorithms for the generation of all maximal bicliques. *Discrete Appl. Math.*, 145(1), 2004.

[2] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama,

E. Torlak, and A. Udupa. Syntax-guided synthesis. In *FMCAD*, 2013.

[3] G. Ammons, R. Bodík, and J. R. Larus. Mining specifications. *SIGPLAN Not.*, 37(1), 2002.

[4] D. Andre and S. J. Russell. State abstraction for programmable reinforcement learning agents. In *Eighteenth National Conference on Artificial Intelligence*. American Association for Artificial Intelligence, 2002.

[5] A. Bhattacharya, D. Culler, D. Hong, K. Whitehouse, and J. Ortiz. Writing scalable building efficiency applications using normalized metadata: Demo abstract. BuildSys '14. ACM, 2014. .

[6] R. Bodik, S. Chandra, J. Galenson, D. Kimelman, N. Tung, S. Barman, and C. Rodarmor. Programming with angelic nondeterminism. In *POPL*, 2010.

[7] H. Chen and J. Marques-Silva. Improvements to satisfiability-based boolean function bi-decomposition. In *VLSI-SoC*, 2011.

[8] A. Cheung, A. Solar-Lezama, and S. Madden. Optimizing database-backed applications with query synthesis. *SIGPLAN Not.*, 48(6):3–14, June 2013. ISSN 0362-1340. . URL `http://doi.acm.org/10.1145/2499370.2462180`.

[9] L. de Alfaro and T. A. Henzinger. Interface automata. In *FSE*, 2001.

[10] G. Dennis, R. Seater, D. Rayside, and D. Jackson. Automating commutativity analysis at the design level. In *ISSTA*, 2004.

[11] M. Diep, S. Elbaum, and M. Dwyer. Trace normalization. In *ISSRE '08*. IEEE Computer Society, 2008.

[12] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3), Dec. 2007.

[13] A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. *SIGPLAN Not.*, 44(6), June 2009.

[14] S. Gulwani. Automating string processing in spreadsheets using input-output examples. POPL '11. ACM, 2011.

[15] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *Programming Language Design and Implementation (PLDI)*, 2011.

[16] I. J. Heath. Unacceptable file operations in a relational data base. SIGFIDET '71. ACM, 1971.

[17] D. Hilbert. Über die stetige abbildung einer linie auf ein flachenstück. *Math. Ann.*, 38, 1891.

[18] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *ICSE*, 2010.

[19] M. Karnaugh. The Map Method for Synthesis of Combinational Logic Circuits. *Trans. AIEE. pt. I*, 72(9), 1953.

[20] I. Krka, Y. Brun, G. Edwards, and N. Medvidovic. Synthesizing partial component-level behavior models from system specifications. ESEC/FSE '09. ACM, 2009.

[21] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. PLDI '14. ACM, 2014.

[22] T. T. Lee and T. Ye. A relational approach to functional decomposition of logic circuits. *ACM Trans. Database Syst.*, 36(2), June 2011.

[23] J. Li, G. Liu, H. Li, and L. Wong. Maximal biclique subgraphs and closed pattern pairs of the adjacency matrix: A one-to-one correspondence and mining algorithms. *IEEE Trans. on Knowl. and Data Eng.*, 19(12), 2007.

[24] A. V. Miranskyy, N. H. Madhavji, M. S. Gittens, M. Davison, M. Wilding, and D. Godwin. An iterative, multi-level, and scalable approach to comparing execution traces. In *ESEC-FSE '07*, 2007.

[25] A. Mishchenko, R. K. Brayton, and S. Chatterjee. Boolean factoring and decomposition of logic networks. In *ICCAD*, 2008.

[26] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. *Technical Report*, 1966.

[27] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard. Automatically patching errors in deployed software. In *SOSP*, 2009.

[28] A. Pnueli and R. Rosner. On the synthesis of a reactive module. In *POPL*, 1989.

[29] Y. Pu, R. Bodik, and S. Srivastava. Synthesis of first-order dynamic programming algorithms. OOPSLA '11. ACM, 2011.

[30] J. Rissanen. Independent components of relations. *ACM Trans. Database Syst.*, 2(4), Dec. 1977.

[31] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. ASPLOS '13. ACM, 2013.

[32] A. Schumann, J. Ploennigs, and B. Gorman. Towards automating the deployment of energy saving approaches in buildings. BuildSys '14. ACM, 2014.

[33] R. Singh, S. Gulwani, and A. Solar-Lezama. Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 15–26, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. . URL `http://doi.acm.org/10.1145/2491956.2462195`.

[34] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. *SIGPLAN Not.*, 41(11), 2006.

[35] A. Solar-Lezama, G. Arnold, L. Tancau, R. Bodik, V. Saraswat, and S. Seshia. Sketching stencils. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 167–178, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. . URL `http://doi.acm.org/10.1145/1250734.1250754`.

[36] E. Torlak and R. Bodik. A lightweight symbolic virtual machine for solver-aided host languages. In *PLDI*, 2014.

[37] E. Torlak and D. Jackson. Kodkod: a relational model finder. In *TACAS '07*. Springer-Verlag, 2007.

[38] S. Uchitel and M. Chechik. Merging partial behavioural models. SIGSOFT '04/FSE-12. ACM, 2004.

[39] C. Wang, M. Said, and A. Gupta. Coverage guided systematic concurrency testing. ICSE '11. ACM, 2011.

[40] B. Xin, W. N. Sumner, and X. Zhang. Efficient program execution indexing. In *PLDI '08*. ACM, 2008.