Dimensions in Program Synthesis

[Invited Talk Paper]

Sumit Gulwani
Microsoft Research, Redmond, WA, USA
sumitg@microsoft.com

Abstract

Program Synthesis, which is the task of discovering programs that realize user intent, can be useful in several scenarios: enabling people with no programming background to develop utility programs, helping regular programmers automatically discover tricky/mundane details, program understanding, discovery of new algorithms, and even teaching.

This paper describes three key dimensions in program synthesis: expression of user intent, space of programs over which to search, and the search technique. These concepts are illustrated by brief description of various program synthesis projects that target synthesis of a wide variety of programs such as standard undergraduate textbook algorithms (e.g., sorting, dynamic programming), program inverses (e.g., decoders, deserializers), bitvector manipulation routines, deobfuscated programs, graph algorithms, text-manipulating routines, mutual exclusion algorithms, etc.

Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming; I.2.2 [*Artificial Intelligence*]: Automatic Programming – Program Synthesis; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

General Terms Languages, Theory

Keywords Deductive Synthesis, Inductive Synthesis, Programming by Examples, Programming by Demonstration, SAT Solving, SMT Solving, Machine Learning, Probabilistic Inference, Belief Propagation, Genetic Programming

1. Introduction

Program Synthesis is the task of discovering an executable program from user intent expressed in the form of some constraints. Unlike compilers, which take as input programs written in a structured language and mostly perform syntax-directed translations, synthesizers can accept a variety and mixed form of constraints (such as input-output examples, demonstrations, logical relations between inputs and outputs, natural language, partial or inefficient programs), and mostly perform some kind of search over some space of programs.

A synthesizer is typically characterized by three key dimensions: the kind of constraints that it accepts as expression of user intent, the space of programs over which it searches, and the search technique it employs. (i) The user intent can be expressed in the form of logical relations between inputs and outputs, input-output examples, demonstrations, natural language, and inefficient or related programs. (ii) The search space can be over imperative or functional programs (with possible restrictions on the control structure or the operator set), or over restricted models of computations such as regular/context-free grammars/transducers, or succinct logical representations. (iii) The search technique can be based on exhaustive search, version space algebras, machine learning techniques (such as belief propagation or genetic programming), or logical reasoning techniques. Most logical reasoning techniques involve two main steps: constraint generation, and constraint solving. (a) Constraint generation can be invariant-based, path-based, or input-based. (b) Constraint solving of resultant second-order quantified formulas typically involves reducing second-order unknowns to first-order unknowns (by use of templates), and eliminating universal quantifiers (by use of techniques such as Farkas lemma, cover algorithms, sampling), and then solving the resultant first-order quantifier-free constraints using off-the-shelf SAT/SMT solvers.

In this paper, we illustrate the above concepts by brief description of various program synthesis projects that target synthesis of a wide variety of programs such as standard undergraduate textbook algorithms (e.g., sorting, dynamic programming), program inverses (e.g., decoders, deserializers), bitvector manipulation routines, deobfuscated programs, graph algorithms, text-manipulating routines, mutual exclusion algorithms, etc.

This paper is organized as follows. We start by describing a few applications where program synthesis would be feasible and valuable (Section 2). We then describe the three dimensions in program synthesis: user intent (Section 3), search space (Section 4), and search technique (Section 5). We then describe the class of search techniques based on logical reasoning in more detail. These techniques typically have two main steps: Constraint Generation (described in Section 6) and Constraint Solving (described in Section 7). Finally, we conclude by describing some open research questions and mentioning how different areas of computer science can play a role in realizing the revolutionary potential of program synthesis technology (Section 8).

2. Applications

Program Synthesis has the potential to influence various classes of users in the technology pyramid ranging from algorithm designers (Section 2.1) and regular programmers (Sections 2.4-2.6) at the top of the pyramid to end-users (Section 2.2) and students (Section 2.3) at the bottom of the pyramid. Synthesis technology can also be useful for designers of cyber-physical systems, which are being increasingly deployed in transportation, health-care and other

PPDP '10 Hagenberg, Austria

societal applications [27, 34]. We discuss below few applications of program synthesis.

2.1 Discovery of New Algorithms

Finding a new algorithmic solution for a given problem requires human ingenuity: "The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music" [40]. Use of computational techniques to discover new algorithmic insights can be the ultimate application of program synthesis.

One domain of algorithms that has been shown amenable to automated synthesis is the class of *bitvector algorithms* [39, 74]. 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" [74]. Such programs can be quite unintuitive and extremely difficult for average, or sometimes even expert, programmers to discover methodically. Initial work [2, 52, 21] on automated synthesis of such algorithms used brute-force search, while recent work [23, 33] uses logical reasoning based techniques (powered by underlying SAT/SMT solvers) and scales much better.

Another domain of algorithms that has been shown amenable to automated synthesis is that of *mutual exclusion algorithms*, which guarantee mutually exclusive access to a critical section among a number of competing processes [17]. The essence of mutual exclusion algorithms is the code before and after the critical section that together ensures properties such as mutual exclusion, deadlock-freedom, and possibly starvation-freedom. Mutual exclusion algorithms differ based on number and size of shared/local variables, number of commands in the entry/exit sections, the kind of conditionals used, and whether starvation-freedom is desired or not. Initial work [5] for automatically discovering such algorithms used brute-force search (carefully trimmed by various optimizations) and successfully discovered many new algorithms (and rediscovered some known ones). Recent work [38] uses a novel technique based on combining genetic programming and model checking. It reports performance better than brute-force methods, but requires carefully designed fitness functions.

2.2 Automating Repetitive Programming for End-Users

Computational devices have become accessible to people at large at an impressive rate. Most computer users are *end-users*, who are at the end of the process of computer programming, far removed from the programmer. These business end-users have a myriad of diverse backgrounds and include commodity traders, graphic designers, chemists, human resource managers, finance pros, marketing managers, underwriters, compliance officers, and even mail-room clerks – they are not professional programmers, but they need to create small, often one-off, applications to support business functions [22].

Graphical user interfaces have been designed to make it easier for these end-users to use computers. However, they are still not ideal since users struggle to find the correct feature or succession of commands to use from a maze of features to accomplish their task. Figuring out the right parameters to provide as trigger/input to the feature is also sometimes difficult. More significantly, programming is still required to perform tedious and repetitive tasks. Examples of such repetitive tasks include: transforming lists of addresses from one format to another, extracting data from several web pages into a single document, renaming files in a directory, managing bibliographies, etc. The programs that end-users use to manage/manipulate their data, such as spreadsheets, databases, browsers, and scripting platforms, are not written with their particular needs in mind. These programs attempt to do an average job for meeting average needs and are far from doing a perfect job for personalized task-oriented needs.

Program synthesis can play an extremely useful role for end-users who can easily illustrate their intent by input-output examples or traces, but find it difficult to write programs. A good start in this direction has been made by *Programming by demonstration* systems that enable users to write programs by demonstrating the program on one or more concrete examples [13, 44].

2.3 Teaching

The importance of automation in teaching cannot be understated. Most public schools, especially in developing countries, struggle to find good teachers. Many parents spend several man-years helping their kids with their homeworks and imparting knowledge. If this motivation is too earthly, consider the role that educational robots play in the frozen embryos approach to inter-stellar travel.[1]

Program synthesis can play a revolutionary role in automating the teaching landscape. In particular, program synthesis technology can help with *automated problem solving*. Manna and Waldinger observed that the ability to construct proofs of theorems can yield a program to solve the corresponding problem [51]. Since then, many new program synthesis approaches have emerged that can also be used to classroom problem solving.

As part of this effort, the author is currently working on a technique to automatically synthesize high-school geometrical constructions such as construction of a regular hexagon given a side. Another example of class of problems (taught in an undergraduate class) whose solutions can be automated (using a current synthesis technology [4]), is that of construction of finite automatas such as the smallest automata that accepts all strings with even number of 1s but not more than 4 consecutive 0s.

Among other teaching tasks that can be automated (besides problem solving) using synthesis technology are construction of problems of a measured difficulty, and interactive grading with explanations of any errors and fixes. The latter task of pointing errors and suggesting fixes might benefit from the work on bug localization and automated debugging in the program analysis and software engineering community.

2.4 General Purpose Programming Assistance

Since synthesis is a very hard problem, fully automated synthesis of *large* pieces of software might not be possible in the near future. However, an effective use of synthesis technology for general purpose programming can be its role as a programmer's assistance. Synthesis can be used to synthesize small program fragments from higher-order logical specifications [43, 31]. Synthesis can be used to find tricky/mundane implementation details after human insight has been expressed in the form of a partial program [65]. Various interesting forms of interactivity can also be employed between the programmer and the synthesizer in a semi-automated software development process [64, 6, 72].

Another interesting application of synthesis technology in the software development process can be to assist with the debugging process, as explained below.

[1] Wikipedia defines it as: "A robotic space mission carrying some number of frozen early stage human embryos is another theoretical possibility. This method of space colonization requires, among other things, the development of a method to replicate conditions in a uterus, the prior detection of a habitable terrestrial planet, and advances in the field of fully autonomous mobile robots and *educational robots* which would replace human parents." on http://en.wikipedia.org/wiki/Interstellar_travel#Frozen_embryos.

Automated Debugging Debugging is a time-consuming and difficult process that is today mostly done manually, often requiring hours to fix a single bug, and unnecessary taxation of brain power, better saved for less mundane tasks. Debugging can be phrased as a program synthesis problem after identification of a region of code that is suspected of incorrect behavior, and a description of the desired behavior, possibly in the form of input-output examples or symbolic relations between values of variables at the beginning and end of the identified region of code. The buggy region of code may be identified by the programmer [36] or may be guessed to be those regions of code that are executed on the failing runs, but not on the passing runs [75].

The debugging application also provides for some unique advantages that can be exploited by program synthesis techniques, especially those based on machine learning techniques.

The probabilistic inference based program synthesis technique described in [36] can take advantage of the existence of initial buggy piece of code. The core idea of this technique is to model the program as a graph consisting of instructions and states as potentially unknown variables, connected by constraint nodes, and then use *belief propagation* to infer both the intermediate program states and the instructions that satisfy all the constraints. In the debugging scenario, the beliefs about instructions can be initialized based on the initial buggy piece of code, and belief propagation is performed (which leads to altering the code) until the resultant code satisfies the input-output constraints. It is shown that such an initialization speeds up inference. This technique has been used to fix small bugs involving up to six incorrect instructions. These bugs may be simple ones like use of a wrong variable name, resetting a variable to zero at the wrong location, inverting the order of two instructions, forgetting to increase a counter (or decreasing it instead), etc, or more complex ones that are best addressed by completely replacing a short code fragment.

The genetic programming based program synthesis technique described in [75] takes advantage of the hypothesis that a missing important functionality in a program can be copied and adapted from another location in the program. The core idea of this genetic programming technique is to evolve program variants until one is found that both retains required functionality and also avoids the defect in question. This technique has been used to automatically generate repairs for ten C programs totaling 63,000 lines of code.

2.5 Synthesis of Program Inverses

The problem of program inversion is to derive a program P^{-1} that negates the computation of a given program P. More formally, it is the problem of inverting an injective program P by finding another program P^{-1} that is its left-inverse. This problem arises naturally in paired computations such as compression/decompression, encryption/decryption, serialization/deserialization, insert/delete operations on data structures, transactional memory rollback, bidirectional programming, and even client-server applications. Given the prevalence of program inverses and the cost associated with maintaining two closely related programs, automatic program inversion can be beneficial in ensuring correctness and maintainability.

Initial work on deriving program inverses used proof-based [16] or grammar-based [18] approaches. Recent work [67] phrases the program inversion problem as a finite synthesis problem, and uses a novel path-based inductive synthesis technique (described in Section 6.2) to invert a larger class of programs than was possible before. In particular, it can synthesize inverses for compressors (e.g., LZ77), packers (e.g., UUEncode), and arithmetic transformers (e.g., image rotations). The inverses for these non-trivial programs range from 5 to 20 lines of code, and are automatically synthesized in a median time of 40 seconds.

2.6 Program Understanding

A given program may often be non-trivial to understand if it is, for example, obfuscated, too low-level, or missing documentation. Program synthesis technology can help in program understanding by translating such a given piece of code to a semantically equivalent code written in some target language that is more readable/understandable/higher-level. The given program acts as a specification of the desired equivalent program in the target language.

One instance of program understanding arises in the context of malware deobfuscation, where the challenge is to understand what the malicious code is doing. The need for deobfuscation techniques has arisen in recent years, especially due to an increase in the amount of malicious, and mostly obfuscated, code (malware) [70]. Currently, human experts use decompilers and manually deobfuscate the resulting code (see, e.g., [59]). Clearly, this is a tedious task that could benefit from automated tool support. The program synthesis technique described in [33] has been applied to malware deobfuscation by deobfuscating examples drawn from and inspired by the Conficker [59] and MyDoom [54] viruses.

A related instance is in the context of reverse engineering of a binary, when the source code is not available. Program synthesis techniques for learning programs from traces are well suited to be applied in this context since the program state can easily be observed even though the source code is not [47].

Another instance arises in the context of software maintenance tasks (such as bug fixing or refactoring), where the challenge is to first understand a given piece of poorly documented low-level code, possibly by translating it into some higher level representation involving, for example, sets and second-order logic. This is an application of synthesis technology that is an antithesis of its more traditional application of converting programs written using higher-order primitives such as sets and higher-order quantifiers into efficient code (as is done in [31]). The QuickSpec tool [12] takes another interesting approach in this context and generates algebraic specifications for modules consisting of pure functions. It has been applied to understanding of a heap library for Haskell and a fixed-point arithmetic library for Erlang.

3. First Dimension: User Intent

The most important dimension in program synthesis from the perspective of the user is the mechanism for describing intent. There are various choices possible that have been briefly explained below: logical specifications, natural language, input-output examples, traces, and higher-order, inefficient or partial programs. A particular choice may be more suited in a given scenario depending on the underlying task as well as on the technical background of the user. An important open research question is to identify how to combine these various choices in a unified interface.

3.1 Logical Specifications

A logical specification is a logical relation between inputs and outputs of a program. It can act as a precise and succinct form of functional specification of the desired program.

For example, the logical specification for a sorting algorithm would be the following logical relation that asserts that the output array B is *sorted* (Eqn 1) and is a *permutation* of the input array A of size n (Eqn 2).[2]

[2] Eqn 2 correctly states the permutation requirement under the assumption that all elements in the input array are distinct, which is assumed for simplicity.

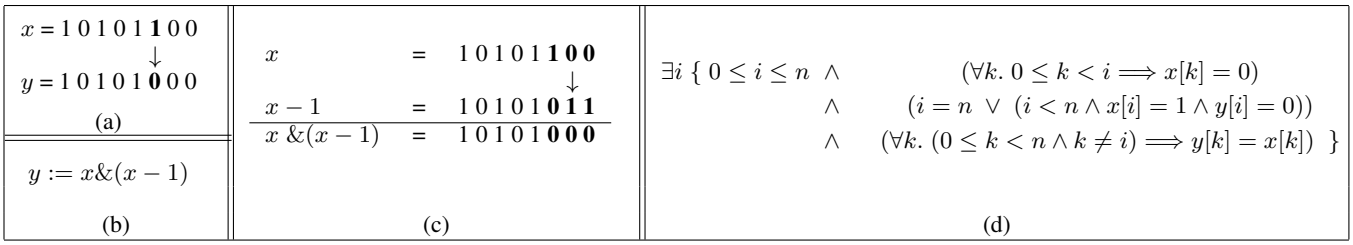

Figure 1. Consider the problem of masking off the rightmost significant one-bit in a given bitvector. (a) describes an example input-output pair (x, y). (b) describes a 2-step program to solve the problem. (c) describes the working of the 2 step program. (d) describes the intent using a logical relation between input bitvector x and output bitvector y. For notational convenience, we represent an n-bit bitvector as an array of n bits, with the bit positions starting from 0 and numbered from right to left.

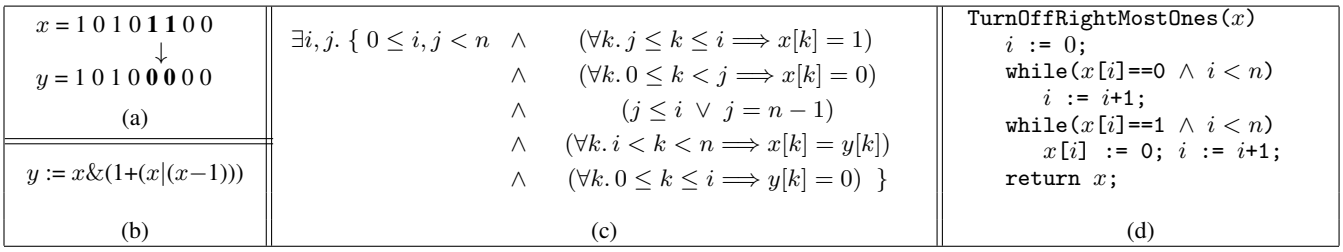

Figure 2. Consider the problem of masking off the rightmost contiguous sequence of 1's in a given bitvector. (a) describes an example input-output pair (x, y). (b) describes a 4-step program to solve the problem. (c) describes the intent using a logical relation between input bitvector x and output bitvector y, both of which are of size n. (d) describes the intent using an inefficient program.

$$\forall k.\ (0 \leq k < n-1) \implies (B[k] \leq B[k+1]) \quad (1)$$
$$\wedge \quad \forall k\ \exists j.\ (0 \leq k < n) \implies (0 \leq j < n \wedge B[j] = A[k]) \quad (2)$$

Note that the above specification of sorting property does not suggest in any way how to efficiently implement a sorting algorithm. The approach described in [69] can discover algorithms from logical specifications as above. In particular, given the above sorting specification, it discovers five sorting algorithms: InsertionSort, SelectionSort, BubbleSort, MergeSort, and QuickSort[3].

As another example, consider the logical specifications provided in Figure 1(d) and Figure 2(c) for the corresponding bitvector tasks illustrated in Figure 1(a) and Figure 2(a) respectively. The technique described in [23] is specialized to discovering efficient bitvector tricks (as in Figure 1(b) and Figure 2(b)) from such logical specifications.

Several synthesis systems accept user intent in the form of logical specifications [51, 69, 23, 31]. Compared to other forms of specifications such as input-output examples and demonstrations, logical relations require additional knowledge of logic and might be harder to get right, and may not be a preferred form of specification for end-users.

3.2 Natural Language

Given advances in natural language processing, it is possible to map natural language sentences into logical representations [77]. Natural language can be used as a substitute for logical relations, and end-users might find it very valuable. In particular, natural language interfaces have been designed to query databases to accommodate the need of end-users who interact with databases, but are intimidated by the idea of using languages such as SQL [3, 49].

One disadvantage with natural language is that it can be ambiguous. This issue can potentially be resolved by using the concept of paraphrasing where the system interacts with the user to resolve any ambiguity in the user-provided description [29].

3.3 Input-Output Examples

In several scenarios, input-output examples can act as the simplest form of specification, with relatively little chances of error. Quite contrary to what it may seem initially, input-output examples, in conjunction with interactive rounds, can often play the role of a full functional specification.

It is natural to ask what prevents the synthesizer from synthesizing a trivial program that simply performs a table lookup as follows, when provided with the set $\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$ of input-output pairs.

```
switch x
   case x1: return y1;
   case x2: return y2;
      ⋮
   case xn: return yn;
```

The restriction on the search space is the first line of defense against allowing such trivial solutions. In particular, the search space might permit only a bounded number of statements or conditionals.

Another concern with input-output examples is the selection criterion and the number of input-output examples. In other words, identifying what constitutes a good input-output example, and how many examples should the user provide? The user may start by providing few input-output examples (possibly a couple of examples for each of the corner cases) and then add more input-output exam-

[3] One might ask why the system described in [69] only discovers these five sorting algorithms. This is because the underlying search space was expressive enough to represent only these algorithms.

User	Oracle		
Input → Output	**Program 1**	**Program 2**	**Distinguishing Input ?**
$01011 \rightarrow 01000$	$(x+1)\,\&\,(x-1)$	$(x+1)\,\&\,x$	00000 ?
$00000 \rightarrow 00000$	$-(\neg x)\,\&\,x$	$(((x\&-x)\mid -(x-1))\&x)\oplus x$	00101 ?
$00101 \rightarrow 00100$	$(x+1)\&x$	$\cdots$	01111 ?
$01111 \rightarrow 00000$	$\cdots$	$\cdots$	00110 ?
$00110 \rightarrow 00000$	$\cdots$	$\cdots$	01100 ?
$01100 \rightarrow 00000$	$\cdots$	$\cdots$	01010 ?
$01010 \rightarrow 01000$	$(((x-1)\mid x)+1)\&x$	None	Program is $(((x-1)\mid x)+1)\&x$

Figure 3. An illustration of the synthesizer driven interaction model for synthesis from input-output examples of the task illustrated in Figure 2(a). Program 1 and Program 2 are two semantically different programs generated by the synthesizer that are consistent with the past set of input-output pairs provided by the user. The synthesizer also produces a distinguishing input on which the two programs yield different results, and asks the user for the output corresponding to the distinguishing input. The process is repeated until the synthesizer can find at most one program.

ples in each interactive round. The interaction may either be driven by the user or by the synthesizer, as explained below.

User Driven Interaction The user may inspect the program returned by the synthesizer, either by studying the program itself, or by testing it on several other inputs. If the user finds any discrepancy in the behavior of the program and the expected behavior on some new input, the user may repeat the synthesis process after adding the new input-output pair to the previous set of input-output examples.

Synthesizer Driven Interaction Given a set of input-output pairs, the synthesizer searches for programs that map each input in the given set to the corresponding output. The number of such programs may usually be unbounded, if the search space consists of all possible programs. However, since the search space is usually restricted, the number of such programs may either be 0, 1, or more than 1. If the synthesizer is unable to find any such program over the search space, the synthesizer declares failure. If the synthesizer finds exactly 1 program, the synthesizer declares success and presents the program to the user.

If the synthesizer finds at least two programs P_1 and P_2, both of which map each input in the given set to the corresponding output, the synthesizer declares the user specification to be partial. It then generates a *distinguishing input*, an input on which the two programs P_1 and P_2 yield different results, and asks the user to provide the output corresponding to the distinguishing input. The synthesis process is then repeated after adding this new input-output pair to the previous set of input-output examples. This interaction model is described in [33].

For example, consider the task of synthesizing the bitvector program that masks off the rightmost contiguous sequence of 1s in the input bitvector (i.e., the problem described in Figure 2). One would agree that it is easier to provide input-output examples for the task than write down the logical specification described in Figure 2(c). The synthesizer driven input-output interaction process is illustrated in Figure 3. The user may start out by providing one input-output example $(01011, 01000)$ for the desired program. The synthesizer generates a candidate program $(x+1)\,\&\,(x-1)$ that is consistent with the input-output pair $(01011, 01000)$. Then, it checks whether a semantically different program also exists and comes up with an alternative program $(x+1)\,\&\,x$ and a distinguishing input 00000 that distinguishes the two programs, and asks the user for the output for the distinguishing input. The newly obtained input-output pair $(00000, 00000)$ rules out one of the candidate programs, namely, $(x+1)\,\&\,(x-1)$. In the next iteration, with the updated set of input-output pairs, the synthesizer finds two different programs $-(\neg x)\,\&\,x$ and $(((x\&-x)\mid -(x-1))\&x)\oplus x$ and a distinguishing input 00101. It then asks the user for the output for 00101. The newly added pair $(00101, 00100)$ rules out $(((x\&-x)\mid -(x-1))\&x)\oplus x$. Note that at this stage, the program $(x+1)\&x$ remains a candidate, since it was not ruled out in the earlier iterations. In next four iterations, the synthesizer driven interaction leads to four more input-output pairs: $(01111, 00000)$, $(00110, 00000)$, $(01100, 00000)$ and $(01010, 01000)$. The semantically unique program generated from the resulting set of input-output pairs is the desired program: $(((x-1)\mid x)+1)\&x$.

Number of Interactive Rounds A concern in an interactive process might be the number of interactive rounds required. In the worst case, the number of interactive rounds required might be the size of the input space. However, this may not happen in practice because the search would typically be restricted to space of programs with low Kolmogorov (descriptive) complexity [48], or small teaching dimension [20]. In fact, experimental results for synthesis of bitvector programs indicate that the number of interactive rounds required is roughly equal to the number of instructions in the synthesized program [33].

3.4 Traces

A *trace* is a detailed step-by-step description of how the program should behave on a given input. A trace is a more detailed description than an input-output example since it also illustrates how a specific input should be transformed into the corresponding output as opposed to just describing what the output should be. This input model can be used when the user already knows how to perform the task, and the goal is to create a program to perform the task. Traces are an appropriate model for programming by demonstration systems for end-users [13], where the intermediate states resulting from the user's successive actions on a user interface constitute a valid trace. Traces may also be easier to provide than writing a program in the context of general purpose programming [7]. Traces are also readily available for reverse engineering scenarios [47].

From the perspective of the synthesizer, traces are preferable to input-output examples since the former contains more information. The synthesizer needs to generalize from given demonstrations or traces such that the generalized trace or the program can work for other inputs as well. This is a relatively easier challenge than searching for a program that is consistent with a given input-output example.

From the perspective of the user, providing demonstrations in general may be more taxing than providing input-output examples.

However, in certain scenarios, demonstrations may be more preferable than input-output examples. This happens when the output is not easy to compute. For example, suppose the user wants to synthesize a program for computing factorial. A demonstration for input 7 can be the string "$7 \times 6 \times 5 \times 4 \times 3 \times 2$" or the recursive trace "$7 \times \text{factorial}(6)$", which is easier to describe than providing the final simplified output 5040.

3.5 Programs

Programmers might sometimes find a programming language as the best means of specifying their intent. This happens quite trivially for certain applications such as deobfuscation (Section 2.6) and synthesis of program inverses (Section 2.5), where the program to be deobfuscated or inverted respectively forms the specification.

However, even for applications such as discovery of new algorithms, some people might find it easier to write the specification as an inefficient program than a logical relation. For example, consider the bitvector problem described in Figure 2, and contrast its logical representation shown in Figure 2(c) with its program representation shown in Figure 2(d). The latter might be more appealing to programmers.

4. Second Dimension: Search Space

The second dimension in program synthesis is the space of programs over which the desired program will be searched. This choice is made by the developer of the synthesizer and may optionally be restricted by the user of the synthesizer.

The developer of the synthesizer needs to strike a good balance between expressiveness and efficiency of the search space. On one hand, the space of the programs should be large/expressive enough to include programs that users care about. While on the other hand, the space of the programs should be restrictive enough so that it is amenable to efficient search, and it should be over a domain of programs that are amenable to efficient reasoning.

The user of the synthesizer can optionally restrict the search space to obtain programs with specific resource usage. For example, the user might desire a loop-free program, or a program whose memory usage does not exceed a specified amount.

Broadly speaking, the search space can be over the space of (Turing-complete) programs, or restricted form of computational models such as grammars or logics.

4.1 Programs

The space of programs can be qualified by at least two attributes: (i) the operators used in the program, and (ii) the control structure of the program.

4.1.1 Operators

The choice of operators can be restricted to comparison operators (sufficient for algorithms such as sorting), arithmetic operators, APIs exported by a given library, combination of arithmetic and bitwise operators etc. We discuss the choice of combination of arithmetic and bitwise operators in some detail below.

Arithmetic + Bitwise Operators We use the term *bitvector programs* to refer to those programs that involve combination of the following two kinds of operators:

- Arithmetic Operators such as Addition $+$, Subtraction $-$, Multiplication $*$, Division $/$.
- Bitwise Operators such as bitwise-and $\&$, bitwise-or $|$, bitwise-xor $\oplus$, left-shift, right-shift, rotate, etc.

Bitvector programs (also described in Section 2.1) are useful because they often provide the most efficient way to accomplish various tasks on standard architectures. Such programming problems often arise while developing low level embedded code, network applications or in other domains where bit-level manipulation is needed. They 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. Bitvector tricks are also helpful for designing specialized hardware.

For example, consider the problem of masking off the rightmost significant 1-bit in a bitvector, as shown in Figure 1(a). An approach that comes to mind immediately is to use a loop to iterate over the bitvector from right to left, checking each bit one by one, until a 1-bit is found, and then convert that bit to 0. The problem with this approach is that it involves too many steps (worst-case linear in the number of bits in the bitvector), and additionally each step involves use of conditionals. However, the desired task can be achieved by computing $x\&(x-1)$ (which involves composing the bitwise $\&$ operator and the arithmetic subtraction operator in an unintuitive manner), as illustrated in Figure 1(c). [23] describes a system for synthesis of bitvector programs from logical specifications while [33] describes a system for bitvector program synthesis using the synthesizer driven input-output interaction model.

4.1.2 Control Structure

The control structure of the program may be restricted to a given looping template [69], a program with bounded number of statements [5], a partial program with holes [65], or even to loop-free programs [23, 33, 76, 46, 32, 43, 50]. We discuss the choice of loop-free programs in some detail below.

Loop-free Programs The class of straight-line programs, or more generally, loop-free programs, parametrized by the set of operators/components used, can often express a wide range of useful computations. Following are some interesting examples of such programs.

- Bitvector algorithms [23, 33]. The set of components includes arithmetic operators and bitwise operators.
- Text-editing programs [76, 46]. The set of components includes basic editing commands (available in a text-editor) such as insert, locate, select and delete.
- Geometrical constructions [32]. The set of components includes geometrical constructors such as ruler and compass.
- Unbounded data type manipulation [43]. The set of components include those that the underlying decision procedure can reason about such as linear arithmetic operators and set operators.
- API call sequences [50]. The set of components includes the API calls.

Even though these programs may not involve loops, they may still be challenging to synthesize. The space of loop-free programs is still huge: the number of possible programs is exponential in the number of components in the program, and hence brute-force search methods do not usually scale. The problem of synthesis of straight-line programs may be likened to the problem of solving a Jigsaw puzzle.

4.2 Grammars

There has been a lot of work in learning various kinds of grammars such as regular expressions [55, 63], DFAs [8, 4], NFAs [10], context-free grammars [15], regular transducers [73, 56]. Learning grammars has a wide range of applications ranging over robotics and control systems, pattern recognition, computational linguistics, computational biology, data compression, data mining, etc. [14].

Learning regular expressions, in particular, is important for at least two scenarios:

- Regular expressions are a key component of text-editing programs, where they are used to select substrings inside a piece of text. Learning such regular expressions [55, 63] is a key step of learning text-editing programs.
- Regular expressions can represent the control structure of a program. Learning such regular expressions (as done in [60]) can increase the usability of general program synthesis schemes that assume that the control structure has been provided, for example, in the form of a sketch [65] or a scaffold [69].

4.3 Logics

Logical representations, because of their succinctness, can serve as good target languages for program synthesis. In particular, the class of first order logic together with fixed point equals the class of PTIME algorithms over ordered structures such as graphs, trees, strings [30]. Hence, this class and also some of its useful subclasses (such as those with a fixed quantifier depth) can serve as good target languages for synthesizing efficient graph/tree algorithms [31].

5. Third Dimension: Search Technique

In this section, we provide a high-level description of some general techniques that have been used for program synthesis. The applicability and the specifics of these techniques may depend on the search space of the programs and may also depend on the form of input constraints.

5.1 Brute-force Search

The brute-force search technique refers to the process of enumerating the programs in the search space in some order and checking for each program whether it satisfies the input constraints. There have been few success stories of using brute-force search to discover new algorithms: mutual-exclusion algorithms [5] and bitvector algorithms [2, 52, 21].

Brute-force search has also been experimented with in the context of functional programs. [37] presents a system that searches for desired small functional programs by generating a sequence of type-correct programs in a systematic and exhaustive manner and evaluating them against given specifications. The QuickSpec tool [12] automatically generates algebraic specifications for modules consisting of pure functions for the purpose of program understanding.

Brute-force search is the simplest search strategy, but is more often than not prohibitively expensive. Each of the above-mentioned system makes brute-force search feasible by implementing various optimizations for pruning the search space.

5.2 Version Space Algebra

Version space algebra is another relatively simple, but efficient, concept that has been used for program synthesis. Version space algebras have been used for learning repetitive robot programs [57], shell scripts [45], text-editing programs [46], and imperative Python programs [47].

It is based on the notion of *version space* that was introduced by Mitchell who proposed a general search technique for discovering boolean functions given positive/negative examples and a language/space over which to search for the boolean function [53]. The basic idea is to maintain a set of all boolean functions in the language that could be the desired unknown function (referred to as *version space*), i.e., those that correctly classify the given training inputs, and then iteratively refine the hypothesis as more data becomes available. The notion of *generality* of functions imposes a partial order that allows more efficient representation of the version space by the boundary sets representing the most specific and most general functions in the space.

Lau et.al. later extended Mitchell's version space concept to *version space algebra* for learning more complex functions that have any range [46]. The basic idea is to build up a complex version space by composing together version spaces containing simpler functions as opposed to defining a single version space containing the entire function space. Just as two functions can be composed to create a new function, two version spaces can be composed to create a new version space, containing functions that are composed from the functions in the original version spaces. As Pardowitz et.al. point out, this allows to represent hypotheses on programs and underlying loop structures in a hierarchical manner [57]. They are constructed out of atomic or other compound hypotheses, resembling the tree-like structure of the syntax-tree of a program.

5.3 Machine Learning Based Techniques

We now discuss two very different techniques from the machine learning community that have applied to program synthesis.

5.3.1 Probabilistic Inference

A program can be modeled as a graph consisting of instructions and states, connected by constraint nodes. Each constraint node establishes the semantics of some instruction by relating the instruction with the state immediately before the instruction and the state immediately after the instruction. Such a modeling allows for use of a probabilistic inference technique known as belief propagation [58] to infer states and/or instructions. [24] uses this modeling to infer abstract states (i.e., invariants) given the program instructions for the purpose of program verification. [36] uses this modeling to infer both the instructions and concrete states (before and after each instruction) that satisfy a given set of concrete input-output examples for the purpose of program synthesis. [36] applies this technique to synthesis of imperative program fragments that execute polynomial computations and list manipulations.

5.3.2 Genetic Programming

Genetic programming is a computational method inspired by biological evolution, which discovers computer programs tailored to a particular task [42]. It maintains a population of individual programs. Computational analogs of biological mutation and crossover produce program variants. Mutation introduces random changes, while crossover facilitates sharing of useful pieces of code between programs being evolved. The referential transparency of functional programs, such as Lisp, makes crossover possible, as the state does not impact the evaluation of the exchanged subexpressions. In case of imperative programs, the meaning of a code fragment depends on the state in which the fragment is executed, hence there is no referential transparency and the crossover can produce faulty code or dead code. These problems may be avoided by discarding variants that do not compile or using a post-processing step to remove dead code. Each variant's suitability is evaluated using a user-defined fitness function, and successful variants are selected for continued evolution. The success of a genetic programming based system crucially depends on the fitness function, which require non-trivial creativity.

Genetic programming has been used to discover mutual exclusion algorithms [38] and to fix bugs in imperative programs [75]. Genetic programming may be an interesting alternative to consider for program synthesis when other conventional techniques fail to provide solutions, or when an approximate solution is acceptable. An interesting prospect is to develop techniques that combine the strengths of genetic programming with that of logical reasoning [35, 38].

5.4 Logical Reasoning Based Techniques

This class of techniques reduce the program synthesis problem to that of solving a SAT/SMT formula and let an off-the-shelf SAT/SMT solver efficiently explore the search space. The idea is to exploit the recent advances made in the Satisfiability (SAT) and Satisfiability Modulo Theory (SMT) solving technology[4] to efficiently explore the search space of programs. The reduction typically involves two main steps:

1. *Constraint Generation*, which generates a logical constraint, referred to as the *synthesis constraint*, such that the solution to the constraint yields the desired program. This is mostly a syntactic process.
2. *Constraint Solving* for solving the synthesis constraint. This involves reducing the generated constraint to a corresponding SAT/SMT constraint that can be solved using off-the-shelf constraint solvers.

We next describe these two steps in a little detail. We rather present a simplistic view; for more details, we refer the reader to work on template based program synthesis [69, 67, 71] and component based program synthesis [23, 33]. The former requires a background into template based program verification [68, 25, 26, 66, 27] with the connection that the synthesis problem is a more general form of the verification problem.

6. Constraint Generation

We illustrate the principle of constraint generation by briefly describing few methodologies for the special case when the user intent is specified in the form of a precondition and postcondition pair $(\text{Pre}, \text{Post})$, which is a form of a logical relation between inputs and outputs, and the search space consists of a single while loop with guard c and body S. Figure 4 briefly summarizes the relative merits and challenges of these methodologies.

These methodologies can be extended to generate constraints for other forms of intent specifications, and more general forms of program structures. Other methodologies may also be possible.

6.1 Invariant-based

This methodology (described in [69]) generates a synthesis constraint that asserts that the unknown program should be such that it behaves correctly on all inputs. It is inspired by proof-based deductive synthesis [51]. It has been applied to synthesis of several undergraduate textbook algorithms such as sorting, dynamic programming, Strassen's matrix multiplication, Bresenham's line drawing [69]. Unlike inductive synthesis based techniques that require an external verifier (in a refinement loop) for checking correctness of the generated program, this methodology guarantees correctness of the generated solution by integrating verification as part of synthesis.

$$\exists I, r, c, S \;\; \forall x, x_1 \left(\begin{array}{rl} \text{Pre} \Rightarrow I & (1) \\ \wedge \;\; (I \wedge \neg c) \Rightarrow \text{Post} & (2) \\ \wedge \;\; (I \wedge c \wedge S) \Rightarrow I_1 & (3) \\ \wedge \;\; I \Rightarrow (r \geq 0) & (4) \\ \wedge \;\; (I \wedge c \wedge S) \Rightarrow (r_1 < r) & (5) \end{array} \right)$$

[4] 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].

Constraint Generation Methodology	Constraint Size	Sophistication of Constraints	Coverage
Invariant-based	Smallest	Most	Full
Path-based	Medium	Medium	Medium
Input-based	Largest	Least	Least

Figure 4. This figure illustrates the trade-off between the following aspects for various constraint generation methodologies: size of generated constraints, sophistication of generated constraints, and level of coverage achieved.

The occurrence of S, as a formula in the above equation, refers to its transition system representation that assigns to fresh version x_1 of original program variables x. I_1 and r_1 refers to the formula obtained from I and r respectively by replacing variables x by x_1.

Eqns 1 and 2 encode that I is an inductive invariant of the while loop (Eqn 1 encodes the base case while Eqn 2 encodes the inductive case). Eqn 3 encodes that I is strong enough to imply the postcondition Post. Eqns 4 and 5 encode that r is a ranking function of the loop, i.e., the loop terminates.

6.2 Path-based

This methodology (described in [67]) generates a synthesis constraint that asserts that the unknown program should be such that it behaves correctly on all inputs that exercise a given set of paths. It combines ideas from symbolic execution based testing and constraint based program analysis. It has been applied to synthesis of program inverses (see Section 2.5 for examples) and client-server applications [67]. Unlike the invariant-based methodology, the path-based methodology does not guarantee that the generated program always meets the desired pre/post specification, and hence requires an external verifier. On the other hand, it leads to simpler constraints that avoid quantification over invariants. The language of invariants is usually far more sophisticated than the program constructs (guards and statements) and usually itself involves quantifiers.

$$\exists c, S \;\; \forall x_0, .., x_n \left(\begin{array}{rl} (\text{Pre}_0 \wedge \neg c_0) \Rightarrow \text{Post}_0 & (6) \\ \wedge \; (\text{Pre}_0 \wedge \phi_1 \wedge \neg c_1) \Rightarrow \text{Post}_1 & (7) \\ \wedge \; (\text{Pre}_0 \wedge \phi_2 \wedge \neg c_2) \Rightarrow \text{Post}_2 & (8) \\ \vdots & \\ \wedge \; (\text{Pre}_0 \wedge \phi_n \wedge \neg c_n) \Rightarrow \text{Post}_n & (9) \end{array} \right)$$

where $\phi_i \equiv (c_0 \wedge S_0 \wedge c_1 \wedge S_1 \wedge \ldots \wedge c_{i-1} \wedge S_{i-1})$. c_i and Post_i are obtained from c and Post respectively by replacing program variables x by x_i. S_i is obtained from S by replacing variables x and x_1 by x_i and x_{i+1} respectively.

Eqn 6 asserts that the condition c and loop body S are such that the while loop satisfies the pre-post specification for all inputs x that do not go through the loop body. Eqns 7, 8, and 9 assert the same thing for all inputs that go through one, two, and n iterations respectively of the loop. A larger number of such constraints (i.e., a larger value of n) ensures more correctness, and in the limit would guarantee full correctness. The hope is that few such constraints would constrain the search space enough that it would lead to a correct solution.

6.3 Input-based

This methodology generates a synthesis constraint that asserts that the unknown program should be such that it behaves correctly on a given set of inputs. It is used when the input constraints are in the form of input-output examples (as in oracle guided synthesis applied to bitvector algorithms and deobfuscation [33]), or in a counterexample guided iterative synthesis technique like sketching [65]. This methodology provides even less coverage than the path-based methodology, but has the advantage of generating constraints that are simplest to solve. In particular, there is no quantification over invariants, and no universal quantification.

$$\exists c, S\left(\psi(P_1, Q_1) \wedge \ldots \wedge \psi(P_k, Q_k)\right) \tag{10}$$

where $\psi(P, Q)$ is as follows:

$$\exists x_0, .., x_n \left(\begin{array}{rl} (\neg c_0 \wedge x_0 = P \wedge x_0 = Q) & (11) \\ \vee\ (\phi_1 \wedge \neg c_1 \wedge x_0 = P \wedge x_1 = Q) & (12) \\ \vee\ (\phi_2 \wedge \neg c_2 \wedge x_0 = P \wedge x_2 = Q) & (13) \\ \vdots & \\ \vee\ (\phi_n \wedge \neg c_n \wedge x_0 = P \wedge x_n = Q) & (14) \\ \vee\ (\phi_n \wedge c_n) & (15) \end{array} \right)$$

ϕ_i and c_i are as defined in Section 6.2. Eqn 10 asserts that the unknown program should be such that it satisfies the input-output pairs $(P_1, Q_1), \ldots, (P_k, Q_k)$. The formula $\psi(P, Q)$ asserts that if the unknown program is executed in the initial state P, then its final state satisfies Q, if it terminates within n iterations of the loop body. A larger value of n would thus achieve more coverage. The conjuncts $\phi_i \wedge \neg c_i$ (in Eqns 12-14) encode that the program should terminate in exactly i iterations of the loop, while the conjuncts $x_0 = P \wedge x_i = Q$ encode that the initial state should be P and the output state should be Q. Eqn 15 encodes that the program executes more than n loop iterations.

7. Constraint Solving

The synthesis constraints generated (in Section 6) are second-order logic formulas with universal quantification. Unfortunately, these constraints cannot be solved using off-the-shelf constraint solvers (such as SAT/SMT solvers) because of two challenges: presence of second-order unknowns, and presence of universal quantification. Next, we describe few techniques for eliminating second order unknowns (Section 7.1) and universal quantification (Section 7.2). This allows for use of off-the-shelf constraint solvers to solve the resultant constraints, which yields the desired program.

7.1 Reducing Second-order Unknowns to First-order Unknowns

The key idea here is to assume some template structure for the various second-order unknowns (e.g., invariants, conditional guards, and assignment statements) in the synthesis constraint. The templates are such that the holes/unknowns in the templates are only first-order entities. This reduces the second-order constraints to first-order constraints over the unknown parameters of the templates. Examples of templates for invariants include boolean combination of arithmetic constraints [25, 27], boolean combination of predicates from a given set [26], quantified formulas over a given set of predicates [66]. Examples of templates for guards and statements include those that involve arithmetic and predicate abstraction [69, 67, 71].

7.2 Universal Quantifier Elimination

The synthesis constraint also contains universal quantification over the program variables. We describe below few options for eliminating universal quantifiers.

7.2.1 Domain-specific Methods

Farkas Lemma is a beautiful concept for reasoning about the theory of linear arithmetic and can be used to translate universal quantifiers into existential quantifiers [25]. For the theory of predicate abstraction, we can use *cover algorithms* to eliminate universal quantifiers [26, 66].

7.2.2 Sampling

Sampling is a generic method for approximating a universally quantified formula of the form $\forall x\ \phi(x)$ by $\phi(P_1) \wedge \phi(P_2) \wedge \ldots \wedge \phi(P_k)$ for few values $P_1, \ldots, P_k$. The values $P_1, \ldots, P_k$ can be chosen using various means described below.

Counterexample Driven This is a general approach that requires an external verifier that either validates the correctness of a given solution to the synthesis constraint against the user intent, or provides a new counterexample. The new counterexample yields a new value P_i. This approach forms the basis of counterexample guided iterative synthesis technique [19, 62], which involves choosing some initial set of test values for the universally quantified variables and then solving for the 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 approach is used in [65, 23, 31].

Random In this approach, P_i's are simply chosen randomly. This approach may work well for synthesis of programs whose correctness can be verified using randomly chosen inputs (with high probability over the choice of those inputs). In other words, this approach works well for programs for which a few randomly chosen inputs can act as a full functional specification of the program (with high probability over the choice of the random inputs). This includes polynomials [61] and free boolean graphs [9].

Biased Not all inputs in the input space are equally important. In certain domains, combining some form of bias with randomness works better compared to simply using pure randomness. For example, a program may take an integer input i, but have the same behavior for all $i > 5$ and have interesting behaviors only on values $0 \leq i \leq 5$. For many applications, the user knows apriori which inputs are more crucial in defining the overall program.

Biased sampling works well for synthesis of bitvector programs, which combine fixed-width arithmetic operations with bitwise operations [33]. The input space for bitvector programs consists of all (tuples of) bitvectors of a certain bit width. It is well-known that, for a very large class of commonly-used bitvector functions, the rightmost bits influence the output more than the leftmost bits.

PROPERTY 1 (See [74], Chapter 2). *A function mapping bitvectors to bitvectors can be implemented with add, subtract, bitwise and, bitwise or, and bitwise not instructions if and only if each bit of the output depends only on bits at and to the right of that bit in each input operand.*

This suggests biasing the sampling so that there is more variety on the rightmost bits. For example, if 4 values can be selected, then this should include bitvectors whose rightmost two bits include all possible four combinations: 00, 01, 10, and 11, while the leftmost bits can be kept random.

Basis For some class of programs, the behavior of a program is fully specified by its behavior on a certain subset of its inputs, referred to as *basis inputs*. For example, the behavior of a comparison machine (programs that only use comparison operators) is determined by its behavior on boolean arrays [41]. The behavior of functions over a vector space is determined by their behavior over any basis of the vector space. For such programs, we can restrict sampling to the basis inputs.

8. Conclusion

Program Synthesis has the potential to bring about a revolution in computing. Computing has become accessible to people at an impressive rate over the last few years. However, the fundamental programming model has not changed much since the last several years. Program Synthesis has the potential to make the computer understand our "what" language, as opposed to us interacting with the computer in its language of step-by-step detailed instructions on "how" to accomplish the desired task.

The true success of program synthesis would require a multi-disciplinary effort for each of the dimensions in program synthesis. For the first dimension of understanding user intent, *Human Computer Interface* can help in identifying the most natural interface for solving problems in a particular domain. *Natural Language Processing* can also play a key role since natural language could be an integral part of most interfaces. The second dimension of selecting useful space/domain of programs for synthesis (which is usually limited by availability of reasoning technology for those programs) can benefit from domain expertise in corresponding disciplines such as *Information Extraction* (for text manipulating programs), *Graphics* (for image manipulating programs), and *Programming Languages* (for data-structure manipulating programs). For the third dimension of searching a program over a given space, *Logical Reasoning* and *Machine Learning* communities can play a big role.

Research Questions Following are some important research questions for developing next generation of program synthesis tools and techniques.

- How do we combine various forms of user intent (such as logical specifications, natural language, input-output examples, traces, and higher-order, inefficient, or partial programs) in a unified programming interface?
- How do we ensure a modular architecture that would allow reuse of various synthesis components including domain knowledge (both facts and rules for manipulating objects such as text, images, data-structures) and search techniques across different synthesis tools and applications?
- How do we combine the power of various search techniques such as version space algebras, logical reasoning techniques (based on abstraction, mathematical induction, theorem proving, SAT/SMT/QBF solving) and machine learning techniques (probabilistic inference, bayesian learning, genetic programming) for automated program synthesis from user intent? How do we further incorporate quantitative program analysis techniques [28, 11] to produce programs that are not just functionally correct, but also meet performance and resource availability constraints?

Acknowledgments

I am grateful to my family members who let me work on this paper during my vacation that was long over-due. I would like to thank my collaborators on various program synthesis projects, some of which are referenced in this paper: Vijay Anand, Ras Bodik, Swarat Chaudhuri, Jeff Foster, Neil Immerman, Susmit Jha, Nebojsa Jojic, Shachar Itzhaky, Francesco Logozzo, Madhusudan Parthasarthy, Chris Quirk, Mooly Sagiv, Sanjit Seshia, Rishabh Singh, Armando Solar-Lezama, Saurabh Srivastava, Nikhil Swamy, Ashish Tiwari, Ramarathnam Venkatesan, Ben Zorn.

References

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

[2] The Aha! (a hacker's assistant) superoptimizer, 2008. Download: `http://www.hackersdelight.org/aha.zip`, Documentation: `http://www.hackersdelight.org/aha/aha.pdf`.

[3] I. Androutsopoulos, G. Ritchie, and P. Thanisch. Natural language interfaces to databases — an introduction. *Natural Language Engineering*, 1(1):29–81, 1995.

[4] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2):87–106, 1987.

[5] Y. Bar-David and G. Taubenfeld. Automatic discovery of mutual exclusion algorithms. In *DISC*, pages 136–150, 2003.

[6] S. Barmany, R. Bodik, S. Chandra, J. Galensony, D. Kimelman, C. Rodarmory, and N. Tungy. Programming with angelic nondeterminism. In *POPL*, pages 339–351, 2010.

[7] A. W. Biermann, R. I. Baum, and F. E. Petry. Speeding up the synthesis of programs from traces. *IEEE Trans. Computers*, 24(2):122–136, 1975.

[8] A. W. Biermann and J. A. Feldman. On the synthesis of finite-state machines from samples of their behavior. *IEEE Trans. Comput.*, 21(6):592–597, 1972.

[9] M. Blum, A. K. Chandra, and M. N. Wegman. Equivalence of free boolean graphs can be decided probabilistically in polynomial time. *Inf. Process. Lett.*, 10(2):80–82, 1980.

[10] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. In *IJCAI*, pages 1004–1009, 2009.

[11] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *POPL*, pages 57–70, 2010.

[12] K. Claessen, N. Smallbone, and J. Hughes. QuickSpec: Guessing formal specifications using testing. In *Tests and Proofs, Fourth International Conference, TAP 2010*. To appear.

[13] A. Cypher, D. C. Halbert, D. Kurlander, H. Lieberman, D. Maulsby, B. A. Myers, and A. Turransky, editors. *Watch what I do: programming by demonstration*. MIT Press, Cambridge, MA, USA, 1993.

[14] C. de la Higuera. A bibliographical study of grammatical inference. *Pattern Recognition*, 38(9):1332 – 1348, 2005. Grammatical Inference.

[15] C. de la Higuera, P. W. Adriaans, M. van Zaanen, and J. Oncina, editors. *Proceedings of the Workshop and Tutorial on Learning Context-Free Grammars*. Ruder Boskovic Institute, Zagreb, Croatia, 2003.

[16] E. W. Dijkstra. Program inversion. In *Program Construction,* `http://www.cs.utexas.edu/~EWD/ ewd06xx/EWD671.PDF`, pages 54–57. Springer-Verlag, 1979.

[17] E. W. Dijkstra. Solutions of a problem in concurrent programming control (reprint). *Commun. ACM*, 26(1):21–22, 1983.

[18] R. Glück and M. Kawabe. A method for automatic program inversion based on LR(0) parsing. *Fundam. Inf.*, 66(4):367–395, 2005.

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

[20] S. A. Goldman and M. J. Kearns. On the complexity of teaching. *Journal of Computer and System Sciences*, 50:20–31, 1995.

[21] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *PLDI*, 1992.

[22] M. Gualtieri. Deputize end-user developers to deliver business agility and reduce costs. In *Forrester Report for Application Development and Program Management Professionals*, April 2009.

[23] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Component based synthesis applied to bitvector circuits. Technical Report MSR-TR-2010-12, Microsoft Research, 2010.

[24] S. Gulwani and N. Jojic. Program verification as probabilistic inference. In *POPL*, pages 277–289, 2007.

[25] S. Gulwani, S. Srivastava, and R. Venkatesan. Program analysis as constraint solving. In *PLDI*, pages 281–292, 2008.

[26] S. Gulwani, S. Srivastava, and R. Venkatesan. Constraint-based invariant inference over predicate abstraction. In *VMCAI*, pages 120–135, 2009.

[27] S. Gulwani and A. Tiwari. Constraint-based approach for analysis of hybrid systems. In *CAV*, pages 190–203, 2008.

[28] S. Gulwani and F. Zuleger. The reachability-bound problem. In *PLDI*, 2010. To appear.

[29] B. A. Hockey and M. Rayner. Using paraphrases of deep semantic represention to support regression testing in spoken dialogue systems. In *Proceedings of the Workshop on Software Engineering, Testing, and Quality Assurance for Natural Language Processing (SETQA-NLP 2009)*, pages 14–21, Boulder, Colorado, June 2009. Association for Computational Linguistics.

[30] N. Immerman. Upper and lower bounds for first order expressibility. *J. Comput. Syst. Sci.*, 25(1):76–98, 1982.

[31] S. Itzhaky, S. Gulwani, N. Immerman, and M. Sagiv. A simple inductive synthesis methodology and its applications. Technical Report MSR-TR-2010-35, Microsoft Research, April 2010.

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

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

[34] S. Jha, S. Gulwani, S. Seshia, and A. Tiwari. Synthesizing switching logic for safety and dwell-time requirement. In *ICCPS*, 2010.

[35] C. G. Johnson. Genetic programming with fitness based on model checking. In *EuroGP*, pages 114–124, 2007.

[36] V. Jojic, S. Gulwani, and N. Jojic. Probabilistic inference of programs from input/output examples. Technical Report MSR-TR-2006-103, Microsoft Research, 2006.

[37] S. Katayama. Systematic search for lambda expressions. In *Trends in Functional Programming*, 2005.

[38] G. Katz and D. Peled. Genetic programming and model checking: Synthesizing new mutual exclusion algorithms. In *ATVA*, pages 33–47, 2008.

[39] D. E. Knuth. *The Art of Computer Programming, Volume IV*. `http://www-cs-faculty.stanford.edu/~knuth/taocp.html`.

[40] D. E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms, 2nd Edition*. Addison-Wesley, 1973.

[41] D. E. Knuth. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley, 1973.

[42] J. R. Koza. *Genetic programming: on the programming of computers by means of natural selection*. MIT Press, Cambridge, MA, USA, 1992.

[43] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI*, 2010. To appear.

[44] T. Lau. *Programming by Demonstration: a Machine Learning Approach*. PhD thesis, University of Washington, Seattle, 2001.

[45] T. Lau, L. Bergman, V. Castelli, and D. Oblinger. Programming shell scripts by demonstration. In *Workshop on Supervisory Control of Learning and Adaptive Systems, AAAI*, 2004.

[46] T. A. Lau, P. Domingos, and D. S. Weld. Version space algebra and its application to programming by demonstration. In *ICML*, pages 527–534, 2000.

[47] T. A. Lau, P. Domingos, and D. S. Weld. Learning programs from traces using version space algebra. In *K-CAP*, pages 36–43, 2003.

[48] M. Li and P. M. B. Vitanyi. *An Introduction to Kolmogorov Complexity and Its Applications*. Springer-Verlag, Berlin, 1993.

[49] Y. Li, H. Yang, and H. V. Jagadish. Nalix: an interactive natural language interface for querying xml. In *SIGMOD*, pages 900–902, 2005.

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

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

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

[53] T. M. Mitchell. Generalization as search. *Artif. Intell.*, 18(2):203–226, 1982.

[54] MyDoom Wikipedia Article, URL accessed Sep. 2009.

[55] R. P. Nix. Editing by example. *ACM Trans. Program. Lang. Syst.*, 7(4):600–621, 1985.

[56] J. Oncina, P. García, and E. Vidal. Learning subsequential transducers for pattern recognition interpretation tasks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 15(5):448–458, 1993.

[57] M. Pardowitz, B. Glaser, and R. Dillmann. Learning repetitive robot programs from demonstrations using version space algebra. In *RA '07: Proceedings of the 13th IASTED International Conference on Robotics and Applications*, pages 394–399. ACTA Press, 2007.

[58] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, 1988.

[59] P. Porras, H. Saidi, and V. Yegneswaran. An analysis of conficker's logic and rendezvous points. Technical report, SRI International, March 2009.

[60] S. Schrödl and S. Edelkamp. Inferring flow of control in program synthesis by example. In *KI '99: Proceedings of the 23rd Annual German Conference on Artificial Intelligence*, pages 171–182. Springer-Verlag, 1999.

[61] J. T. Schwartz. Fast probabilistic algorithms for verification of polynomial identities. *J. ACM*, 27(4):701–717, 1980.

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

[63] M. Smellie. String generalisation for editing by example. `http://www.mcs.vuw.ac.nz/comp/graduates/archives/honours/1997/michael-smellie.ps.gz`, 1997.

[64] D. R. Smith. KIDS – a semi-automatic program development system. *IEEE TSE*, 16(9):1024–1043, 1990.

[65] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, University of California, Berkeley, 2008.

[66] S. Srivastava and S. Gulwani. Program verification using templates over predicate abstraction. In *PLDI*, pages 223–234, 2009.

[67] S. Srivastava, S. Gulwani, S. Chaudhuri, and J. Foster. Program inversion revisited. Technical Report MSR-TR-2010-34, Microsoft Research, 2010.

[68] S. Srivastava, S. Gulwani, and J. S. Foster. VS3: SMT solvers for program verification. In *CAV*, pages 702–708, 2009.

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

[70] Symantec Corporation. Internet security threat report volume XIV. `http://www.symantec.com/business/theme.jsp?themeid=threatreport`, April 2009.

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

[72] M. T. Vechev and E. Yahav. Deriving linearizable fine-grained concurrent objects. In *PLDI*, pages 125–135, 2008.

[73] J. M. Vilar. Query learning of subsequential transducers. In *ICG '96: Proceedings of the 3rd International Colloquium on Grammatical Inference*, pages 72–83. Springer-Verlag, 1996.

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

[75] W. Weimer, T. Nguyen, C. L. Goues, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE*, pages 364–374, 2009.

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

[77] L. S. Zettlemoyer and M. Collins. Learning context-dependent mappings from sentences to logical form. In *ACL-IJCNLP*, pages 976–984, 2009.