A Simple Inductive Synthesis Methodology and its Applications

Shachar Itzhaky
Tel-Aviv University
shachar@tau.ac.il

Sumit Gulwani
Microsoft Research
sumitg@microsoft.com

Neil Immerman
University of Massachusetts
immerman@cs.umass.edu

Mooly Sagiv
Tel-Aviv University
msagiv@acm.org

Abstract

Given a high-level specification and a low-level programming language, our goal is to automatically synthesize an efficient program that meets the specification. In this paper, we present a new algorithm and methodology for inductive synthesis that allows us to do this.

We use Second Order logic as our generic high level specification logic. For our low-level languages we choose small application-specific logics that can be immediately translated into code that runs in expected linear time in the worst case.

We explain our methodology and provide examples of the synthesis of several graph classifiers, e.g, linear-time tests of whether the input graph is connected, acyclic, etc. In another set of applications we automatically derive many finite differencing expressions equivalent to ones that Paige built by hand in his thesis [Pai81]. Finally we describe directions for automatically combining such automatically generated building blocks to synthesize efficient code implementing more complicated specifications.

The methods in this paper have been implemented in Python using the SMT solver Z3 [dMB].

Keywords: Transformational Programming, Finite Differencing, High Level Program, Inductive Synthesis

1. Introduction

We describe an algorithmic methodology that takes a high level specification and a low-level programming language, and searches for an efficient implementation of the specification. If there is no such implementation then our methodology reports failure.

The input is a program in a declarative and expressive programming language. A naive syntax-directed translation of this program into machine code usually produces an inefficient and sometimes exponential-time implementation. Instead, we automatically convert the input specification into an equivalent low-level specification. Using a simple syntax-directed translation, the resulting program is then converted into imperative code that is guaranteed run in expected linear time in the worst case.

The high level specifications are written in subsets of second-order logic (SO), second-order existential logic (SO$\exists$) which by Fagin's Theorem expresses exactly those properties checkable in NP, and first-order logic plus transitive closure (FO(TC)) which expresses exactly the properties checkable in NSPACE$[\log n]$ [Imm99].

We describe two simple target languages, both of which can express only linear-time properties, one for expressing graph properties, and another for expressing properties of sets.

Of course, our algorithm cannot succeed if there is no program in the target logic that implements the specification in question. However, when there is such an implementation, we do succeed in a large number of cases.

In the first problem domain we automatically generate linear-time graph classifiers for many simple properties such as connectivity, acyclicity, etc. In the second domain, we automatically generate constant-time finite differencing expressions for many of the examples that Paige worked out by hand in his Ph.D. thesis [Pai81].

Our methods are fairly simple and general. We use the SMT solver Z3 to generate structures satisfying the specifications. Using these models we learn candidate implementations in the low-level languages that are correct on the generated structures. Then we use the solver repeatedly to see if there exist any other examples that satisfy the original specifications but for which the candidate implementation fails. When there are no further counterexamples, we are done.

We feel that it is surprising how well this simple method works to find asymptotically optimal algorithms which are implied by the high-level specifications, but definitely not obvious from the specifications. Indeed a naive implementation of the high-level specifications would lead to very inefficient algorithms.

While many people have tried to do automatic synthesis, much of our inspiration comes from the work of Bob Paige and Jack Schwartz on automatically generating efficient data structures and algorithms for specifications written in the very high-level programming language SETL [Pai81]. An important analogy of our work is the automatic optimization of SQL, in which the specification in SQL (essentially first-order logic plus counting, FO(COUNT) [Imm99, Thm 14.9]) is transformed into a logically equivalent specification, also in SQL but with better run time. In our setting, since the specification language is SO – a language exponentially more powerful than FO – the possible transformations include a much wider range of possibilities and difficulties.

This paper is organized as follows: §2 provides background and definitions. §3 explains our synthesis methodology at a high level. In §4 we explain the details of an implementation of the methodology. §5 shows our first applications: learning efficient code to check whether input graphs have certain properties. In §6 we automatically generate numerous examples of finite differencing code equivalent to code that Paige worked out by hand in his Ph.D. thesis [Pai81]. In §7 we discuss possible extensions of this work to employ known efficient data structures, and to use the finite differencing methods of §6 to synthesize efficient implementations of various graph algorithms. In §8 we discuss some related work. In §9 we conclude and suggest some future directions of this work.

2. Background

We use standard notation from mathematical logic. For background in descriptive complexity we recommend [Imm99]. All our logical structures are finite and ordered. For example, an ordered graph, G, is a finite logical structure: $G = ([n], E)$, whose universe of vertices is the set $|G| = [n] = \{0, 1, \ldots n-1\}$, and whose edge relation is a subset of the set of ordered pairs of vertices: $E \subseteq |G|^2$. In this case the vocabulary of G consists of a single binary relation symbol, $\sigma = \{E^2\}$, which we sometime describe as an input relation symbol.

We use STRUC$[\sigma]$ to denote the set of all finite, ordered structures of vocabulary σ, and STRUC$_{\leq k}[\sigma]$ denotes the subset of these structures with universe size at most k. If $\sigma' \supset \sigma$ is a larger vocabulary, and if $\mathcal{A} \in$ STRUC$[\sigma]$ and $\mathcal{A}' \in$ STRUC$[\sigma']$ is identical to $\mathcal{A}$ except that it also interprets the symbols of $\sigma' - \sigma$ then we say that $\mathcal{A}'$ is an *expansion* of $\mathcal{A}$ to σ' and we write $\mathcal{A} < \mathcal{A}'$. If $\mathcal{A} \in$ STRUC$[\sigma]$ and α, φ are formulas of vocabulary σ, then we write $\mathcal{A} \models \varphi$ to mean that $\mathcal{A}$ satisfies φ, and $\alpha \vdash \varphi$ to mean that there is a proof of φ from assumption α. We write $\alpha \equiv \varphi$ to mean that α is equivalent to φ, i.e., $\alpha \vdash \varphi$ and $\varphi \vdash \alpha$.

The logical languages we consider include first-order logic, FO; first-order logic plus transitive closure, FO(TC) (the closure of first-order logic under the transitive closure operation, i.e, if we can express the binary relation R, then we can also express its transitive closure, R^+, and its reflexive, transitive closure, $R^\star$); first-order logic plus a least-fixed-point operator, FO(LFP) (The least-fixed-point operator formalizes the process of making inductive definitions, so FO(LFP) is the closure of first-order logic under the ability to define new relations by induction.); second-order existential logic, SO$\exists$; and full second-order logic, SO. We assume that these languages have access to the numeric ordering relation $(\leq)$ and the numeric constant symbols, $0, 1, \ldots, \max$.

It is well known that natural logical languages capture natural complexity classes, for example, FO $=$ CRAM$[1]$, FO(TC) $=$ NSPACE$[\log n]$, FO(LFP) $=$ P, SO$\exists$ $=$ NP, and SO $=$ PH. Here CRAM$[1]$ is the set of problems checkable in constant time by a parallel random access machine with polynomially much hardware, PH is the polynomial-time hierarchy.

Thus, SO is an extremely rich, very expressive algorithmic language . Therefore, we decided use SO as the main input to our tool. In the future we plan to develop an equally expressive, but more user-friendly specification language.

3. A Methodology for Inductive Synthesis

We start with a specification $\varphi \in$ SO and a target language L. We are hoping to derive an $\alpha \in L$ such that $\alpha \equiv \varphi$. Thus α will be an efficient implementation of the specification, φ. We have an input vocabulary, σ, which contains all of the numeric and input predicate, constant, and function symbols. For example, the numeric symbols might be $0, 1, \max, \leq^2$. The input symbols might be E^2, r, for a rooted graph with constant symbol, r, denoting a specified root node. The output vocabulary σ' consists of σ together with the output symbols. For example, for topological sort, σ' would include the output function symbol, g^1, that denotes a topological ordering, and for the minimum spanning tree algorithm, σ', would include the relation symbol, T^2, denoting the tree edges. In the finite differencing examples, we usually have input relations symbols such as E^2, S^1, and their new values one step later as the output, i.e., E'^2, S'^1. Note that the exponents denoting arity are shown in the vocabularies, but are omitted when these symbols are used in formulas.

Our synthesis algorithm works in two settings both of which come up often in practice. Sometimes we are asked to compute a well-defined single valued function, e.g., connected components, strongly connected components, etc., in this case, the desired answer is fully specified. However, sometimes the problem corresponds to a relation and any answer — or the answer that there is no solution — is what is desired, e.g., minimum spanning tree, topological sort, depth-first search, max network flow, and, of course, SAT. In this case, the answer is only partially specified. The two cases are distinguished as follows:

- φ is a *full specification*. In this setting, for all $\mathcal{A} \in$ STRUC$[\sigma]$ there exists a unique expansion $\mathcal{A} < \mathcal{A}' \in$ STRUC$[\sigma']$ such that $\mathcal{A}' \models \varphi$. In this case we wish to synthesize a formula $\alpha \in L$ such that $\alpha \equiv \varphi$.
- φ is a *partial specification*. In this setting, for all $\mathcal{A} \in$ STRUC$[\sigma]$ there exists at least one expansion $\mathcal{A} < \mathcal{A}' \in$ STRUC$[\sigma']$ such that $\mathcal{A}' \models \varphi$. In this case we wish to synthesize a formula $\alpha \in L$ such that $\alpha \vdash \varphi$ and for all $\mathcal{A} \in$ STRUC$[\sigma]$ there exists at least one expansion $\mathcal{A} < \mathcal{A}' \in$ STRUC$[\sigma']$ such that $\mathcal{A}' \models \alpha$.

While our methodology can handle both cases, all the examples in this paper are full specifications. Thus we make the exposition simpler by assuming we are always given a full specification $\varphi \in$ SO. We first generate a set of instance structures $\mathcal{M} = \{\mathcal{A}_1, \ldots \mathcal{A}_k\} \subset$ STRUC$_{\leq n}[\sigma']$ such that $\mathcal{M} \models \varphi$. Here k, n, the initial number of structures, and the upper bound on the number of elements in the universe of each structure, are parameters.

In this paper we make the simplifying assumption that our target language L consists of conjunctions from the set of *base formulas*, B. We first compute a set of *good formulas*, $G \subseteq B$, having the property that $\mathcal{M} \models G$, i.e., every instance structure satisfies every good formula.

Next we use a greedy algorithm to compute a minimal cover w.r.t. $\mathcal{M}$, $C \subseteq G$. C is a *cover* w.r.t. $\mathcal{M}$ iff $\mathcal{M} \models C$ and for all $\mathcal{A} \in \mathcal{M}$, C determines all the output bits of $\mathcal{A}$. In symbols,

$$\mathcal{M} \models C \quad \wedge \quad \left(\bigwedge_{c \in C} c\right) \wedge \Delta_\sigma(\mathcal{A}) \vdash \Delta_{\sigma'}(\mathcal{A}) \;. \qquad (1)$$

Here, $\Delta_\tau(\mathcal{A})$, is the diagram of $\mathcal{A}$, i.e., the conjunction of all ground literals (from the vocabulary τ together with constants for the elements of the universe of $\mathcal{A}$) that are satisfied by $\mathcal{A}$.

The greedy algorithm incrementally chooses C by successively choosing an element of G that determines the maximum number of output bits not yet determined by C, and then adding it to C.

Once we have such a cover, C, it gives us a candidate $\alpha = \bigwedge_{c \in C} c$. To determine whether α is a correct candidate, we ask two questions of the SMT solver.

1. $(\exists \mathcal{A} \in \text{STRUC}_{\leq n}[\sigma'])(\mathcal{A} \models \varphi \wedge \neg\alpha)$
2. $(\exists \mathcal{A} \in \text{STRUC}_{\leq n}[\sigma'])(\mathcal{A} \models \varphi \wedge (\alpha \wedge \Delta_\sigma(\mathcal{A}) \nvdash \Delta_{\sigma'}(\mathcal{A}))$

That is, (1) Is there a small instance for which α is not good? and (2) Is there a small instance for which α does not determine the answer? If the answer to either of these questions is, "yes", then we add the instance structure to $\mathcal{M}$, and repeat the above construction.

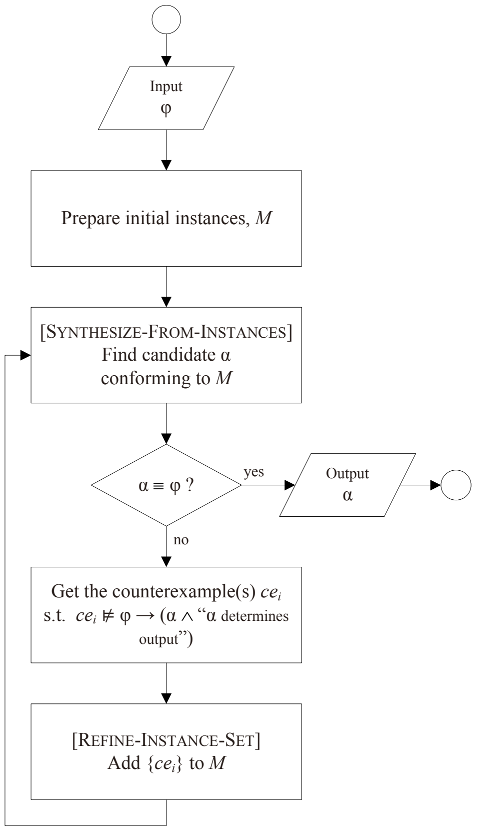

Input
φ
Prepare initial instances, M
[SYNTHESIZE-FROM-INSTANCES]
Find candidate α
conforming to M
α ≡ φ ?
yes
Output
α
no
Get the counterexample(s) ce_i
s.t. $ce_i \nvDash \varphi \rightarrow (\alpha \wedge$ "α determines output")
[REFINE-INSTANCE-SET]
Add $\{ce_i\}$ to M

Figure 1. An abstract flowchart of the synthesis algorithm.

4. An Implementation of the Methodology

We now explain our more detailed algorithm that implements the above methodology. Fig. 1 shows a flowchart of the algorihtm, and pseudocode is shown in Fig. 2. We now fill in the details of each step. The main phases of the algorithm are:

1. Generate $\mathcal{M} \subset$ STRUC$_{\leq n}[\sigma']$, a set of instance structures that all satisfy the specification φ.
2. Find a cover C w.r.t. $\mathcal{M}$ (Eqn. 1).
3. Let $\alpha = \bigwedge_{c \in C} c$. Test whether there is a new instance structure, i.e., $\mathcal{A} \in$ STRUC$_{\leq n}[\sigma']$ and $\mathcal{A} \models \varphi$, such that α is not good for $\mathcal{A}$, or α does not determine the answer for $\mathcal{A}$. If so, add $\mathcal{A}$ to $\mathcal{M}$ and repeat, otherwise output α.

```
Inductive-Synthesis$_L(\varphi)$
  $\mathcal{M}$ := GENERATE-INSTANCES($\varphi$);
  do {
      $t$ := Synthesize-From-Instances$_L(\mathcal{M})$;
  }
  while (Refine-Query($\varphi$, t, $\mathcal{M}$))
```

Figure 2. The main counterexample-guided refinement loop for synthesizing efficient programs.

4.1 Generate the Instances

In both the first and the last step we are required to search for models of the specification φ. For all examples in this paper, the specifications are fully characterized by finite structures, and in fact fairly small such structures. We will fix a parameter, n, throughout this paper to bound the size of these structures. For all the examples we have tested so far, $n = 10$ has been sufficient. Once we have a formula α that passes the test in phase (3), we have found no further counterexamples of size ≤ 20.

We use the subset SO$\exists$ of SO to express our specifications. By Fagin's Theorem (SO$\exists$ = NP) it follows that we can phrase the searches for structures in (1) and (3) as single calls to a SAT solver [Imm99, Thms 7.8, 7.16]. The target languages, L, that we use are always a small subset of SO$\exists \cap$ SO$\forall$ and thus queries that involve φ and α, or $\neg\alpha$, can still be translated to a single instance of SAT, see §A for details.

4.2 Synthesize the Cover

Given the set of instances, $\mathcal{M}$, we must build a cover, C, satisfying Eqn. 1. In the two problem domains explored in this paper, the target languages, L_1 in §5 and L_2 in §6, are sets of conjunctions of *base formulas*, $B_1 \subset L_1$, $B_2 \subset L_2$.

We say that a formula, β, is *good* iff $\mathcal{M} \models \beta$, i.e., it satisfies all the current instances. Let $\mathcal{G}$ denote the set of good base formulas and assume for the sake of discussion that $\mathcal{G}$ is finite.

Now, if there is any cover, then $\gamma = \bigwedge_{\beta\in\mathcal{G}} \beta$ is a cover because it is the strongest good formula. However, note that γ would not be an appropriate candidate for α because typically $\mathcal{G}$ and thus γ will be huge. Furthermore, γ would probably include too much information about the particular chosen instances instead of the specification φ that they all satisfy.

Instead it makes sense to use the principle of Occam's razor, and search for the smallest cover, i.e., the smallest good formula that determines the output for all the given instances.

Recall from §2 that the universe of each structure of size n is $[n] = \{0, \dots, n-1\}$ and that we have the numeric constants, $0, 1, \dots, \max$, available. Assume for simplicity in the following discussion that the output vocabulary $\sigma' - \sigma$ consists of a single unary function symbol, f. (Additional function symbols would be treated similarly, and relations would be treated as boolean functions.)

```
Synthesize-From-Instances$_L(\mathcal{M})$
  Good := $\{\beta \mid \beta$ a base formula of $L, \forall\mathcal{A} \in \mathcal{M}\ \mathcal{A} \models \beta\}$
  Find a minimal subset
  $C \subset$ Good such that $\bigwedge_{c\in C} c$ determines all output bits;
  return $C$
```

Figure 3. Compute Minimal Cover.

```
Refine-Query$_L(\varphi, \alpha, \mathcal{M})$
  example := GENERATE-INSTANCE($\alpha$ not good or does
  not determine answer)
  if (example $\neq$ null) add example to $\mathcal{M}$
  return (example $\neq$ null)
```

Figure 4. Procedure to Generate New Instances.

Let $\mathcal{A} \in$ STRUC$[\sigma']$, $i \in |\mathcal{A}|$ and let $\alpha \in L$. We say that α *determines* $f(i)$ for $\mathcal{A}$ if there is a value $j \in |\mathcal{A}|$ such that

$$\alpha \ \wedge \ \Delta_\sigma(\mathcal{A}) \vdash f(i) = j$$

The following proposition is just a restatement of the cover property (Eqn. 1):

PROPOSITION 4.1. *If for all $\mathcal{A} \in \mathcal{M}$, $\mathcal{A} \models \alpha$ and for all $i \in |\mathcal{A}|$, α determines $f(i)$ for $\mathcal{A}$, then α is a cover.*

It is relatively straightforward to keep track of which output bits are determined by the current set, C, of base formulas. The process SYNTHESIZE-FROM-INSTANCES is defined in Fig. 3. It uses a greedy algorithm to find a minimal cover, i.e., it chooses a good base formula that adds the largest number of points to the determined set and adds that formula to C until C is a cover. Note that it could be the case that there is no cover. This can only happen if there is no formula $\alpha \in L$ that is equivalent to φ. In this case our procedure will of course fail.

4.3 Refine-Query

An α found by the previous phase may still be incorrect since it is only guaranteed to produce correct results for the instances represented by $\mathcal{M}$. To test for the existence of instances outside of $\mathcal{M}$ for which α violates the specification φ, we use the subroutine GENERATE-INSTANCE again, but instead of just trying to find instances satisfying the specification, we attempt to find models of φ that do not satisfy α, or for which α does not determine the answer. This is implemented in the procedure `Refine-Query` shown in Fig. 4.

5. Deriving Graph Classifiers

In this section, we apply the methodology of §3 to derive expected linear-time algorithms for checking properties of directed graphs. Such algorithms have many applications. For

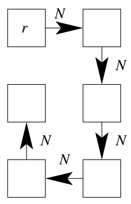
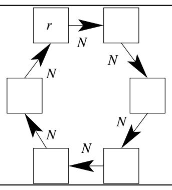
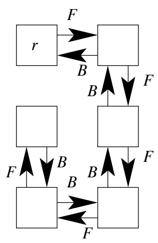
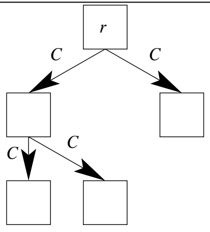
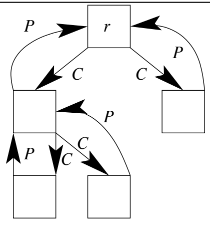
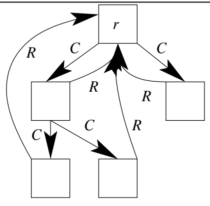
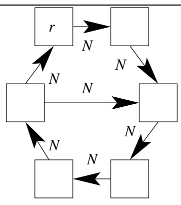
Name	Example	Input Specification (+ Integrity Constaints)	Synthesized Formula	Inst.	Time
SLL Singly Linked List		1:1 $N \quad \wedge$ $\forall u(\neg N^{+}(u,u))$ $\left(\begin{array}{c}\text{root } r \text{ via } N\\ \text{functional } N\end{array}\right)$	$\#p_N(r) = 0 \wedge$ $\forall v(\#p_N(v) \leq 1)$	4	14 sec
CYCLE Cyclic Linked List		$\forall u, v(N^{\star}(u,v))$ $\left(\begin{array}{c}\text{root } r \text{ via } N\\ \text{functional } N\end{array}\right)$	$\#p_N(r) = 1$	4	11 sec
DLL Doubly Linked List		1:1 $F \quad \wedge \quad$ 1:1 $B \quad \wedge$ $\forall u, v\big((F(u,v) \leftrightarrow B(v,u))$ $\wedge \quad \neg F^{+}(u,u)\big)$ $\left(\begin{array}{c}\text{root } r \text{ via } F\\ \text{functional } F, B\end{array}\right)$	$\#p_F(r) = 0 \wedge$ $\forall v(s_F(v) = p_B(v))$	18	149 sec
TREE Directed Tree		1:1 $C \quad \wedge$ $\forall u(\neg C(u,r))$ $\big(\ \text{root } r \text{ via } C\ \big)$	$\#p_C(r) = 0 \wedge$ $\forall v(\#p_C(v) \leq 1)$	8	21 sec
TREEPP Tree with Parent Ptr.		1:1 $C \quad \wedge$ $\forall u, v\big((C(u,v) \leftrightarrow P(v,u))$ $\wedge \quad \neg C(u,r)\big)$ $\left(\begin{array}{c}\text{root } r \text{ via } C\\ \text{functional } P\end{array}\right)$	$\#s_P(r) = 0 \wedge$ $\forall v(s_P(v) = p_C(v))$	26	181 sec
TREERP Tree with Root Ptr.		1:1 $C \quad \wedge$ $\forall u, v\big(\neg C(u,s) \wedge \neg R(r,u)$ $\wedge\ (u \neq s \rightarrow R(u,r))\big)$ $\left(\begin{array}{c}\text{root } r \text{ via } C\\ \text{functional } R\end{array}\right)$	$\#p_C(r) = 0 \wedge$ $p_R(r) = s_{C^{+}}(r) \wedge$ $\forall v(\#p_C(v) \leq 1)$	48	359 sec
SC Strongly Connected		$\forall u, v\big(E^{\star}(u,v)\big)$ $\big(\ \text{root } r \text{ via } E\ \big)$	$p_{E^{\star}}(r) = s_{E^{\star}}(r)$	4	9 sec

Table 1. Graph-classifiers with their input specifications, synthesized formulas, number of instances needed and time to do synthesis. A global integrity constraint is that no relation has self-loops.

Abbrev.	Meaning
self-loop-free N	$\forall u(\neg N(u,u))$
root r via N	$\forall u(N^\star(r,u))$
functional N	$\forall u,v,x\big(N(x,u) \wedge N(x,v) \to u = v\big)$
1:1 N	$\forall u,v,x\big(N(u,x) \wedge N(v,x) \to u = v\big)$

Table 2. Common Abbreviations. The first three are used as integrity constraints.

example, they can be used to dynamically check runtime assertions for programs that manipulate dynamicaly allocated data structures.

We ask the reader to familiarize herself with the seven example input specifications to our algorithm shown in Table 1. All these specifications are in first-order logic plus transitive closure (FO(TC)), a subset of SO.

5.1 Integrity Constraints and Abbreviations

In the specifications shown in Table 1, certain integrity constraints are assumed in addition to the written formulas. When we specify "*root r via N*" we assume that every vertex is reachable from the root by following edges labeled "N". When a relation, N, is specified as *functional*, we assume that it has outdegree at most one. Furthermore, unless otherwise stated, we make the default assumption that no relation has self-loops. In addition to the integrity constraints, we also use the abbreviaton "1:1 N to meant that N is one-to-one.

These abbreviations and their meanings are given in Table 2. Note that they may all be checked in linear time. We assume that the integrity constraints are part of the input specification and the synthesized specification.

Since the specifications in Tables 1 and 2 make use of transitive closure and doubly nested quantifiers, some of their naive implementations would run in cubic time. (The situation becomes even more interesting when in later sections we use second-order quantification because then the naive implementation would have exponential runtime.)

Table 1 also shows the linear-time specifications in L_1 that our tool automatically synthesized. We next define the language L_1. To help the reader get a feeling for L_1 we give an example first. The base formula $\forall v(\#s_N(v) \le 1)$ express the fact the the relation N is functional. The function symbol $s_N(v)$ denotes the set of vertices that are successors of v via an edge labeled N. Thus the formula says that for every vertex, v, the cardinality of that set is at most *one*. Note that the formula is checkable in linear time by doing a depth first search and recording the number of edges labeled N that leave each vertex.

5.2 Target Language L_1

Let σ be a vocabulary that includes the constant symbol, r, which is assumed to be present and a root throughout this section. The target language $L_1(\sigma)$ — or just L_1 when σ is understood — is based on single traversals of the graph, classifying the paths along the way using regular expressions. L_1 is a two-sorted first-order logic, with sorts nodes and sets of nodes. An important restriction of L_1 is that it only has one variable, v, ranging over nodes, and no variables ranging over sets of nodes.

As function symbols, L_1 has set constructors, i.e, maps from nodes to sets of nodes, based on regular expressions for classifying paths. For the vocabulary, σ, when the starting point of a path is a constant, we restrict our attention to the following simple regular expressions,

$$R(\sigma) = \{A^\star B^\star, A^\star B^\star C, A^\star B, A^\star B C^\star \mid A^2, B^2, C^2 \in \sigma\},$$

although we may abbreviate $A^\star A$ as A^+. When the starting point is the variable, v, so we may be computing n sets, we restrict the regular expressions to single letters: $S(\sigma) = \{A \mid A^2 \in \sigma\}$.

In order to define the function symbols of L_1 we need some notation involving regular expressions.

DEFINITION 5.1. *For a graph $G \in$ STRUC$[\sigma]$ and a regular expression $e \in R(\sigma)$, we write $x \xrightarrow{e} y$ to mean that there is a path from vertex x to vertex y such that the resulting sequence of edge labels is an element of $L(e)$.*

EXAMPLE 5.2. *Let $G = ([4], A^G, B^G, r^G)$ be a structure of vocabulary $\sigma = (A^2, B^2, r)$ such that $A^G = \{(0,1),(1,2),(3,0)\}$, $B^G = \{(0,3),(2,3)\}$, and $s^G = 0$ Then*

$$G \models 0 \xrightarrow{A^\star} 0 \ \wedge\ r \xrightarrow{A^\star} 2 \ \wedge\ \neg r \xrightarrow{A^\star} 3\ .$$

The functions of $L_1(\sigma)$ mapping nodes to sets of nodes are defined as follows,

DEFINITION 5.3. *For each regular expression $e \in R(\sigma)$ the function symbols s_e, p_e are available in $L_1(\sigma)$ and have meaning,*

$$s_e(i) \ = \ \{j \mid i \xrightarrow{e} j\} \qquad\qquad p_e(i) \ = \ \{j \mid j \xrightarrow{e} i\}$$

In words, $s_e(i)$ is the set of e-successors of i, i.e., those points reachable from i via a path of label $L(e)$; and $p_e(i)$ is the set of e-predecessors of i.

EXAMPLE 5.4. *Using the graph G from Example 5.2,*

- $s_{A^\star}(r) \ = \ \{0,1,2\}$
- $s_{A^+}(r) = \{1,2\}$
- $s_{A^\star B}(r) = \{3\}$
- $p_B(3) = \{0,2\}$

Finally, we define the set of base formulas $B_1(\sigma)$ to be the context-free language given by the grammar shown in Table 3. $L_1(\sigma)$ is the set of conjunctions of base formulas in $B_1(\sigma)$.

The symbol "d" is a boolean-valued constant that is true iff the graph satisfies the specified property. Thus instances

$\langle\text{base}\rangle$	$::=$	$\langle\text{clause}\rangle \to d \mid \langle\text{clause}\rangle \to \neg d \mid$ $\neg\langle\text{clause}\rangle \to d \mid \neg\langle\text{clause}\rangle \to \neg d$	
$\langle\text{clause}\rangle$	$::=$	$\langle\text{atom}\rangle \mid \forall v\ \langle\text{atom}\rangle \mid$ $\forall v\ (v \neq r \to \langle\text{atom}\rangle)$	
$\langle\text{atom}\rangle$	$::=$	$\langle\text{int}\rangle = \langle\text{const}\rangle \mid$ $\langle\text{int}\rangle \leq \langle\text{const}\rangle \mid \langle\text{set}\rangle = \langle\text{set}\rangle$	
$\langle\text{int}\rangle$	$::=$	$\langle\text{const}\rangle \mid \#\langle\text{set}\rangle$	
$\langle\text{const}\rangle$	$::=$	$0 \mid 1$	
$\langle\text{set}\rangle$	$::=$	$\{r\} \mid s_e(r) \mid p_e(r) \mid$	$e \in R(\sigma)$
		$s_\ell(v) \mid p_\ell(v)$	$\ell \in S(\sigma)$

Table 3. CFG for $B_1(\sigma)$, the base formulas of $L_1(\sigma)$.

satisfying the specification will have d true while instances not satisfying the specification will have d false. The term, "$\#A$" denotes the cardinality of set A.

THEOREM 5.5. *Every element of the language L_1 runs in expected linear time in the worst case.*

Proof: This follows from the fact that each set $s_e(r)$ or $p_e(r)$ can be computed in linear time via a depth first search starting at r. Furthermore, each set of sets $s_\ell(v)$ or $p_\ell(v)$, $v \in [n]$ has a total number of elements bounded by the number of edges in the input graph, and can be computed in linear time by examining each edge of label ℓ once. Finally, by maintaining the sets via hash tables, we can test equality of sets in expected linear time. □

5.3 Result Summary

The current implementation is Python-based and does not meet high standards for efficiency, but still, the running time is in the order of minutes rather than hours. The number of instances shows how many examples were needed before the correct formula was synthesized. We suspect that the reason so many examples were needed in the case of TREERP and TREEPP is that on small examples there were many alternate characterizations of the extra pointers, P, R.

We are confident that this algorithm will be similarly successful in quickly and automatically deriving linear-time tests for many other simple properties of graphs and data structures.

6. Finite Differencing

In his Ph.D. thesis, Bob Paige considered the common programming situation in which an expression,

$$C = f(x_1, \ldots, x_k) \tag{2}$$

is repeatedly evaluated in a block of code after some of the variables x_i may have been slightly modified [Pai81]. It may be the case that a slight change to the variable x_i, $x_i = e$, results in a slight change from C to C' so that it is much easier to compute the change incrementally than to recompute $C' = f(x_1, \ldots, x_i', \ldots, x_k)$ from scratch. If so, Paige says informally that C is "continuous" with respect to this change, and he calls the code that incrementally computes C' the "formal derivative" of C with respect to the change.

Paige calls the part of the code that goes before the change $x_i = e$ the pre-derivative ($\partial^-(C, x_i = e)$) and the part of the code that goes after the change he calls the post-derivative ($\partial^+(C, x_i = e)$). However, for simplicity we will assume that we have access to the before and after values, x_i and x_i'. Thus we will refer to the code to compute C' and thus reestablish the invariant Eqn 2 as the *derivative* of C w.r.t. the change $x_i = e$, ($\partial(C, x_i = e)$).

See Table 4 for fourteen examples of formal derivatives, the code we automatically synthesized to evaluate them, and the required time. In each case the code — which is constant time and thus asymptotically optimal — is equivalent to the derivatives that Paige computed by hand and listed in tables in his thesis.

As in §5, our algorithm follows the methodology described in §3 and §4. The program was written in Python making use of the built in implementation for sets.

6.1 Target Language, L_2

For each input vocabulary, σ, the target language, $L_2(\sigma)$ is similar to $L_1(\sigma)$ but somewhat simpler. We again have a single domain variable, v. For each relation symbol R (or function symbol f) from σ that may change, the vocabulary, σ' of $L_2(\sigma)$ contains both R and R' (or f and f') denoting the value before and after the change. When the target formula is the relation C, we model this in L_2 as the boolean function c. When the target formula is an integer, or boolean variable, c, we model it as a domain valued, or boolean valued function also called c.

DEFINITION 6.1. *The base formulas $B_2(\sigma)$ consist of all universally quantified literals or implications of literals from σ', $B_2(\sigma) = \{\forall v(\ell_1), \forall v(\ell_1 \to \ell_2) \mid \ell_1, \ell_2 \text{ literals from } \sigma'\}$. L_2 consists of arbitrary conjunctions of base formulas from $B_2(\sigma)$.*

6.2 Results

We are pleased that for many examples we can automatically generate formal derivatives that run in constant time and are thus asymptotically optimal. This is a validation of our methodology and will be useful in future work on synthesis. As we sketch in the next section, the automatic derivation of asymptotically optimal formal derivatives can be a useful building block for the automatic synthesis of efficient data structures and algorithms for a richer class of algorithmic specifications.

7. Extensions

We are taking steps towards fulfilling Paige and Schwartz's dream of writing the specification φ in a very high level lan-

Expression	Change	Synthesized Derivative	Code	Time
$C = T + S$	$T \mathrel{+}= \{a\}$	$v = a \rightarrow c'(v) = 1$ $v \neq a \rightarrow c'(v) = c(v)$	$C \mathrel{+}= \{a\}$	121.88 sec
$C = T + S$	$S \mathrel{+}= \{a\}$	$v = a \rightarrow c'(v) = 1$ $v \neq a \rightarrow c'(v) = c(v)$	$C \mathrel{+}= \{a\}$	121.94 sec
$C = T + S$	$T \mathrel{-}= \{a\}$	$v \neq a \rightarrow c'(v) = c(v)$ $\neg T(a) \rightarrow c'(a) = 0$ $T(a) \rightarrow c'(a) = c(a)$	$\texttt{if}\ a \notin S : C \mathrel{-}= \{a\}$	87.95 sec
$C = T + S$	$S \mathrel{-}= \{a\}$	$v \neq a \rightarrow c'(v) = c(v)$ $\neg S(a) \rightarrow c'(a) = 0$ $S(a) \rightarrow c'(a) = c(a)$	$\texttt{if}\ a \notin T : C \mathrel{-}= \{a\}$	96.5 sec
$C = T - S$	$T \mathrel{+}= \{a\}$	$v \neq a \rightarrow c'(v) = c(v)$ $\neg S(a) \rightarrow c'(a) = 1$ $S(a) \rightarrow c'(a) = 0$	$\texttt{if}\ a \notin S : C \mathrel{+}= \{a\}$	69.85 sec
$C = T - S$	$T \mathrel{-}= \{a\}$	$c(v) = 0 \rightarrow c'(v) = 0$ $v \neq a \rightarrow c'(v) = c(v)$ $v = a \rightarrow c'(a) = 0$	$C \mathrel{-}= \{a\}$	115.57 sec
$C = T - S$	$S \mathrel{+}= \{a\}$	$c(v) = 0 \rightarrow c'(v) = 0$ $v \neq a \rightarrow c'(v) = c(v)$ $v = a \rightarrow c'(a) = 0$	$C \mathrel{-}= \{a\}$	117.0 sec
$C = T - S$	$S \mathrel{-}= \{a\}$	$v \neq a \rightarrow c'(v) = c(v)$ $\neg T(a) \rightarrow c'(a) = 0$ $T(a) \rightarrow c'(a) = 1$	$\texttt{if}\ a \in T : C \mathrel{+}= \{a\}$	115.57 sec
$C = f(S)$	$S \mathrel{+}= \{a\}$	$v \neq f(a) \rightarrow c'(v) = c(v)$ $v = a \rightarrow c'(f(a)) = 1$	$C \mathrel{+}= \{f(a)\}$	100.91 sec
$C = f^{-1}(S)$	$f(a) = b$	$v \neq a \rightarrow c'(v) = c(v)$ $S(b) \rightarrow c'(a) = 1$ $\neg S(b) \rightarrow c'(a) = 0$	$\texttt{if}\ b \notin S : C \mathrel{-}= \{a\}$ $\texttt{else} : C \mathrel{+}= \{a\}$	71.35 sec
$c_S = \#S$	$S \mathrel{+}= \{a\}$	$S(a) \rightarrow c'_S = c_S$ $\neg S(a) \rightarrow c_S + 1 = c'_S$	$\texttt{if}\ a \notin S : c_S \mathrel{+}= 1$	70.28 sec
$c_S = \#S$	$S \mathrel{-}= \{a\}$	$\neg S(a) \rightarrow c'_S = c_S$ $S(a) \rightarrow c'_S + 1 = c_S$	$\texttt{if}\ a \in S : c_S \mathrel{-}= 1$	61.29 sec
$c = (\#S == 0)$	$S \mathrel{+}= \{a\}$	$v = a \rightarrow c' = 0$	$c = \texttt{false}$	4.46 sec
$c = (\#S == 0)$	$S \mathrel{-}= \{a\}$	$c_S \neq 1 \rightarrow c' = c$ $c'_S = c_S \rightarrow c = c'$ $c'_S = 0 \rightarrow c' = 1$	$\texttt{if}\ a \in S : c_S \mathrel{-}= 1$ $c = (c_S == 0)$	7.59 sec

Table 4. Finite differencing problems with their automatically synthesized, asymptotically optimal formal derivatives in L_2, assumed to be universally quantified conjuncts; When the expression is a set C, c and c' are the characteristic functions of C before and after the change, respectively.

guage and having a system to automatically choose good data structures and algorithms that implement φ. This section describes two examples that may give a sense of how we are considering using our methodology to attack more complicated problems.

7.1 Bipartite Graphs

The following SO$\exists$ sentence expresses the bipartite property for undirected graphs.

$$\Phi_{\text{bp}} \quad = \quad \exists S^1 \forall xy (E(x,y) \rightarrow (S(x) \leftrightarrow \neg S(y))) \ .$$

Φ_{bp} suggests the abstract algorithm for testing bipartiteness shown in Fig. 5. This algorithm starts with the empty graph, for which $S = \emptyset$ is a witness for bipartiteness. Then while not all edges have been added, we add a new edge and try to update S so that it is still a witness for bipartiteness. If that is not possible we answer that the graph is not bipartite. If we succeed in adding all the edges without failure then we answer that the graph is bipartite.

It is trivial to automatically generate the algorithm shown in Fig. 5, including the "bipartiteness invariant" β which is just the first-order part of Φ_{bp},

```
0. On input $G = ([n], E)$,
1. $H = ([n], F, S) = ([n], \emptyset, \emptyset)$.
2. while $(F \neq E)$
3.     $F += e = \ni(E - F)$
4.     if $(\neg$ updatePreserving$_\beta(S, e))$:
           return(not bipartite)
5. return(bipartite)
```

Figure 5. An abstract bipartiteness algorithm. Line 3 chooses an element e from $E - F$ and adds it to F.

$$\beta \quad \equiv \quad \forall xy(E(x,y) \to (S(x) \leftrightarrow \neg S(y))) \ .$$

In the base case, $H = ([n], F, S) = ([n], \emptyset, \emptyset)$, we have that $H \models \beta$, i.e., $S = \emptyset$ is a witness for the bipartiteness of the empty graph. At line 3 we inductively add a new edge. The operator $\ni$ chooses some element $e \in (E - F)$ to add to F. This is one place that inductive synthesis will apply: we want to learn a good order in which to add the edges of the graph, i.e., one that makes it easy to update S in a way that preserves the invariant β. The second place we use inductive synthesis is in choosing the updatePreserving$_\beta$ method.

We now point out informally that a good choice of either the order in which to add edges, or the updatePreserving$_\beta$ method will give us an optimal algorithm for testing bipartitenss. This is related to the concept of angelic nondeterminism in which a good choice — also achieved by a form of inductive synthesis — will give us a good algorithm [BCG$^+$10].

Ordering: If the angels tell us to insert the edges via a depth first search traversal of the graph, then an obvious updatePreserving$_\beta$ method works correctly in constant time per edge, resulting in a linear and thus optimal bipartiteness algorithm. The method upon adding edge $e = (i,j)$ is: if $S(j) = S(i)$, then fail, else if $S(j)$ is undefined then set it to $1 - S(i)$.

updatePreserving$_\beta$ method: Even if we have no control over the order that the edges are inserted, there is a natural choice of the updatePreserving$_\beta$ method that leads to an essentially optimal bipartiteness algorithm.

Inductively assume that $H = ([n], F, S)$ satisfies β, and that edge $e = (i,j)$ is added to F. The first question is whether now that H has been updated it still satisfies β with S unchanged. This holds iff $(H, i/x, j/y) \models (S(x) \leftrightarrow \neg S(y))$. If so, then we can let leave S unchanged and β is maintained.

Otherwise, we have that $(H, i/x, j/y) \models (S(x) \leftrightarrow S(y))$. In this case, in order to make β true again, we have to change the value of one of $S(i), S(j)$.

The situation is symmetrical so WLOG assume that we have to change the value of $S(i)$. In order to preserve β we will then have to change all neighbors of i and thus all neighbors of neighbors of i, and so on. A naive incremental algorithm is thus suggested which changes the value of $S(k)$ for all k in the connected component of i with respect the the edge set $F - \{e\}$. Furthermore, if j is in this connected component then it is impossible to make β true for any choice of S and we must report failure. This naive algorithm takes time $O(nm)$ where $m = |E|$ is the number of edges in the whole graph.

In order to synthesize a more efficient algorithm we look more closely at the change that must be made: We have to change the value of S on every element of i's connected component. Let $S(i)$ be the part of i's connected component that is in S and $\overline{S}(i)$ be the part of i's connected component not in S. Then we want to flip $S(i)$ and $\overline{S}(i)$ and merge them with the corresponding parts of j's connected component, i.e.,

```
1. if $(S(i) = S(j))$: return(not bipartite)
2. $S(j) = S(j) \cup \overline{S}(i)$;    $\overline{S}(j) = \overline{S}(j) \cup S(i)$
```

With the sets $S, \overline{S}$ instantiated using Union/Find, the complexity of this incremental algorithm is then essentially linear [AH74].

Finally, since we have changed the data structure for storing S we must revisit the case where $S(i) \leftrightarrow \neg S(j)$. The invariant β is not violated by adding the edge (i,j) but we must still update the data structure, i.e., execute the following command:

$S(j) = S(j) \cup S(i)$; $\overline{S}(j) = \overline{S}(j) \cup \overline{S}(i)$.

Note that in this second case since we constructed an algorithm that works for any insertion order, we have given an essentially optimal incremental algorithm for bipartiteness.

7.2 Topological Sort

The Topological Sort problem asks us to compute an ordering of a directed graph that honors all edge constraints. To express this in SO$\exists$ note that a new ordering on the set $[n]$ of vertices can be specified as an injective (and thus bijective) function $g : [n] \to [n]$. Let Inj(g) mean that g is an injective function,

$$\text{Inj}(g) \quad \equiv \quad \forall xy\, (g(x) = g(y) \ \to \ x = y) \ .$$

Using this notation, a natural formulation of topological sort is,

$$\Phi_{\text{ts}} \quad \equiv \quad \exists g^1 \big(\text{Inj}(g) \ \wedge \ \forall x\, y\, (E(x,y) \to g(x) < g(y))\big) \ .$$

Again we can read the invariant we must maintain directly from the specification,

$$\tau \quad \equiv \quad \text{Inj}(g) \ \wedge \ \forall x\, y\, (E(x,y) \to g(x) < g(y)) \ .$$

```
0. On input $G = ([n], E)$,
1. $H = ([n], E, g, S) = ([n], E, \emptyset, \emptyset)$.
2. while $(S \neq [n])$
3.     $S+ = i \ = \ \ni ([n] - S)$
4.     if $(\neg$ updatePreserving$_\tau(g, i))$: return(cyclic)
5. return(acyclic, $g$)
```

Figure 6. An abstract topological sort algorithm. S is the initially empty domain of the function g.

Fig. 6 shows an abstract topological sort algorithm which could again be derived easily from the specification. In this case we incrementally add the vertices to the domain S of the function g. As for the abstract bipartite algorithm, we can synthesize an optimal algorithm if an angel tells us which order to insert vertices into S.

best ordering: It is well known that if we do a depth first search of G and then insert the vertices into S in order of reverse finish time, then that is a topological sort, so with a trivial updatePreserving$_\tau$ method we get an optimal, linear-time algorithm.

more interesting ordering: It is also well known that if we choose vertices to add to S that have no incoming edges from $[n] - S$, then we get a topological sort. If the angel suggests this ordering, then we are very happy, because that lets us automatically use the finite differencing algorithms that we derived in Section 6. In this case we can derive code to automatically maintain the indegree of each vertex in $[n] - S$, and to maintain a set of those vertices that have indegree zero. Thus the kth iteration of the updatePreserving$_\tau(g, i)$ method simply lets $g(k) = i$. If before the end of the while loop there are no vertices of indegree 0, then we report failure, i.e., G was cyclic. Fig. 7 shows a modified topological sort algorithm that includes the hint from the angel: "add vertices of indegree 0". From here we can automatically derive the constant-time finite differencing code to maintain the definitions of $f(v)$ as the indegree of v and Z as the set of vertices of indegree 0, and thus synthesize an optimal topological sort algorithm.

```
0. On input $G = ([n], E)$,
1. $H = ([n], E, g, S) = ([n], E, \emptyset, \emptyset)$.
2. $f(v) = \#\{u \in [n] - S \mid (u, v) \in E\}$   for $v \in [n]$
3. $Z = \{u \in [n] - S \mid f(u) = 0\}$
4. while $(S \neq [n])$
5.     if $(Z = \emptyset)$: return(cyclic)
6.     $S+ = i \ = \ \ni Z$
7.     updatePreserving$_{\tau'}(g, i)$
8. return(acyclic, $g$)
```

Figure 7. Revision of the abstract topological sort algorithm from Fig. 6 with anglelic hint; $\tau' \equiv \tau \wedge (2) \wedge (3)$.

8. Related Work

Counterexample Guided Inductive Synthesis Inductive synthesis refers to the process of generating a system from input-output examples. This process involves using each new input-output example to refine the hypothesis about what the correct system should be until convergence is reached. Inductive synthesis had its origin in the pioneering work by Gold on language learning [Gol67] and by Shapiro on algorithmic debugging and its application to automated program construction [Sha83]. The inductive approach [Mug92, FP94] for synthesizing a program involves *debugging* the program with respect to positive and negative examples until the correct program is synthesized. The negative examples can be counterexamples discovered while trying to prove a program's correctness. Counterexamples have been used in incremental synthesis of programs [SLTB$^+$06] and discrete event systems [BMM04]. The technique presented in this paper is also a form of inductive synthesis, but applied in a novel program synthesis setting.

SAT/SMT Based Program Synthesis Recent advances in SAT/SMT technologies have revived interest in program synthesis techniques. Recent work shows how to user SMT constraint based program verification techniques for synthesizing programs from first order logic specifications [SGF10]. In contrast, we allow for more expressive second order logic constraints and use a very different methodology based on inductive synthesis. Sketching[SLTB$^+$06] uses SAT solvers to implement the inductive program synthesis technique, but requires the programmer to write a partial program whose holes are limited to taking values over finite domains. Though our inductive synthesis based technique also uses SAT solvers, it searches for full programs (as opposed to values for holes) over the space of their representations as logical formulas (as opposed to small finite set of values for holes).

Finite Differencing This paper was inspired in part by Bob Paige's work on transformational program. He used finite differencing to try to automatically derive efficient data structures and algorithms for high level specifications in SETL [Pai81, CP87]. Annie Liu is doing some notable work pushing this methodology forward [LT95, LS09].

9. Conclusion

We have shown that a simple, general inductive synthesis algorithm can automatically translate high-level algorithmic specifications into asymptotically optimal code. There is much further work to do. In particular,

- This methodology as presented is widely applicable and should be used and tested in many settings to see how far it can go in this simple form.
- Once these limits are better understood, there is room to test richer learning algorithms on richer target languages.
- Most exciting to us is the idea very slightly hinted at in §7 that building blocks such as the automatic finite differencing can let us take second order existential specifications and derive efficient algorithms to maintain their implied invariants as we start with the empty graph and add edges incrementally until we have the entire input graph. This might also be an approach for synthesizing incremental algorithms.

A. Propositional Encoding of First and Second-Order Formulas

In this appendix, we explain how to use a SAT solver to test whether there exists a model of a given size for a second-order exsitential formula, Ψ. Our construction is immediate from the proof of Cook's Theorem from Fagin's Theorem in [Imm99, Thms 7.8, 7.16].

PROPOSITION A.1. *Given a second-order existenital formula Ψ and a positive integer n, we can construct a boolean formula φ such that $\varphi \in \mathrm{SAT}$ iff Φ has a model of size n.*

Proof: Let's take as an example $\Phi \equiv \exists R^1 \psi$ with first-order part $\psi \equiv \forall x \exists y (E(x,y) \wedge R(y))$. We want to know if there exists a structure $\mathcal{A} = ([n], E^{\mathcal{A}})$ that satisfies Ψ. That is the same question as whether there exists a structure $\mathcal{B} = ([n], E^{\mathcal{B}}, R^{\mathcal{B}})$ that satisfies ψ.

To guess such a structure, $\mathcal{B}$, we must guess the answers to $n^2 + n$ binary questions, i.e, whether $E(0,0)$ holds, whether $E(1,0)$ holds, …, whether $E(n-1,n-1)$ holds, whether $R(0)$, …, whether $R(n-1)$ holds. Thus in the boolean formula φ that we construct there will be n^2+n boolean variables: $E(0,0), E(1,0), \ldots, E(n-1,n-1)$; $R(0), \ldots, R(n-1)$. The boolean formula φ will assert that the chosen structure $\mathcal{B}$ satisfies ψ:

$$\varphi \equiv \bigwedge_{i=0}^{n-1} \bigvee_{j=0}^{n-1} (E(i,j) \wedge R(j)) .$$

We believe that the reader will be able to synthesize the general algorithm from this one example[1]. Note that the formula φ can be constructed in time linear in its size. The size of φ is $O(n^{\max(a,r)}|\psi|)$ where a is the maximum arity of the relations existentialy quantified, and r is the depth of nesting of first-order quantifiers in ψ. □

[1] One of us has been accused of being too terse in his writing. For him, the elegance and simplicity of Proposition A.1 is embedded in the simple example given. Other coauthors said, "What about the cases when a function or a numeric relation occurs in ψ?" Well, each value of a function is determined by $\log n$ bits, so we need $n \log n$ boolean variables to encode a unary function. For numeric relations, if $x < y$ occurs in ψ, then in φ it would be replaced by $i < j$ for each fixed value of i and j. For example, $3 < 7$ would then be replaced by true and $5 < 2$ would be replaced by false.

References

[AH74] Alfred V. Aho and John E. Hopcroft. *The Design and Analysis of Computer Algorithms*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1974.

[BCG$^+$10] Rastislav Bodik, Satish Chandra, Joel Galenson, Doug Kimelman, Nicholas Tung, Shaon Barman, and Casey Rodarmor. Programming with angelic nondeterminism. *SIGPLAN Not.*, 45(1):339–352, 2010.

[BMM04] B.A. Brandin, R. Malik, and P. Malik. Incremental verification and synthesis of discrete-event systems guided by counter examples. *Control Systems Technology*, 12(3), May 2004.

[CP87] Jiazhen Cai and Robert Paige. Binding performance at language design time. In *POPL*, pages 85–97, 1987.

[dMB] Leonardo de Moura and Nikolaj Bjørner. Z3 an efficient smt solver. `http://research.microsoft.com/en-us/um/redmond/projects/z3/`.

[FP94] Pierre Flener and Lubos Popelmnsky. On the use of inductive reasoning in program synthesis: Prejudice and prospects. In *LOBSTR'94*. 1994.

[Gol67] E. Mark Gold. Language identification in the limit. *Information and Control*, 10(5):447–474, 1967.

[Imm99] Neil Immerman. *Descriptive Complexity*. Springer, New York, 1999.

[LS09] Yanhong A. Liu and Scott D. Stoller. From datalog rules to efficient programs with time and space guarantees. *ACM Trans. Program. Lang. Syst.*, 31(6), 2009.

[LT95] Yanhong A. Liu and Tim Teitelbaum. Systematic derivation of incremental programs. *Sci. Comput. Program.*, 24(1):1–39, 1995.

[Mug92] Stephen Muggleton, editor. *Inductive Logic Programming*, volume 38 of *The APIC Series*. Academic Press, 1992.

[Pai81] Robert Paige. *Formal Differentiation - A Program Synthesis Technique*. UMI Press, 1981.

[SGF10] Saurabh Srivastava, Sumit Gulwani, and Jeff Foster. From program verification to program synthesis. In *POPL*, pages 313–326, 2010.

[Sha83] Ehud Y. Shapiro. *Algorithmic Program DeBugging*. MIT Press, Cambridge, MA, USA, 1983.

[SLTB$^+$06] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodík, Sanjit Seshia, and Vijay Saraswat. Combinatorial sketching for finite programs. In *ASPLOS*, 2006.