



# Synthesis of Semantic Actions in Attribute Grammars

Pankaj Kumar Kalita   
 Computer Science and Engineering  
 Indian Institute of Technology Kanpur  
 Kanpur, India  
 pkalita@cse.iitk.ac.in

Miriyala Jeevan Kumar  
 Fortanix Tech. India Pvt. Ltd.  
 Bengaluru, India  
 gl.miriyala@gmail.com

Subhajit Roy   
 Computer Science and Engineering  
 Indian Institute of Technology Kanpur  
 Kanpur, India  
 subhajit@iitk.ac.in

**Abstract**—Attribute grammars allow the association of semantic actions to the production rules in context-free grammars, providing a simple yet effective formalism to define the semantics of a language. However, drafting the semantic actions can be tricky and a large drain on developer time. In this work, we propose a synthesis methodology to automatically infer the semantic actions from a set of examples associating strings to their meanings. We also propose a new coverage metric, *derivation coverage*. We use it to build a sampler to effectively and automatically draw strings to drive the synthesis engine. We build our ideas into our tool, PĀṆINI, and empirically evaluate it on twelve benchmarks, including a forward differentiation engine, an interpreter over a subset of Java bytecode, and a mini-compiler for C language to two-address code. Our results show that PĀṆINI scales well with the number of actions to be synthesized and the size of the context-free grammar, significantly outperforming simple baselines.

**Index Terms**—Program synthesis, Attribute grammar, Semantic actions, Syntax directed definition

## I. INTRODUCTION

Attribute grammars [1] provide an effective formalism to supplement a language syntax (in the form of a context-free grammar) with semantic information. The semantics of the language is described using *semantic actions* associated with the grammar productions. The semantic actions are defined in terms of *semantic attributes* associated with the non-terminal symbols in the grammar.

Almost no modern applications use hand-written parsers anymore; instead, most language interpretation engines today use automatic parser generators (like YACC [2], BISON [3], ANTLR [4] etc.). These parser generators employ the simple, yet powerful formalism of attribute grammars to couple parsing with semantic analysis to build an efficient frontend for language understanding. This mechanism drives many applications like model checkers (eg. SPIN [5]), automatic theorem provers (eg. Q3B [6], CVC5 [7]), compilers (eg. CIL [8]), database engines (eg. MYSQL [9]) etc.

However, defining appropriate semantic actions is often not easy: they are tricky to express in terms of the inherited and synthesized attributes over the grammar symbols in the respective productions. Drafting these actions for large grammars requires a significant investment of developer time.

In this work, we propose an algorithm to automatically synthesize semantic actions from *sketches* of attribute grammars.

$S \mapsto E$ <sup>[1]</sup> $E \mapsto E + F$ <sup>[2]</sup>   $E - F$ <sup>[3]</sup>   $F$ <sup>[4]</sup> $F \mapsto F * K$ <sup>[5]</sup>   $K$ <sup>[6]</sup> $K \mapsto K \wedge \text{num}$ <sup>[7]</sup>   $\text{SIN}(K)$ <sup>[8]</sup>   $\text{COS}(K)$ <sup>[9]</sup>   $\text{num}$ <sup>[10]</sup>   $\text{var}$ <sup>[11]</sup>	$\text{output}(E.\text{val});$ $E.\text{val} \leftarrow h_1^*(E.\text{val}, F.\text{val});$ $E.\text{val} \leftarrow h_2^*(E.\text{val}, F.\text{val});$ $E.\text{val} \leftarrow F.\text{val};$ $F.\text{val} \leftarrow h_3^*(F.\text{val}, K.\text{val});$ $F.\text{val} \leftarrow K.\text{val};$ $K.\text{val} \leftarrow h_4^*(K.\text{val}, \text{num});$ $K.\text{val} \leftarrow h_5^*(K.\text{val});$ $K.\text{val} \leftarrow h_6^*(K.\text{val});$ $K.\text{val} \leftarrow \text{getVal}(\text{num}) + 0\varepsilon;$ $K.\text{val} \leftarrow \text{lookUp}(\Omega, \text{var}) + 1\varepsilon;$
--	--

Fig. 1: Attribute grammar for automatic forward differentiation ( $\Omega$  is the symbol table)

Fig. 1 shows a sketch of an attribute grammar for automatic forward differentiation using dual numbers (we explain the notion of dual numbers and the example in detail in §III-A). The production rules are shown in green color while the semantic actions are shown in the blue color. Our synthesizer attempts to infer the definitions of the holes in this sketch (the function calls  $h_1^*$ ,  $h_2^*$ ,  $h_3^*$ ,  $h_4^*$ ,  $h_5^*$ ,  $h_6^*$ ); we show these holes in yellow background. As an attribute grammar attempts to assign “meanings” to language strings, the meaning of a string in this language is captured by the output construct.

This is a novel synthesis task: the current program synthesis tools synthesize a program such that a desired specification is met. In our present problem, we attempt to synthesize semantic actions within an attribute grammar: the synthesizer is required to *infer* definitions of the holes such that for *all* strings in the language described by the grammar, the computed semantic value (captured by the output construct) matches the intended semantics of the respective string—this is a new problem that cannot be trivially mapped to a program synthesis task.

Our core observation to solve this problem is the follows: for any string in the language, the sequence of semantic actions executed for the syntax-directed evaluation of any string is a loop-free program. This observation allows us to reduce attribute grammar synthesis to a *set program synthesis tasks*. Unlike a regular program synthesis task where we are interested in synthesizing a single program, the above reduction requires us to solve a set of *dependent* program

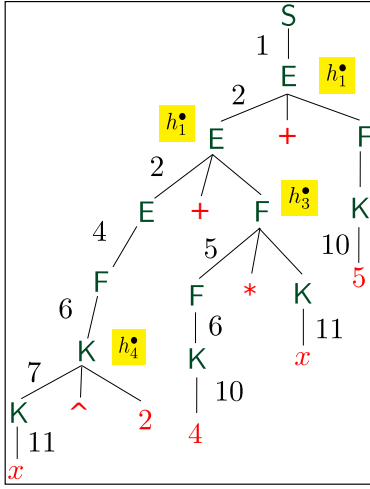


Fig. 2: Parse tree for input  $x^2+4*x+5$

synthesis instances simultaneously. These program synthesis tasks are dependent as they contain common components (to be synthesized) shared by multiple programs, and hence, they cannot be solved in isolation—as the synthesis solution from one instance can influence others.

Given a set of examples  $E$  (strings that can be derived in the provided grammar) and their expected semantics  $O$  (semantic values in `output`); for each example  $e_i \in E$ , we apply the reduction by sequencing the set of semantic actions for the productions that occur in the derivation of  $e_i$ . This sequence of actions forms a loop-free program  $P_i$ , with the expected semantic output  $o_i \in O$  as the specification. We collect such programs-specification pairs,  $\langle P_i, o_i \rangle$ , to create a set of dependent synthesis tasks. An attempt at *simultaneous synthesis* of all these set of tasks by simply conjoining the synthesis constraints does not scale.

Our algorithm adopts an incremental, counterexample guided inductive synthesis (CEGIS) strategy, that attempts to handle only a “small” set of programs simultaneously—those that violate the current set of examples. Starting with only a single example, the set of satisfied examples are expanded incrementally till the specifications are satisfied over all the programs in the set.

Furthermore, to relieve the developer from providing examples, we also propose an example generation strategy for attribute-grammars based on a new coverage metric. Our coverage metric, *derivation coverage*, attempts to capture distinct behaviors due to the presence or absence of each of the semantic actions corresponding to the syntax-directed evaluation of different strings.

We build an implementation, PĀṆINI<sup>1</sup>, that is capable of automatically synthesizing semantic actions (across both synthesized and inherited attributes) in attribute grammars. For the attribute grammar sketch in Fig. 1, PĀṆINI automatically synthesizes the definitions of holes as shown in Fig. 3 in a mere 39.2 seconds. We evaluate our algorithm on a set of attribute grammars, including a Java bytecode interpreter and

<sup>1</sup>PĀṆINI (पाणिनि) was a Sanskrit grammarian and scholar in ancient India.

a mini-compiler frontend. Our synthesizer takes a few seconds on these examples.

To the best of our knowledge, ours is the first work on automatic synthesis of semantic actions on attribute grammars. The following are our contributions in this work:

- We propose a new algorithm for synthesizing semantic actions in attribute grammars;
- We define a new coverage metric, *derivation coverage*, to generate effective examples for this synthesis task;
- We build our algorithms into an implementation, PĀṆINI, to synthesize semantic actions for attribute grammars;
- We evaluate PĀṆINI on a set of attribute grammars to demonstrate the efficacy of our algorithm. We also undertake a case-study on the attribute grammar of the parser of the SPIN model-checker to automatically infer the constant-folding optimization and abstract syntax tree construction.

An extended version of this article is available [10]. The implementation and benchmarks of PĀṆINI are available at <https://github.com/pkalita595/Panini>.

## II. PRELIMINARIES

Attribute grammars [1] provide a formal mechanism to capture language semantics by extending a context-free grammar with *attributes*. An attribute grammar  $\mathcal{G}$  is specified by  $\langle S, P, T, N, F, \Gamma \rangle$ , where

- $T$  and  $N$  are the set of *terminal* and *non-terminal* symbols (resp.), and  $S \in N$  is the *start symbol*;
- A set of (context-free) productions,  $p_i \in P$ , where  $p_i : X_i \mapsto Y_{i1}Y_{i2} \dots Y_{in}$ ; a production consists of a *head*  $X_i \in N$  and *body*  $Y_{i1} \dots Y_{in}$ , such that each  $Y_{ik} \in T \cup N$ .
- A set of *semantic actions*  $f_i \in F$ ;
- $\Gamma : P \rightarrow F$  is a map from the set of productions  $P$  to the set of *semantic actions*  $f_i \in F$ .

The set of productions in  $\mathcal{G}$  describes a *language* (denoted as  $\mathcal{L}(\mathcal{G})$ ) to capture the set of *strings* that can be *derived* from  $S$ . A *derivation* is a sequence of applications of productions  $p_i \in P$  that transforms  $S$  to a string,  $w \in \mathcal{L}(\mathcal{G})$ ; unless specified, we will refer to the *leftmost* derivation where we always select the leftmost non-terminal for expansion in a sentential form. As we are only concerned with parseable grammars, we constrain our discussion in this paper to *unambiguous* grammars.

The semantic actions associated with the grammar productions are defined in terms of *semantic attributes* attached to the non-terminal symbols in the grammar. Attributes can be *synthesized* or *inherited*: while a synthesized attributes are computed from the children of a node in a parse tree, an inherited attribute is defined by the attributes of the parents or siblings.

Fig. 1 shows an attribute grammar with *context-free productions* and the associated *semantic actions*. Fig. 2 shows the parse tree of the string  $x^2 + 4x + 5$  on the provided grammar; each internal node of the parse tree have associated semantic actions (we have only shown the “unknown” actions that need to be inferred).

$$\begin{array}{lll}
h_1^\bullet (a_1 + a_2\varepsilon, b_1 + b_2\varepsilon): & h_2^\bullet (a_1 + a_2\varepsilon, b_1 + b_2\varepsilon): & h_3^\bullet (a_1 + a_2\varepsilon, b_1 + b_2\varepsilon): \\
r \leftarrow a_1 + b_1 & r \leftarrow a_1 - b_1 & r \leftarrow a_1 * b_1 \\
d \leftarrow a_2 + b_2 & d \leftarrow a_2 - b_2 & d \leftarrow a_2 * b_1 + a_1 * b_2 \\
\mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon \\
\text{(a)} & \text{(b)} & \text{(c)} \\
h_5^\bullet (a_1 + a_2\varepsilon): & h_6^\bullet (a_1 + a_2\varepsilon): & h_4^\bullet (a_1 + a_2\varepsilon, c): \\
r \leftarrow \sin(a_1) & r \leftarrow \cos(a_1) & r \leftarrow \text{pow}(a_1, c) \\
d \leftarrow a_2 * \cos(a_1) & d \leftarrow a_2 * \sin(a_1) * -1 & d \leftarrow a_2 * \text{pow}(a_1, c - 1) \\
\mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon & \mathbf{return} \ r + d\varepsilon \\
\text{(d)} & \text{(e)} & \text{(f)}
\end{array}$$

Fig. 3: Synthesized holes for holes in Fig. 1

*Parser generators* [2] accept an attribute grammar and automatically generate parsers that perform a *syntax-directed evaluation* of the semantic actions. For ease of discussion, we assume that the semantic actions are *pure* (i.e. do not cause side-effects like printing values or modifying global variables) and generate a deterministic *output value* as a consequence of applying the actions.

An attribute grammar is *non-circular* if the dependencies between the attributes in every syntax tree are acyclic. Non-circularity is a sufficient condition that all strings have unique evaluations [11].

**Notations.** We notate production symbols by serif fonts, non-terminal symbols (or placeholders) by capital letters (eg.  $X$ ) and terminal symbols by small letters (eg.  $a$ ). Sets are denoted in capital letters. We use arrows with tails ( $\rightarrow$ ) in productions and string derivations to distinguish it from function maps. We use the notation  $e[g_1/g_2]$  to imply that all instances of the subexpression  $g_2$  are to be substituted by  $g_1$  within the expression  $e$ . We use the notation of Hoare logic [12] to capture program semantics:  $\{P\}Q\{R\}$  implies that if the program  $Q$  is executed with a *precondition*  $P$ , it can only produce an output state in  $R$ ;  $P$  and  $R$  are expressed in some base logic (like first-order logic).

### III. OVERVIEW

**Sketch of an attribute grammar.** We allow the *sketch*  $\mathcal{G}^\bullet$  of an attribute grammar (as syntax directed definition (SDD)),  $\mathcal{G}^\bullet = \langle S, P, T, N, H^\bullet, \Gamma \rangle$ , to contain *holes* for unspecified functionality within the semantic actions  $h_i^\bullet \in H^\bullet$ . For example, in Fig. 1, the set of holes comprises of the functions  $H = \{h_1^\bullet, h_2^\bullet, h_3^\bullet, h_4^\bullet, h_5^\bullet, h_6^\bullet\}$ . If the semantic action corresponding to a production  $p$  contains hole(s), we refer to the production  $p$  as a *sketchy* production; when the definitions for all the holes in a sketchy production are resolved, we say that the production is *ready*. The *completion* (denoted  $\mathcal{G}^{\{f_1, \dots, f_n\}}$ ) of a grammar sketch  $\mathcal{G}^\bullet$  denotes the attribute grammar where a set of functions  $f_1, \dots, f_n$  replace the holes  $h_1^\bullet, \dots, h_n^\bullet$ . We denote the syntax-directed evaluation of a string  $w$  on an attribute grammar  $\mathcal{G}$  as  $\llbracket w \rrbracket^{\mathcal{G}}$ ; we consider that any such evaluation results in a *value* (or  $\perp$  if  $w \notin \mathcal{G}$ ).

**Example Suite.** An *example* (or *test*) for an attribute grammar  $\mathcal{G}$  can be captured by a tuple  $\langle w, v \rangle$  such that  $w \in \mathcal{L}(\mathcal{G})$  and

$\llbracket w \rrbracket^{\mathcal{G}} = v$ . A set of such examples constitutes an *example suite* (or *test suite*).

If the language described by the grammar  $\mathcal{G}$  supports *variables*, then any evaluation of  $\mathcal{G}$  needs a *context*,  $\beta$ , that binds the free variables to *input values*. We denote such examples as  $\llbracket w \rrbracket_\beta^{\mathcal{G}} = v$ . When the grammar used is clear from the context, we drop the superscript and simplify the notation to  $\llbracket w \rrbracket_\beta = v$ . Consider the example  $\llbracket [x^3]_{x=2} = 8 + 12\varepsilon \rrbracket$ , where “ $x^3$ ” is a string from the grammar shown in Fig. 1 and the input string evaluates to  $8 + 12\varepsilon$  under the binding  $x = 2$ . Clearly, if the language does not support variables, the context  $\beta$  is always empty.

**Problem Statement.** Given a sketch of an attribute grammar,  $\mathcal{G}^\bullet$ , an example set  $E$  and a domain-specific language (DSL)  $D$ , synthesize instantiations of the holes by strings,  $w \in D$ , such that the resulting attribute grammar agrees with all examples in  $E$ .

In other words, PĀṆINI synthesizes functions  $f_1, \dots, f_n$  in the domain-specific language  $D$  such that the completion  $\mathcal{G}^{\{f_1, \dots, f_n\}}$  satisfies all examples in  $E$ .

#### A. Motivating example: Automated Synthesis of a Forward Differentiation Engine

We will use synthesis of an automatic forward differentiation engine using dual numbers [13] as our motivating example. We start with a short tutorial on how dual numbers are used for forward differentiation.

1) *Forward Differentiation using Dual numbers:* Dual numbers, written as  $a + b\varepsilon$ , captures both the value of a function  $f(x)$  (in the *real* part,  $a$ ), and that its differentiation with respect to the variable  $x$ ,  $f'(x)$ , (in the *dual* part,  $b$ )—within the same number. Clearly,  $a, b \in \mathbb{R}$  and we assume  $\varepsilon^2 = 0$  (as it refers to the second-order differential, that we are not interested to track). The reader may draw parallels to complex numbers that are written as  $a + ib$ , where ‘ $i$ ’ identifies the imaginary part, and  $i^2 = -1$ .

Let us understand forward differentiation by calculating  $f'(x)$  at  $x = 3$  for the function  $f(x) = x^2 + 4x + 5$ .

First, the term  $x$  needs to be converted to a dual number at  $x = 3$ . For  $x = 3$ , the real part is clearly 3. To find the dual

part, we differentiate the term with respect to variable  $x$ , i.e.  $\frac{dx}{dx}$  that evaluates to 1. Hence, the dual number representation of the term  $x$  at  $x = 3$  is  $3 + 1\varepsilon$ .

Now, dual value of the term  $x^2$  can be computed simply by taking a square of the dual representation of  $x$ :

$$\overbrace{(3 + 1\varepsilon)^2}^{x^2} = \overbrace{(3 + 1\varepsilon)}^x * \overbrace{(3 + 1\varepsilon)}^x = 3^2 + (2*3*\varepsilon) + \varepsilon^2 = 9 + 6\varepsilon + 0$$

Finally, the dual number representation for the constant 4 is  $4 + 0\varepsilon$  (as differentiation of constant is 0). Similarly, the dual value for  $4x$ :  $(4 + 0\varepsilon) * (3 + 1\varepsilon) = 12 + 4\varepsilon + 0$ . So, we can compute the dual number for  $f(x) = x^2 + 4x + 5$  as:

$$\overbrace{(9 + 6\varepsilon)}^{x^2} + \overbrace{(12 + 4\varepsilon)}^{4x} + \overbrace{(5 + 0\varepsilon)}^5 = 26 + 10\varepsilon$$

Hence, the value of  $f(x)$  at  $x = 3$  is  $f(3) = 26$  (real part of the dual number above) and that of its derivative,  $f'(x) = 2x + 4$  is  $f'(3) = 10$ , which is indeed given by the the dual part for the dual number above.

2) *Synthesizing a forward differentiation engine*: The attribute grammar in Fig. 1 (adapted from [14]) implements forward differentiation for expressions in the associated context-free grammar; we will use this attribute grammar to illustrate our synthesis algorithm. `lookUp( $\Omega$ , var)` returns the value of the variable `var` from symbol table  $\Omega$ .

We synthesize programs for the required functionalities for the holes from the domain-specific language (DSL) shown in Equation 1. Function `pow( $a, c$ )` calculates  $a$  raised to the power of  $c$ . We assume the availability of an input-output oracle, `Oracle( $w(\beta)$ )`, that returns the expected semantic value for string  $w$  under the context  $\beta$ .

$$\begin{aligned} \text{Fun} &::= C + C\varepsilon \\ C &::= \text{var} \mid \text{num} \mid 1 \mid 0 \mid -C \mid C + C \mid C - C \mid C * C \\ &\quad \mid \sin(C) \mid \cos(C) \mid \text{pow}(C, \text{num}) \end{aligned} \quad (1)$$

### B. Synthesis of semantic actions

Our core insight towards solving this synthesis problem is that the sequence of semantic actions corresponding to the syntax-directed evaluation of any string on the attribute grammar constitutes a loop-free program.

Fig. 5 shows the loop free program from the semantic evaluation of the example

$\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3} = 26 + 10\varepsilon$ ; the Hoare triple captures the synthesis constraints over the holes.

Similarly, our algorithm constructs constraints (as Hoare triples) over the set of all examples  $E$  (e.g.

$$\begin{aligned} \llbracket x + x \rrbracket_{x=13} = 26 + 2\varepsilon, \quad \llbracket 3 - x \rrbracket_{x=7} = -4 - 1\varepsilon, \\ \llbracket x * x \rrbracket_{x=4} = 16 + 8\varepsilon, \\ \llbracket \sin(x^2) \rrbracket_{x=3} = 0.41 - 5.47\varepsilon, \end{aligned}$$

$$\begin{aligned} \llbracket \cos(x^2) \rrbracket_{x=2} = -0.65 + 3.02\varepsilon, \\ \llbracket x * \cos(x) \rrbracket_{x=4} = -2.61 + 2.37\varepsilon. \end{aligned}$$

Synthesizing definitions for holes that satisfy Hoare triple constraints of all the above examples yields a valid completion of the sketch of the attribute grammar (see Fig. 3). As the above queries are “standard” program synthesis queries, they can be answered by off-the-shelf program synthesis tools [15], [16]. Hence, our algorithm reduces the problem of synthesizing semantic actions for attribute grammars to solving a conjunction of program synthesis problems.

While the above conjunction can be easily folded into a single program synthesis query and offloaded to a program synthesis tool, quite understandably, it will not scale. To scale the above problem, we employ a refutation-guided inductive synthesis procedure: we sort the set of examples by increasing complexity, completing the holes for the easier instances first.

The synthesized definitions are *frozen* while handling new examples; however, unsatisfiability of a synthesis call with frozen procedures *refutes* the prior synthesized definitions. Say we need to synthesize definitions for  $\{h_0^*, \dots, h_9^*\}$  and examples  $\{e_1, \dots, e_{i-1}\}$  have already been handled, with definitions  $\{h_1^* = f_1, \dots, h_5^* = f_5\}$  already synthesized. To handle a new example,  $e_i$ , we issue a synthesis call for procedures  $\{h_6^*, \dots, h_9^*\}$  with definitions  $\{h_1^* = f_1, \dots, h_5^* = f_5\}$  frozen. Say, the constraint corresponding to  $e_i$  only includes calls  $\{h_2^*, h_4^*, h_6^*, h_8^*\}$  and the synthesis query is unsatisfiable. In this case, we unfreeze *only* the participating frozen definitions (i.e.  $\{h_2^*, h_4^*\}$ ) and make a new synthesis query. As new query only attempts to synthesize a few new calls (with many participating calls frozen to previously synthesized definitions), this algorithm scales well.

For example, consider the grammar in Fig. 1: the loop-free program resulting from the semantic evaluation of the input  $\llbracket x + x \rrbracket_{x=13} = 26 + 2\varepsilon$  (say trace  $t_1$ ) includes only one  $h_1^*$ . Hence, we synthesize  $h_1^*$  with only the constraint  $\{x = 13\}t_1\{output = 26 + 2\varepsilon\}$ , that results in the definition shown in Fig. 3a. Next, we consider the input  $\llbracket x * x \rrbracket_{x=4} = 16 + 8\varepsilon$ ; its constraint includes the holes  $h_3^*$ , which is synthesized as the function definition shown in Fig. 4. Now, with  $\{h_1^*, h_3^*\}$  frozen to their respective synthesized definitions, we attempt to handle  $\llbracket x * \cos(x) \rrbracket_{x=4} = -2.61 + 2.37\varepsilon$ . Its constraint includes the holes  $\{h_3^*, h_6^*\}$ ; now we *only* attempt to synthesize  $h_6^*$  while constraining  $h_3^*$  to use the definition in Fig. 4.

```

{x = 3}
K1.val ← 3 + 1ε;
K2.val ← h4*(K1.val, 2);
K3.val ← 4 + 0ε;
F1.val ← h3*(K3.val, K1.val);
E1.val ← h1*(K2.val, F1.val);
K4.val ← 5 + 0ε;
output ← h1*(E1.val, K4.val);
{output = 26 + 10ε}

```

Fig. 5: Hoare triple constraint for  $x^2 + 4x + 5$  at  $x = 3$

```

h3*(a1 + a2ε, b1 + b2ε):
  r ← a1 * b1
  d ← b1 + b2 + 3 * a2
  return r + dε

```

Fig. 4: Wrong definition of  $h_3^*$

In this case the synthesizer fails to synthesize  $h_6^\bullet$  since the synthesized definition for  $h_3^\bullet$  is incorrect, thereby *refuting* the synthesized definition of  $h_3^\bullet$ . Hence, we now unfreeze  $h_3^\bullet$  and call the synthesis engine again to synthesize both  $h_3^\bullet$  and  $h_6^\bullet$  together. This time we succeed in inferring correct synthesized definition as shown in Fig. 3.

### C. Example Generation

We propose a new coverage metric, *derivation coverage*, to generate good samples to drive synthesis. Let us explain derivation coverage with an example,  $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3}$  from the grammar in Fig. 1. The leftmost derivation of this string covers eight productions,  $\{1, 2, 4, 5, 6, 7, 10, 11\}$  out of a total of 11 productions. Intuitively, it implies that the Hoare triple constraint from its semantic evaluation will *test* the semantic actions corresponding to these productions.

Similarly, the Hoare logic constraint from the example  $\llbracket x^2 + 7 * x + \sin(x) \rrbracket_{x=2}$  will cover 9 of the productions,  $\{1, 2, 4, 5, 6, 7, 8, 10, 11\}$ . As it *also* covers the semantic action for the production 8, it tests an additional behavior of the attribute grammar. On the other hand, the example  $\llbracket x^3 + 5x \rrbracket_{x=5}$  invokes the productions,  $\{1, 2, 4, 5, 6, 7, 10, 11\}$ . As all these semantic actions have already been *covered* by the example  $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3}$ , it does not include the semantic action of any new set of productions.

In summary, derivation coverage attempts to abstract the derivation of a string as the *set of productions in its leftmost derivation*. It provides an effective metric for quantifying the quality of an example suite and also for building an effective example generation system.

**Validation.** Our example generation strategy can start off by *sampling* strings  $w$  from the grammar (that improve derivation coverage), and context  $\beta$ ; next, it can query the oracle for the intended semantic value  $v = Oracle(w \langle \beta \rangle)$  to create an example  $\llbracket w \rrbracket_\beta^{\mathcal{G}} = v$ .

Consider that the algorithm finds automatically an example  $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=3}$ . Now, there are two possible, semantically distinct definitions that satisfy the above constraint (see Fig. 3c and Fig. 4), indicating that the problem is underconstrained. Hence, our system needs to select additional examples to resolve this. One solution is to sample multiple contexts on the same string to create multiple constraints:

- $\{x = 2\} K_1.val \leftarrow 2 + 1\epsilon; \dots \{output = 13 + 6\epsilon\}$
- $\{x = 4\} K_1.val \leftarrow 4 + 1\epsilon; \dots \{output = 29 + 10\epsilon\}$

The above constraints resolve the ambiguity and allows the induction of a semantically unique definitions. The check for semantic uniqueness can be framed as a check for *distinguishing inputs*: given a set of synthesized completion  $\mathcal{G}^{\{f_1, \dots, f_n\}}$ , we attempt to synthesize an alternate completion  $\mathcal{G}^{\{g_1, \dots, g_n\}}$  and an example string  $w$  (and context  $\beta$ ) such that  $\llbracket w \rrbracket_\beta^{\mathcal{G}^{\{f_1, \dots, f_n\}}} \neq \llbracket w \rrbracket_\beta^{\mathcal{G}^{\{g_1, \dots, g_n\}}}$ . In other words, for the same string (and context), the attribute grammar returns different evaluations corresponding to the two completions. For example,  $\llbracket x^2 + 4 * x + 5 \rrbracket_{x=2}$  is a *distinguishing inputs*

---

### Algorithm 1: SYNTHHOLES( $\mathcal{G}^\bullet, T, R, D$ )

---

```

1  $\varphi \leftarrow \top$ ;
2  $\mathcal{G}_1^\bullet \leftarrow \mathcal{G}^\bullet[R]$ ;
3 for  $\langle w, v \rangle \in T$  do
4    $t \leftarrow \text{GENTRACE}(\llbracket w \rrbracket_{\mathcal{G}_1^\bullet})$ ;
5    $\varphi \leftarrow \varphi \wedge (out(t) = v)$ ;
6  $B \leftarrow \text{SYNTHESIZE}(\varphi, D)$ ;
7 return  $B$ ;
```

---

witnessing the ambiguity between the definitions shown in Fig. 3c and Fig. 4.

On the other hand, the algorithm could have sampled other strings (instead of contexts) for additional constraints. PĀÑINI prefers the latter; that is, it first generates a *good* example suite (in terms of derivation coverage) and only uses distinguishing input as a validation (post) pass. If such inputs are found, additional contexts are added to resolve the ambiguity.

We provide the detailed algorithm of example generation in the extended version [10].

## IV. ALGORITHM

Given an attribute grammar  $\mathcal{G}^\bullet$ , a set of holes  $h_i \in H$ , a domain-specific language  $D$ , an example suite  $E$  and a context  $\beta$ , PĀÑINI attempts to find instantiations  $g_i$  for  $h_i$  such that,

**Find**  $\{g_1, \dots, g_{|H|}\} \in D$  **such that**  $\forall \langle s, \beta, v \rangle \in E. \llbracket s \rrbracket_\beta^{\mathcal{G}} = v$  (2)

where the attribute grammar  $\mathcal{G} = \mathcal{G}^\bullet[g_1/h_1, \dots, g_{|H|}/h_{|H|}]$  and variable bindings  $\beta$  maps variables in  $s$  to values.

### A. Basic Scheme: ALLATONCE

Our core synthesis procedure (Algorithm 1), SYNTHHOLES( $\mathcal{G}^\bullet, E, R, D$ ), accepts a sketch  $\mathcal{G}^\bullet$ , an example (or test) suite  $E$ , a set of *ready* functions  $R$  and a DSL  $D$ ; all holes whose definitions are available are referred to as *ready* functions. When SYNTHHOLES is used as a top-level procedure (as in the current case),  $R = \emptyset$ ; if not empty, the definitions of the ready functions are substituted in the sketch  $\mathcal{G}^\bullet$  to create a new sketch  $\mathcal{G}_1^\bullet$  on the remaining holes (Line 2). We refer to the algorithm where  $R = \emptyset$  at initialization as the ALLATONCE algorithm.

Our algorithm exploits the fact that a syntax-directed semantic evaluation of a string  $w$  on an attribute grammar  $\mathcal{G}$  produces a loop-free program. It attempts to compute a symbolic encoding of this program trace in the formula  $\varphi$  (initialized to  $\text{true}$  in Line 1). GENTRACE() instruments the semantic evaluation on the string  $w$  to collect a symbolic trace (the loop-free program) consisting of the set of instructions encountered during the syntax-directed execution of the attribute grammar (Line 4); an output from an operation that is currently a *hole* is appended as a symbolic variable. The assertion that the expected output  $v$  matches the final symbolic output  $out(t)$  from the trace  $t$  is appended to the list of constraints (Line 5). Finally, we use a program synthesis procedure, *Synthesize* with the constraints  $\varphi$  in an attempt to synthesize suitable function

definitions for the holes in  $\varphi$  (Line 6). Given a constraint in terms of a set of input vector  $\vec{x}$  and function symbols (corresponding to holes)  $\vec{h}$ ,

$$\text{SYNTHESIZE}(\varphi(\vec{x}, \vec{h})) := \vec{h} \text{ such that } \exists \vec{h}. \forall \vec{x}. \varphi(\vec{x}, \vec{h}) \quad (3)$$

We will use an example from forward differentiation (Fig. 1) to illustrate this. Let us consider two inputs,  $\llbracket x^2 - 2 * x \rrbracket_{x=3} = 3 + 4\varepsilon$  and  $\llbracket 3 * x + 6 \rrbracket_{x=2} = 12 + 3\varepsilon$ . For the first input  $\llbracket 3 * x + 6 \rrbracket_{x=2} = 12 + 3\varepsilon$ , the procedure `GENTRACE()` (Line 4) generates a symbolic trace (denoted  $t_1$ ):

$$\{x_1 = 2 + 1\varepsilon; \alpha_1 = h_3^\bullet(3 + 0\varepsilon, x_1); out_{t_1} = h_1^\bullet(\alpha_1, 6 + 0\varepsilon)\}$$

The following symbolic constraint is generated from above trace:

$$\varphi_{t_1} \equiv (x_1 = 2 + 1\varepsilon \wedge \alpha_1 = h_3^\bullet(3 + 0\varepsilon, x_1) \wedge out_{t_1} = h_1^\bullet(\alpha_1, 6))$$

In the trace  $t_1$ , operations  $h_1^\bullet$  and  $h_3^\bullet$  are holes and variables, i.e.,  $\alpha_1, out_{t_1}$  are the fresh symbolic variables. In the next step (line 5), the constraints generated from trace  $t_1$  is added,

$$\varphi \equiv \top \wedge (\varphi_{t_1} \wedge out_{t_1} = 12 + 3\varepsilon)$$

In the next iteration of the loop at line 3, the algorithm will take the second input, (i.e.,  $\llbracket x^2 - 2 * x \rrbracket_{x=3} = 3 + 4\varepsilon$ ). In this case, `GENTRACE()` will generate following trace ( $t_2$ ),

$$\{x_2 = 3 + 1\varepsilon; \alpha_2 = h_4^\bullet(x_2, 2); \alpha_3 = h_3^\bullet(2 + 0\varepsilon, x_2); out_{t_2} = h_2^\bullet(\alpha_3, \alpha_4)\}$$

The generated constraints from  $t_2$  will be,

$$\varphi_{t_2} \equiv (x_2 = 3 + 1\varepsilon \wedge \alpha_2 = h_4^\bullet(x_2, 2) \wedge \alpha_3 = h_3^\bullet(2 + 0\varepsilon, x_2) \wedge out_{t_2} = h_2^\bullet(\alpha_3, \alpha_4))$$

At line 5, new constraints will be,

$$\varphi \leftarrow \top \wedge (\varphi_{t_1} \wedge out_{t_1} = 12 + 3\varepsilon) \wedge (\varphi_{t_2} \wedge out_{t_2} = 3 + 4\varepsilon)$$

At line 6, with  $\varphi$  as constraints, the algorithm will attempt to synthesize definitions for the holes (i.e.,  $h_1^\bullet, h_2^\bullet, h_3^\bullet$  and  $h_4^\bullet$ ).

## B. Incremental Synthesis

The `ALLATONCE` algorithm exhibits poor scalability with respect to the size of the grammar and the number of examples. The route to a scalable algorithm could be to incrementally learn the definitions corresponding to the holes and make use of the functions synthesized in the previous steps to discover new ones in the subsequent steps.

However, driving synthesis one example at a time will lead to overfitting. We handle this complexity with a two-pronged strategy: (1) we partition the set of examples by the holes for which they need to synthesize actions, (2) we solve the synthesis problems by their difficulty (in terms of the number of functions to be synthesized) that allows us to memoize their results for the more challenging examples. We refer to this example as the `INCREMENTALSYNTHESIS` algorithm.

---

## Algorithm 2: SYNTHATTRGRAMMAR( $\mathcal{G}^\bullet, E, D$ )

---

```

1  $T \leftarrow \emptyset$ ;
2  $R \leftarrow \emptyset$ ;
3 while  $T \neq E$  do
4    $\langle w, v \rangle \leftarrow \text{SELECTEXAMPLE}(E \setminus T)$ ;
5    $Z \leftarrow \text{GETSKETCHYPRODS}(\mathcal{G}^\bullet, w)$ ;
6   if  $Z \subseteq R$  then
7      $\mathcal{G}_1^\bullet \leftarrow \mathcal{G}^\bullet[R]$ ;
8     if  $\llbracket w \rrbracket_{\mathcal{G}_1^\bullet} = v$  then
9        $T \leftarrow T \cup \{\langle w, v \rangle\}$ ;
10      continue;
11    else
12       $R \leftarrow R \setminus Z$ ;
13       $T_e \leftarrow T \cup \{\langle w, v \rangle\}$ ;
14    else
15       $T_e \leftarrow \{\langle w_i, v_i \rangle \mid w \cong w_i, \langle w_i, v_i \rangle \in E\}$ ;
16     $B \leftarrow \text{SYNTHHOLES}(\mathcal{G}^\bullet, T_e, R, D)$ ;
17    if  $B = \emptyset$  then
18      if  $R \cap Z \neq \emptyset$  then
19         $R \leftarrow R \setminus Z$ ;
20         $B \leftarrow \text{SYNTHHOLES}(\mathcal{G}^\bullet, T \cup T_e, R, D)$ ;
21      if  $B = \emptyset$  then
22        return  $\emptyset$ ;
23     $R_f \leftarrow \{(p_i : \{\dots, h_i \rightarrow B[h_i], \dots\}) \mid p_i \in Z \setminus R, h_i \in \text{holes}(\Gamma(p_i))\}$ ;
24     $R \leftarrow R \cup R_f$ ;
25     $T \leftarrow T \cup T_e$ ;
26 return  $R$ ;
```

---

**Derivation Congruence.** We define an equivalence relation, *derivation congruence*, on the set of strings  $w \in \mathcal{L}(\mathcal{G})$ : strings  $w_1, w_2 \in \mathcal{L}(\mathcal{G})$  are said to be *derivation congruent*,  $w_1 \cong_G w_2$  w.r.t. the grammar  $G$ , if and only if both the strings  $w_1$  and  $w_2$  contain the same set of productions in their respective derivations. For example,  $w_1 : \llbracket 3 * x + 5 \rrbracket_{x=2}$ ,

$$w_2 : \llbracket 5 * x + 12 \rrbracket_{x=3} \text{ and } w_3 : \llbracket 4 * x + 7 * x \rrbracket_{x=3}.$$

Note that though the strings  $w_1, w_2$  and  $w_3$  are derivation congruent to each other, while  $w_1$  and  $w_2$  have similar parse trees,  $w_3$  has a quite different parse tree. So, intuitively, all these strings are definition congruent to each other, as, even with different parse trees, they involve the same set of productions ( $\{1, 2, 4, 5, 6, 10, 11\}$ ) in their leftmost derivation.

Algorithm 2 shows our incremental synthesis strategy. Our algorithm maintains a set of examples (or tests)  $T$  (line 2) that are consistent with the current set of synthesized functions for the holes; the currently synthesized functions (referred to as *ready* functions), along with the respective ready productions, are recorded in  $R$  (line 1). The algorithm starts off by selecting the *easiest* example  $\langle w, v \rangle$  at line 4 such that the cardinality of the set of sketchy production,  $Z$ , in the derivation of  $w$  is the minimal among all examples not in  $T$ . The set  $R$  maintains a map from the set of sketchy productions to a set of assignments to functions synthesized (instantiations) for each hole contained in the respective semantic actions.

If all sketchy productions,  $Z$ , in the derivation of  $w$  are now *ready*, we simply *test* (line 6) to check if a syntax-guided evaluation with the currently synthesized functions in

$R$  yield the expected value  $v$ : if the test passes, we add the new example to the set of passing examples in  $T$  (line 9). Otherwise, as the current hole instantiations in  $R$  is not consistent for the derivation of  $w$ , at line 12 we remove all the synthesized functions participating in syntax-directed evaluation of  $w$  (which is exactly  $Z$ ). Furthermore, removal of some functions from  $R$  requires us to re-assert the new functions on all the past examples (contained in  $T$ ) in addition to the present example (line 13).

If all the sketchy productions ( $Z$ ) in the derivation of  $w$  are not ready, we attempt to synthesize functions for the *missing* holes, *with the set of current definitions in  $R$  provided in the synthesis constraint*.

The synthesis procedure (line 16), if successful, yields a set of function instantiation for the holes. In this case, the solution set from  $B$  is accumulated in  $R$ , and the set of *passing* tests extended to contain the new examples in  $T_e$ .

However, synthesis may fail as some of the current definitions in  $R$  that were assumed to be correct and included in the synthesis constraint is not consistent with the new examples ( $T_e$ ). In this case, we remove the instantiations of all such holes occurring in the syntax-directed evaluation of  $w$  (line 19) and re-attempt synthesis (line 20). If this attempt fails too, it implies that no instantiation of the holes exist in the provided domain-specific language  $D$  (line 22).

We provide a detailed example on the run of this algorithm in the extended version [10].

**Theorem.** If the algorithm terminates with a non-empty set of functions,  $\mathcal{G}^\bullet$  instantiated with the synthesized functions will satisfy the examples in  $E$ ; that is, the synthesized functions satisfy Equation 2.

The proof is a straightforward argument with the inductive invariant that at each iteration of the loop, the  $\mathcal{G}^\bullet$  instantiated with the functions in  $R$  satisfy the examples in  $T$ .

## V. EXPERIMENTS

Our experiments were conducted in Intel(R) Xeon(R) CPU E5-2620 @ 2.00GHz with 32 GB RAM and 24 cores on a set of benchmarks shown in Table I. PĀÑINI uses FLEX [17] and BISON [3] for performing a syntax-directed semantic evaluation over the language strings. PĀÑINI uses SKETCH [15] to synthesize function definitions over loop-free programs, and the symbolic execution engine CREST [18] for generating example-suites guided by derivation coverage.

We attempt to answer the following research questions:

- Can PĀÑINI synthesize attribute grammars from a variety of sketches?
- How do INCREMENTALSYNTHESIS and ALLATONCE algorithms compare?
- How does PĀÑINI scale with the number of holes?
- How does PĀÑINI scale with the size of the grammar?

The default algorithm for PĀÑINI is the INCREMENTALSYNTHESIS algorithm; unless otherwise mentioned, PĀÑINI refers to the implementation of INCREMENTALSYNTHESIS (Algorithm 2) for synthesis using examples generation guided by derivation coverage (detailed explanation available in the

extended version [10]). While ALLATONCE works well for small grammars with few examples, INCREMENTALSYNTHESIS scales well even for larger grammars, both with the number of holes and size of the grammar.

PĀÑINI can synthesize semantic-actions across both synthesized and inherited attributes. Some of our benchmarks contain inherited-attributes: for example, benchmark b8 uses inherited-attributes to pass the type information of the variables. Inherited-attributes pose no additional challenge; they are handled by the standard trick of introducing “marker” non-terminals [19].

### A. Attribute Grammar Synthesis

We evaluated PĀÑINI on a set of attribute grammars adapted from software in open-source repositories [14], [19]–[24]. Table I shows the benchmarks, number of productions (**#P**), number of holes (**#H**), input example, solving time (**Time**, **AAO** for ALLATONCE and **IS** for INCREMENTALSYNTHESIS) and number of times a defined function was refuted (**#R**). Please recall that ALLATONCE refers to Algorithm 1 (§IV-A) and INCREMENTALSYNTHESIS refers to Algorithm 2 (§IV-B).

We provide more detailed descriptions of the benchmarks b1 to b9 in the extended version [10]. The benchmark b10 is the forward differentiation example described in §III-A. Benchmarks b11 and b12 are quite complex benchmarks that interpret a (subset) of Java bytecode and compile C code:

**b11 Bytecode interpreter.** Interpreter for a subset of Java bytecode; it supports around 36 instructions [25] of different type, i.e., load-store, arithmetic, logic and control transfer instructions.

**b12 Mini-compiler.** Fig. 6 shows the different steps of synthesizing semantic actions in mini-compiler. Fig. 6b is a sample input for the mini-compiler. Fig. 6a shows snippet of the attribute grammar for mini-compiler. Fig. 6c shows the two-address code generated from the input code shown in Fig. 6b, where  $h_a^\bullet$  and  $h_b^\bullet$  are two holes in the two-address code. Finally, in Fig. 6d shows the synthesized definition for  $h_a^\bullet$  and  $h_b^\bullet$  in the target language for two-address code.

Fig. 8 attempts to capture the fraction of time taken by the different phases of PĀÑINI: example generation and synthesis. Not surprisingly, the synthesis phase dominates the cost as it requires several invocation of the synthesis engines, whereas, the example generation phase does not invoke synthesis engines or smt solvers. Further, the difference in time spend in these two phases increases as the benchmarks get more challenging.

### B. ALLATONCE *v/s* INCREMENTALSYNTHESIS

*1) Scaling with holes:* Fig. 9a and Fig. 9b show PĀÑINI scales with the sketches with increasingly more holes. We do this study for forward differentiation (b10) and bytecode interpreter (b11). As can be seen, PĀÑINI scales very well. On the other hand, ALLATONCE works well for small instances but soon blows up, timing out on all further instances. The interesting jump in b10 (at #Holes=8) was seen when we

S  $\rightarrow$  MAIN B  
 | MAIN B

...

A  $\rightarrow$  T

| E + T A.val =  $h_a^*(E.val, T.val)$  ;

| E - T A.val =  $h_b^*(E.val, T.val)$  ;

T  $\rightarrow$  F

| T \* F T.val =  $h_c^*(T.val, F.val)$  ;

| T / F T.val =  $h_d^*(T.val, F.val)$  ;

...

(a) Attribute grammar sketch for mini-compiler

```
int main{
  int a, b, c;
  a = 4;
  b = a + 3; //  $h_a^*$ 
  c = a - b; //  $h_b^*$ 
  return c;
}
```

(b) A simple C code

op	arg1	arg2	dst
assign	4		a
$h_a^*$	a	9	T0
assign	T0		b
$h_b^*$	a	b	T1
assign	T1		T2
ret			T2

(c) Three-address code generated from C code in Fig. 6b

$h_a^*$  a b:  
 emit("load r1 a")  
 emit("load r2 b")  
 emit("plus r1 r2")

$h_b^*$  a b:  
 emit("load r1 a")  
 emit("load r2 b")  
 emit("sub r1 r2")

(d) Synthesized definition for  $h_a^*, h_b^*$

Fig. 6: Synthesis of mini-compiler (b12)

TABLE I: Description of benchmarks

Id	Benchmark	#P	#H	Example	#R	Time (s)	
						AAO	IS
b1	Count ones	5	1	11001	0	3.2	3.1
b2	Binary to integer	5	1	01110	0	3.6	2.9
b3	Prefix evaluator	7	4	+ 3 4	0	TO	10.1
b4	Postfix evaluator	7	4	2 3 4 * +	0	TO	10.5
b5	Arithmetic calculator	8	4	5 * 2 + 8	0	TO	12.8
b6	Currency calculator	10	4	USD 3 + INR 8	0	TO	13.6
b7	if-else calculator	10	4	if(3+4 == 3) then 44; else 73;	1	TO	21.7
b8	Activation record layout	10	3	int a , b;	0	TO	13.8
b9	Type checker	11	5	(5 - 2) == 3	1	TO	15.4
b10	Forward differentiation	20	12	x*pow(x,3)	2	TO	39.2
b11	Bytecode interpreter	39	36	bipush 3; bipush 4; iadd;	3	TO	141.4
b12	Mini-compiler	43	6	int main(){ return 2+3;}	0	TO	9.2

```
init {
  run Foo(8+(6-7));
}
```

(e) PROMELA source code

```
node n1 = node(val=8);
node n2 = node(val=6);
node n3 = node(val=7);
node n4 =  $h_a^*$ (n2, n3);
node n5 =  $h_b^*$ (n1, n4);
node n6 =  $h_c^*$ ('Foo', n5);
```

(f) Trace generated

Fig. 7: Trace generation for AST construction

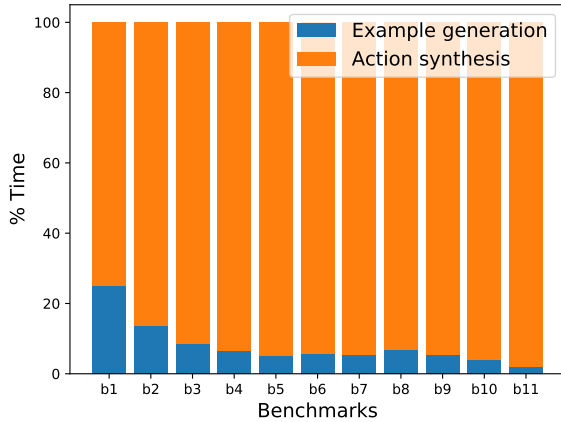


Fig. 8: Stacked bar graph for the % time spent in example creation and synthesis

started adding holes for the definitions of the more complex operators like  $\sin()$  and  $\cos()$ .

2) *Scaling with size of grammar*: Table I shows that INCREMENTALSYNTHESIS scales well with the size of the grammar (by the number of productions). On the other hand, ALLATONCE works well for the benchmarks b1 and b2 as they have only one hole while it times out for the rest.

The complexity of INCREMENTALSYNTHESIS is independent of the size of the attribute-grammar but dependent on the length of derivations and the size of the semantic actions. The current state of synthesis-technology allows PĀṆINI to synthesize practical attribute grammars that have a large number of productions but mostly “small” semantic actions and where short derivations can “cover” all productions. Further, any improvement in program-synthesis technology automatically improves the scalability of PĀṆINI.

## VI. CASE STUDY

We undertook a case-study on the parser specification of the SPIN [5] model-checker. SPIN is an industrial-strength model-checker that verifies models written in the PROMELA [26] language against linear temporal logic (LTL) specifications. SPIN uses YACC [2] to build its parser for PROMELA. The modelling language, PROMELA, is quite rich, supporting variable assignments, branches, loops, arrays, structures, procedures etc. The attribute grammar specification in the YACC language is more than 1000 lines of code (ignoring newlines) having 280 production rules.

The semantic actions within the attribute grammar in the YACC description handle multiple responsibilities. We selected two of its operations:



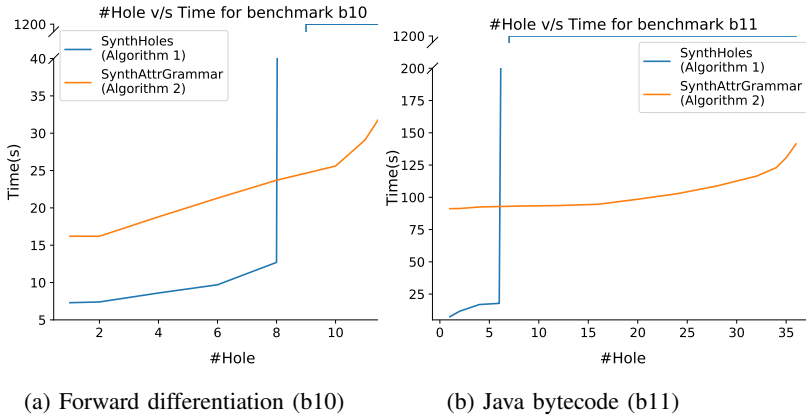


Fig. 9: #Hole v/s Time for benchmarks b10 and b11

a) *Constant folding array indices:* As the PROMELA code is parsed, semantic actions automatically constant-fold array indices (see Fig. 10). We removed all the actions corresponding to constant-folding by inserting 8 holes in the relevant production rules (these correspond to the non-terminal `const_expr`). The examples to drive the synthesis consisted of PROMELA code with arrays with complex expressions and the target output was the optimized PROMELA code. PĀÑINI was able to automatically synthesize this constant-folding optimization within less than 4 seconds.

b) *AST construction:* A primary responsibility of the semantic analysis phase is to construct the abstract syntax tree (AST) of the source PROMELA code. We, next, attempted to enquire if PĀÑINI is capable of this complex task.

In this case, each example includes a PROMELA code as input and a tree (i.e. the AST) as the output value. We removed the existing actions via 23 holes. These holes had to synthesize the end-to-end functionality for a production rule with respect to building the AST: that, the synthesized code would decide the type of AST node to be created and the correct order of inserting the children sub-trees.

Run of the example suite on the sketchy productions generates a set of programs (one such program shown in Fig. 7); these programs produce *symbolic* ASTs that non-deterministically assigns type to nodes and assigns the children nodes. We leverage the support of references in Sketch to define self-referential nodes.

We insert constraints that establish tree isomorphism by recursively matching the symbolic ASTs with the respective output ASTs (available in example suite); for example, in Fig. 11 isomorphism constraints are enforced on the concrete and the symbolic ASTs. Sketch resolves the non-determinism en route to synthesizing the relevant semantic actions. In this case-study, PĀÑINI was able to synthesize the actions corresponding to the 23 holes within 20 seconds.

## VII. RELATED WORK

Program synthesis is a rich area with proposals in varying domains: bitvectors [27], [28], heap manipulations [29]–[33], bug synthesis [34], differential privacy [35], [36], invariant

```
init {
  int flags[(5 * 25) - 42];
  int v = flags[10 - 4 + (9 / 3)];
}
```

(a) PROMELA source

```
init {
  int flags[83];
  int v = flags[9];
}
```

(b) PROMELA optimized

Fig. 10: Constant folding in PROMELA

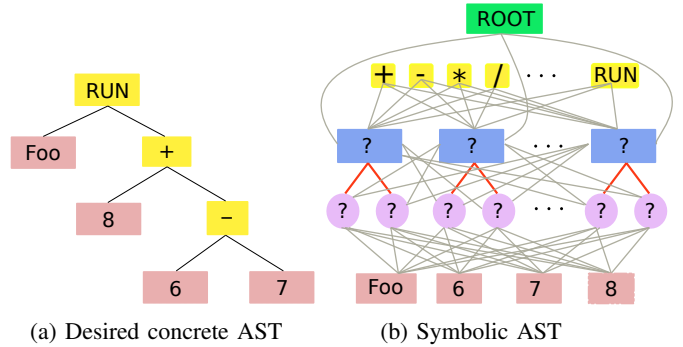


Fig. 11: Desired AST for code in Fig. 7 and symbolic AST. Grey lines (in Fig. 11b) denote symbolic choices.

generation [37], Skolem functions [38]–[40], synthesis of fences and atomic blocks [41] and even in hardware security [42]. However, to the best of our knowledge, ours is the first work on automatically synthesizing semantics actions for attribute grammars.

There has some work on automatically synthesizing parsers: PARSIFY [43] provides an interactive environments to automatically infer grammar rules to parse strings; it is been shown to synthesize grammars for Verilog, Tiger, Apache Logs, and SQL. CYCLOPS [44] builds an encoding for *Parse Conditions*, a formalism akin to *Verification Conditions* but for parseable languages. Given a set of positive and negative examples, CYCLOPS, automatically generates an LL(1) grammar that accepts all positive examples and rejects all negative examples. Though none of them handle attribute grammars, it may be possible to integrate them with PĀÑINI to synthesize *both* the context-free grammar and the semantic actions. We plan to pursue this direction in the future.

We are not aware of much work on testing attribute grammars. We believe that our *derivation coverage* metric can also be potent for finding bugs in attribute grammars, and can have further applications in dynamic analysis [45]–[47] and statistical testing [48], [49] of grammars. However, the effectiveness of this metric for bug-hunting needs to be evaluated and seems to be a good direction for the future.

## REFERENCES

- [1] D. E. Knuth, "Semantics of context-free languages," *Mathematical systems theory*, vol. 2, no. 2, pp. 127–145, Jun 1968. [Online]. Available: <https://doi.org/10.1007/BF01692511>
- [2] S. C. Johnson and M. Hill, "YACC: Yet Another Compiler Compiler," *UNIX Programmer's Manual*, vol. 2, pp. 353–387, 1978.
- [3] GNU Bison. (last accessed 29 Jun 2021). [Online]. Available: <https://www.gnu.org/software/bison/>
- [4] ANTLR. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/antlr/antlr4>
- [5] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997.
- [6] Q3B SMT solver. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/martijnjonas/Q3B>
- [7] H. Barbosa, C. W. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, "cvc5: A versatile and industrial-strength SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 415–442. [Online]. Available: [https://doi.org/10.1007/978-3-030-99524-9\\_24](https://doi.org/10.1007/978-3-030-99524-9_24)
- [8] C Intermediate Language (CIL). (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/cil-project/cil>
- [9] MySQL. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/mysql/mysql-server>
- [10] P. K. Kalita, M. J. Kumar, and S. Roy, "Synthesis of semantic actions in attribute grammars," 2022. [Online]. Available: <https://arxiv.org/abs/2208.06916>
- [11] K. J. Rähä and M. Saarinen, "Testing attribute grammars for circularity," *Acta Informatica*, vol. 17, no. 2, pp. 185–192, Jun 1982. [Online]. Available: <https://doi.org/10.1007/BF00288969>
- [12] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, p. 576–580, Oct. 1969. [Online]. Available: <https://doi.org/10.1145/363235.363259>
- [13] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018. [Online]. Available: <http://jmlr.org/papers/v18/17-468.html>
- [14] Automating differentiation using dual numbers. (last accessed 29 Jun 2021). [Online]. Available: <https://blog.demofox.org/2014/12/30/dual-numbers-automatic-differentiation/>
- [15] A. Solar-Lezama, "Program sketching," *Int. J. Softw. Tools Technol. Transf.*, vol. 15, no. 5–6, p. 475–495, oct 2013. [Online]. Available: <https://doi.org/10.1007/s10009-012-0249-7>
- [16] E. Torlak and R. Bodik, "Growing solver-aided languages with Rosette," in *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, ser. Onward! 2013. New York, NY, USA: Association for Computing Machinery, 2013, p. 135–152. [Online]. Available: <https://doi.org/10.1145/2509578.2509586>
- [17] Flex. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/westes/flex>
- [18] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 443–446. [Online]. Available: <https://doi.org/10.1109/ASE.2008.69>
- [19] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [20] A simple calculator. (last accessed 29 Jun 2021). [Online]. Available: <https://www.dabeaz.com/ply/example.html>
- [21] Evaluate postfix expression using YACC. (last accessed 29 Jun 2021). [Online]. Available: <https://prashantkulkarni17.wordpress.com/2011/09/20/evaluate-postfix-expression-using-yacc/>
- [22] Mini-compiler. (last accessed 29 Jun 2021). [Online]. Available: <https://github.com/SandyaSivakumar/Mini-Compiler/>
- [23] Conversion from binary to decimal. (last accessed 29 Jun 2021). [Online]. Available: <https://myprogworld.wordpress.com/2016/04/30/conversion-from-binary-to-decimal/>
- [24] Type check. (last accessed 29 Jun 2021). [Online]. Available: <http://pages.cs.wisc.edu/~fischer/cs536.s06/course.hold/html/NOTES/4.SYNTAX-DIRECTED-TRANSLATION.html#ex2>
- [25] Java Byte Code. (last accessed 29 Jun 2021). [Online]. Available: [https://en.wikibooks.org/wiki/Java\\_Programming/Byte\\_Code](https://en.wikibooks.org/wiki/Java_Programming/Byte_Code)
- [26] Concise Promela Reference. (last accessed 29 Jun 2021). [Online]. Available: <http://spinroot.com/spin/Man/Quick.html>
- [27] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan, "Synthesis of loop-free programs," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 62–73. [Online]. Available: <https://doi.org/10.1145/1993498.1993506>
- [28] A. Solar-Lezama, R. Rabbah, R. Bodik, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 281–294. [Online]. Available: <https://doi.org/10.1145/1065010.1065045>
- [29] S. Roy, "From concrete examples to heap manipulating programs," in *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*, ser. Lecture Notes in Computer Science, F. Logozzo and M. Fähndrich, Eds., vol. 7935. Springer, 2013, pp. 126–149. [Online]. Available: [https://doi.org/10.1007/978-3-642-38856-9\\_9](https://doi.org/10.1007/978-3-642-38856-9_9)
- [30] A. Garg and S. Roy, "Synthesizing heap manipulations via integer linear programming," in *Static Analysis - 22nd International Symposium, SAS 2015, Saint-Malo, France, September 9-11, 2015. Proceedings*, ser. Lecture Notes in Computer Science, S. Blazy and T. P. Jensen, Eds., vol. 9291. Springer, 2015, pp. 109–127. [Online]. Available: [https://doi.org/10.1007/978-3-662-48288-9\\_7](https://doi.org/10.1007/978-3-662-48288-9_7)
- [31] S. Verma and S. Roy, "Synergistic debug-repair of heap manipulations," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, Eds. ACM, 2017, pp. 163–173. [Online]. Available: <https://doi.org/10.1145/3106237.3106263>
- [32] S. Verma and Subhajit Roy, "Debug-localize-repair: A symbiotic construction for heap manipulations," *Formal Methods Syst. Des.*, vol. 58, no. 3, pp. 399–439, 2021. [Online]. Available: <https://doi.org/10.1007/s10703-021-00387-z>
- [33] N. Polikarpova and I. Sergey, "Structuring the synthesis of heap-manipulating programs," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019. [Online]. Available: <https://doi.org/10.1145/3290385>
- [34] S. Roy, A. Pandey, B. Dolan-Gavitt, and Y. Hu, "Bug synthesis: Challenging bug-finding tools with deep faults," in *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, G. T. Leavens, A. Garcia, and C. S. Pasareanu, Eds. ACM, 2018, pp. 224–234. [Online]. Available: <https://doi.org/10.1145/3236024.3236084>
- [35] S. Roy, J. Hsu, and A. Albarghouthi, "Learning differentially private mechanisms," in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 852–865. [Online]. Available: <https://doi.org/10.1109/SP40001.2021.00060>
- [36] Y. Wang, Z. Ding, Y. Xiao, D. Kifer, and D. Zhang, "DPGen: Automated program synthesis for differential privacy," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 393–411. [Online]. Available: <https://doi.org/10.1145/3460120.3484781>
- [37] S. Lahiri and S. Roy, "Almost correct invariants: Synthesizing inductive invariants by fuzzing proofs," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 352–364. [Online]. Available: <https://doi.org/10.1145/3533767.3534381>
- [38] P. Golia, S. Roy, and K. S. Meel, "Manthan: A data-driven approach for boolean function synthesis," in *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020. Proceedings, Part II*. Berlin, Heidelberg: Springer-Verlag, 2020, p. 611–633. [Online]. Available: [https://doi.org/10.1007/978-3-030-53291-8\\_31](https://doi.org/10.1007/978-3-030-53291-8_31)
- [39] P. Golia, S. Roy, and Kuldeep S. Meel, "Program synthesis as

- dependency quantified formula modulo theory,” in *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence, IJCAI 2021, Virtual Event / Montreal, Canada, 19-27 August 2021*, Z. Zhou, Ed. ijcai.org, 2021, pp. 1894–1900. [Online]. Available: <https://doi.org/10.24963/ijcai.2021/261>
- [40] P. Golia, F. Slivovsky, S. Roy, and K. S. Meel, “Engineering an efficient boolean functional synthesis engine,” in *IEEE/ACM International Conference On Computer Aided Design, ICCAD 2021, Munich, Germany, November 1-4, 2021*. IEEE, 2021, pp. 1–9. [Online]. Available: <https://doi.org/10.1109/ICCAD51958.2021.9643583>
- [41] A. Verma, P. K. Kalita, A. Pandey, and S. Roy, “Interactive debugging of concurrent programs under relaxed memory models,” in *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, ser. CGO 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 68–80. [Online]. Available: <https://doi.org/10.1145/3368826.3377910>
- [42] G. Takhar, R. Karri, C. Pilato, and S. Roy, “HOLL: Program synthesis for higher order logic locking,” in *Tools and Algorithms for the Construction and Analysis of Systems - 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2022, Munich, Germany, April 2-7, 2022, Proceedings, Part I*, ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13243. Springer, 2022, pp. 3–24. [Online]. Available: [https://doi.org/10.1007/978-3-030-99524-9\\_1](https://doi.org/10.1007/978-3-030-99524-9_1)
- [43] A. Leung, J. Sarracino, and S. Lerner, “Interactive parser synthesis by example,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15. New York, NY, USA: Association for Computing Machinery, 2015, p. 565–574. [Online]. Available: <https://doi.org/10.1145/2737924.2738002>
- [44] D. Singal, P. Agarwal, S. Jhunjhunwala, and S. Roy, “Parse condition: Symbolic encoding of LL(1) parsing,” in *LPAR-22. 22nd International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, ser. EPiC Series in Computing, G. Barthe, G. Sutcliffe, and M. Veanes, Eds., vol. 57. EasyChair, 2018, pp. 637–655. [Online]. Available: <https://easychair.org/publications/paper/DtjZ>
- [45] S. Roy and Y. N. Srikant, “Profiling k-iteration paths: A generalization of the Ball-Larus profiling algorithm,” in *Proceedings of the CGO 2009, The Seventh International Symposium on Code Generation and Optimization, Seattle, Washington, USA, March 22-25, 2009*. IEEE Computer Society, 2009, pp. 70–80. [Online]. Available: <https://doi.org/10.1109/CGO.2009.11>
- [46] R. Chouhan, S. Roy, and S. Baswana, “Pertinent path profiling: Tracking interactions among relevant statements,” in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2013, Shenzhen, China, February 23-27, 2013*. IEEE Computer Society, 2013, pp. 16:1–16:12. [Online]. Available: <https://doi.org/10.1109/CGO.2013.6494983>
- [47] G. Kumar and S. Roy, “Online identification of frequently executed acyclic paths by leveraging data stream algorithms,” in *Proceedings of the 28th Annual ACM Symposium on Applied Computing, SAC ’13, Coimbra, Portugal, March 18-22, 2013*, S. Y. Shin and J. C. Maldonado, Eds. ACM, 2013, pp. 1694–1695. [Online]. Available: <https://doi.org/10.1145/2480362.2480680>
- [48] P. Chatterjee, A. Chatterjee, J. Campos, R. Abreu, and S. Roy, “Diagnosing software faults using multiverse analysis,” in *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, C. Bessiere, Ed. ijcai.org, 2020, pp. 1629–1635. [Online]. Available: <https://doi.org/10.24963/ijcai.2020/226>
- [49] V. Modi, S. Roy, and S. K. Aggarwal, “Exploring program phases for statistical bug localization,” in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, PASTE ’13, Seattle, WA, USA, June 20, 2013*, S. N. Freund and C. S. Pasareanu, Eds. ACM, 2013, pp. 33–40. [Online]. Available: <https://doi.org/10.1145/2462029.2462034>