Macho: Writing Programs with Natural Language and Examples

Anthony Cozzie
University of Illinois at Urbana-Champaign
acoz@acoz.net

Samuel T. King
University of Illinois at Urbana-Champaign
kingst@illinois.edu

***Abstract*—Current natural language programming systems avoid the difficulties of dealing with abstract and ambiguous concepts by restricting the input words to those comparable to a normal high-level programming language. Our system, Macho can write programs from significantly more abstract language by asking the programmer to provide a unit test: one or more examples of correct input and output. This may seem unnecessarily complicated, but we show that natural language and examples have a surprising synergy both in constraining the ambiguity of the specification and in generating correct solutions.**

I. Introduction

Writing computer programs in natural language is one of the most obvious yet frustrating tasks in computer science. Such a system sounds incredibly appealing: programming languages are accessible to a small fraction of trained programmers, while natural language is accessible to everyone. In practice things are different. Consider Pegasus [1] in 2006 writing insertion sort:

```
Take the array [3, 5, 7, 4, 6, 2, 1].
Print "Before: " and print the array.

Count from one up to the size of the array:
  Go over the array from the beginning
  to the end minus the counter:
    If the current element is bigger than
    the following element then exchange
    the current element with the following
    element.

Print "After: " and print the array.
```

Although this example is impressively long and complicated, it's very similar to a formal language. It's just as low level, requires just as much precision from the programmer, and is just as hard to read and understand. Ideally, we would like to raise the level of abstraction, but with increased abstraction comes increased ambiguity. If a natural language programming system sees "Mail this letter to the members of the campaign committee", what font should it use? Fedex or first class mail? This causes problems both for the system, which must make choices, and for the programmer, who does not know what choices were made [2].

All attempts at natural language programming [1], [3], [4], [5] that we know of work by restricting the user to a small set of low-level, precise English. This allows their systems to produce working code, but, like the previous example, the resulting natural language is cumbersome, and everyone still uses formal programming languages with syntax characters.

Our system, Macho, can write programs from natural language which is significantly more abstract, like

```
Search a file for a pattern.
```

or

```
Print the files in a directory.
```

or

```
Print a file. Number non-blank lines.
```

Although these examples are simpler and smaller, they are also more challenging to synthesize, because ambiguity is already creeping into the picture. Should 'search' find a regular expression or a literal string? Does 'print' refer to the name or the contents of the directory entries? And what Java libraries will do all of this?

Macho resolves this ambiguity by asking the programmer to provide a small unit test in addition to the natural language: one or more correct input/output pairs. Using two different inputs seems to create unnecessary problems, but in fact the key insight behind Macho is the considerable synergy between the natural language and examples. Natural language provides broad but incomplete information over the entire input space, while examples provide complete information over a small fraction of the input space. The set of programs that satisfy both is much smaller than those that satisfy either input independently. The additional constraints provided by the examples allow Macho to write programs from more abstract natural language without suffering a correspondingly large rise in ambiguity. This actually is not that surprising, because it works for people: textbooks, research papers, and presentations are loaded with examples which help make concepts concrete after high-level discussions.

Natural language and examples also work together during synthesis. Understanding abstract natural language is hard because while each piece may make sense individually, the program must assemble the fragments into a cohesive whole. But any candidate program that passes a nontrivial unit test is likely to be quite reasonable. Programming from examples [6], [7], [8], [9] is hard because there are many programs that will pass the unit test, with the most obvious being a case structure over the inputs, and the search in the space of candidate programs rapidly explodes. But the natural language provides

a perfect set of heuristics to generate reasonable candidate programs.

Macho writes code by performing a breadth-first search in the space of possible programs. Candidate solutions are generated by repeated application of simple handwritten patterns, like 'implement this operation as a Java function' or 'map this noun to that variable'. Their fitness is estimated by a Bayesian probabilistic model that was trained on a database of open source Java code. The best candidates are then executed on each example input in the unit test and their output compared with the reference.

Because it is built around probabilistic machine learning algorithms, Macho has no well-defined input language, which makes it hard to develop a good intuition about the kinds of programs Macho can write. In our evaluation section we discuss example natural language that Macho can and cannot convert to code and why. Three necessary but not sufficient conditions are: the input text must refer to methods and objects in ways similar to the programmers in Macho's database, the input text must be structurally similar to the desired program code, and the resulting code must be composed of existing Java library calls. This is still a fairly low level of abstraction, but Macho was able to write 55 out of 69 of our programs correctly, with solution times between 2 and 20 minutes. We uploaded Macho's full output to http://www.acoz.net/macho_results/. This includes descriptions of every pattern, the patterns Macho used to synthesize every program, and graphs of every candidate solution including details of the probabilistic model.

Macho cannot eliminate all ambiguity. Programs written by Macho *may not be correct*, even if they pass the unit test, which is why we view Macho as a tool to assist programmers, not replace them. Most industrial programmers must write comments and a unit test for each function anyway, so Macho is simply an opportunity to potentially get some code for free.

Our contributions:

- The first system to use dual inputs of natural language and examples to write programs.
- Substantial improvements in the level of abstraction relative to natural language programming systems, and in complexity and scope relative to example-based programming systems.

II. Design and Implementation

A. Overview

Most natural language programming systems rely on grammar as a crude replacement for syntax. While a compiler parses `print(a)` into a method call and a variable use, a natural language programming system will break down `Print the array` into an imperative verb, article, and noun. It will then assign an operation or function to each verb and a variable to each noun. When the sentence structure is simple and the words are precise, this can be done with a small number of deterministic rules.

When the level of abstraction goes up, ambiguity creeps in. Consider the specification `Print the files in the directory`. There are two reasonable parses: `print(files,directory)`, meaning print the files into the directory, or `print(files(directory))`, meaning print the files that are already in the directory. There are multiple legitimate functions that could satisfy `print`: print the name of the file, print the contents of the file, and variants with different formatting.

Macho works like a nondeterministic version of a traditional natural language programming system: it generates multiple candidate solutions for the natural language input and checks them against the unit test at the end to see if one passes. Our first version of Macho [10] was particularly simple: any time it had a choice of different parses or assignments it tried everything. This approach broke down quickly for larger inputs. Adding a probabilistic model not only allows Macho to handle larger problems, but also a larger number of rules.

B. Probabilistic Model

To rank possible implementations of the input text T, Macho assigns each candidate program P some probability $p(P|T)$. Both the input text and the candidate program are too complicated to model directly, and in any case we have no database of training data. Instead, Macho breaks the text into pieces grammatically, maps each piece to part of the candidate program, and estimates the likelihood of this correspondence.

Intuitively, this is very similar to the traditional approach described in the previous section; nouns are mapped to variables and verbs are mapped to functions. We were able to train this model on a large database of open source Java code using only programmer labels. For example, the probability that a variable of type `java.io.File` represents a `file` noun is estimated from the number of times `java.io.File` variables in the database were named 'file'.

We can now define some terminology:

- T is the input text
- c is a concept, like 'file', which we expect to map to a variable in the candidate program. $\boldsymbol{C}$ is the set of concepts in the skeleton program.
- e is an event, like 'print', which we expect to map to a function in the candidate program. $\boldsymbol{E}$ is the set of events in the skeleton program.
- r is a relation, like 'in' or 'of'. $\boldsymbol{R}$ is the set of events in the skeleton program
- u is a property, like 'large'. $\boldsymbol{U}$ is the set of properties in the skeleton program.
- $\mathbf{S}$ is the program skeleton, or all concepts, relations, events, or properties.
- v is a variable in the program, and $\boldsymbol{V}$ is the set of all variables
- f is a function call in the program, and $\boldsymbol{F}$ is the set of all function calls
- P is a candidate program, or all variables and functions; $\boldsymbol{P}$ the entire set of candidates

Macho uses a generative Naive Bayes model, which breaks down as follows:

$$p(P|T) = p(P|\mathbf{S})p(\mathbf{S}|T) \quad (1)$$

$p(\mathbf{S}|T)$ is the probability of a given parse tree of the sentence returned by the parser. Next we apply Bayes' Rule:

$$p(P|\mathbf{S}) = \frac{p(\mathbf{S}|P)|p(P)}{p(\mathbf{S})} \quad (2)$$

and break $\mathbf{S}$ and P into their components:

$$p(P|\mathbf{S}) = \frac{p(\boldsymbol{C}|P)p(\boldsymbol{E}|P)p(\boldsymbol{R}|P)p(\boldsymbol{U}|P)p(\boldsymbol{F})p(\boldsymbol{V})}{\sum_{P\in\boldsymbol{P}} p(\mathbf{S}|P)p(P)} \quad (3)$$

The denominator is an intractable sum over all possible programs, but it does not affect the most likely solution. Our model assumes independence for simplicity, so $p(\boldsymbol{C}|P) = \prod_{c\in\boldsymbol{C}} p(c|P)$ and so on. Each component was trained from our database of open source Java code.

1) Function Prior: $p(f)$: The simplest function prior assumes that all function calls are independent. This can be directly measured from the database. For example, `java.io.File.list()` was seen 273 times out of a total of 4.5 million total calls, giving it a prior of .00006. Of course, these functions are not independent at all. We tried several methods for predicting the type from the surrounding code [11], [12] but we had the best results with something extremely simple.

Macho defines $p(f)$ as the number of times f was seen in the database divided by the number of call sites where all of f's parameters were in scope. If there is no `java.io.File` variable in scope, it would be impossible to call `list()`. This reduces the number of possible call sites from 4.5 million to 225 thousand, giving `java.io.File.list()` a prior of .0012. This change nicely models the object oriented structure of Java: it's very unlikely to call `XMLParser.getToken()`, unless there is an `XMLParser` object in scope which makes it quite likely.

2) Variable Prior: $p(v)$: Macho models the types of variables, but not their names. Input variables are assumed to be independent of each other and are assigned $p(v)$ equal to the number of variables of type v divided by the total number of variables in the database. Intermediate variables have their type constrained by the function generating their value, and are assigned a prior of 1.0, since any other type would break the program (we ignored casts and inheritance for simplicity).

3) Concept Posterior: $p(c|P)$: Each concept (noun) is aligned with a single variable in the program, and the concept posterior measures the probability that this variable would be so labeled in the database. For example, out of the 273 times `java.io.File.list()` was seen in the database, the input variable was named 'dir' (or a variant, like 'outputDir') 66 times, 'f' 41 times, 'file' 38 times, and so on. Macho measures the probability of a given name $p(c|f,i)$ for every input and output of every function in the database; in this case $p(dir|java.io.File.list(), input_0) = 66/273 = 0.24)$. We also measured $p(c|t)$, the probability that any variable of type t would be named c. This is especially useful for functions where Macho does not have enough training data to accurately measure $p(c|f,i)$. Combining this and all functions that read from or write to the variable:

$$p(c|P) = p(c|t) \prod_{f,i\in inputs} p(c|f,i) \prod_{f,o\in outputs} p(c|f,o) \quad (4)$$

4) Event Posterior: $p(e|P)$: The event posterior works exactly like the concept posterior, except that the function has only one name. This caused us surprisingly few problems, but to handle those cases we added a synonym file. For example 'display' is given as a synonym for print, rather than a pdf of {(Display, 1.0)}, it becomes {(Display, 0.5), (Print, 0.5)}. Our synonym file has approximately 10 entries (the main one being show/print/display) and is included in our full results online.

5) Property Posterior: $p(u|P)$: Properties are treated as either concepts or events depending on what they are mapped to. For example, the property 'source' from the text 'source file' will be treated as a concept while the property 'blank' from 'blank lines' will be treated as a function which selects a subset of lines. Macho will try both and pick whichever scores higher.

6) Relation Posterior: $p(r|P)$: All relations are given a uniform posterior when mapped to a function. Macho also assigns uniform probabilities to the program skeleton, including control flow, i.e. the programs $f(g(x))$ and $g(f(x))$ have the same probability as long as they both type check.

7) Incomplete candidates: $\hat{p}(\boldsymbol{S}|P)$: Most of the solutions Macho evaluates are incomplete, i.e. not all of the skeleton has been mapped to a concrete Java function or type. This estimate of the real probability is quite important because it controls which nodes are expanded and which are discarded. $\hat{p}(v)$, $\hat{p}(r|P)$, $\hat{p}(u|P)$, and $\hat{p}(e|P)$ are small constant values. However, this was insufficient for $\hat{p}(c|P)$; when approximated by a small constant Macho tended to map all of its unassigned concepts to the same untyped variable, even when this did not make sense (the same variable would be labeled 'file' and 'line' for example). Therefore Macho estimates a weighted average over possible types, where $\mathbf{C}_o$ is the set of overlapping concepts mapped to that variable:

$$\hat{p}(\mathbf{C}_o) = \sum_{t\in types} p(\mathbf{C}_o|t)\left(\frac{p(\mathbf{C}_o|t)}{\sum_{t\in types} p(\mathbf{C}_o|t)}\right) \quad (5)$$

and

$$p(\mathbf{C}_o|t) = \prod_{c\in\mathbf{C}_o} p(c|t) \quad (6)$$

The same averaging is applied to $\hat{p}(c|f,i)$. Macho also estimates $\hat{p}(F)$. When at least one of the inputs to a function is known, but that function itself is not assigned, then the prior is the weighted average of all functions which could be selected without breaking the type system, i.e. they have the correct number of parameters and there is an assignment from all of the known inputs/outputs to those of the function.

C. Parsing

Macho's first task is to parse the grammatical structure of the input text, which it does using the Stanford Parser [13]. The nodes of the resulting parse tree are either labeled words, e.g. `NN:file`, meaning that 'file' is a noun, or a phrase node which reflects the structure of the sentence, e.g. `NP(DT:a, NN:file)` which reflects the relation between the two words. For a full list of tags see [14]. We were generally very pleased with the Stanford parser, but unfortunately it was trained primarily on a "normal" corpus of newspaper articles which used substantially different vocabulary, causing numerous errors. For example, 'file' is usually a verb, like "the SEC filed charges against Enron today." and print is often a noun, like, "Their foul prints will not soon be cleansed from the financial system.". Macho usually avoids this problem by requesting the best 10 parse trees, which almost always contain the correct solution.

D. Graph Pattern Matching

Macho represents its candidate solutions as graphs, which are flexible enough to represent parse trees, program skeletons, pseudocode, and actual Java code with a single data structure. There are nine types of graph nodes which fall into three groups:

- Parser nodes, which represent the parse trees returned by the Stanford Parser
- Concepts, relations, events, and properties, which represent skeleton programs.
- Variables, functions, constants, and blocks, which represent pseudocode and Java code

The main edges are dataflow and assignments between the skeleton and concrete programs.

Macho's candidate programs start their lives as parse trees and are gradually refined by the repeated application of patterns into pseudocode and finally compilable Java. A pattern is composed of a set of trigger nodes, which must each match a node in the candidate program, and a function which emits one or more new candidate solutions in the event of a match. These patterns are flexible enough to store all of Macho's manually defined knowledge and reusable enough that there are only about 100, which roughly break down as:

- 50% parser conversion patterns, which convert parser nodes to skeletons, like "replace a noun with a concept".
- 10% instantiation patterns, which match/convert skeleton programs to pseudocode, like "match concept x with variable y".
- 10% looping patterns, which handle Macho's iterate control flow block and replace the more complicated Java loops.
- 10% assembly patterns, which convert Macho's graph language to real Java code.
- 10% specialized knowledge, like "get the current working directory by calling `System.getProperty("user.dir")`"
- 5% macros, which perform more complicated tasks, like handling "ignore" by removing a subset of items.
- 5% hacks, many of which deal with the phrase "in sorted order", which the Stanford Parser did not handle well.
- 1 database pattern, which tries to match inferred function calls to Java libraries, by estimating how their selection would change the probabilistic model.

The complete list of patterns is available online as part of our complete output.

E. Custom Libraries

The Java standard library is missing some functions that would be very useful to Macho, like `ReadAllLines()` or `DownloadFile()`, so we were forced to write our own. Without any training data, we had guess at prior values and input/output pdfs by hand. We did our best to make these resemble the normal functions by including large noise terms in the pdfs.

F. Testing and Formatting

After all of the candidates have been generated, Macho runs the most probable 200 against the unit test. The candidates that pass every test are ranked by the probabilistic model, and the most likely one is returned to the programmer. Macho does not attempt to debug candidates that pass some fraction of the unit tests.

However, Macho does try multiple different format strings. Our custom `Print` function prints the data in a serializeable form. After execution Macho reads this list of print statements back in, and tries to find a set of format statements which would generate the output stream. For example, if the output is from `cat -b`:

```
1  To be, or not to be: that is the question:
2  Whether 'tis nobler in the mind to suffer
3  The slings and arrows of outrageous fortune,
```

and Macho's output looks like:

```
(PRN1 1 "To be, or not to be: that is the question:")
(PRN1 2 "Whether 'tis nobler in the mind to suffer")
(PRN1 3 "The slings and arrows of outrageous fortune,")
```

then Macho will encounter a new print statement (PRN1) and use the output as a template to generate candidate format strings. PRN1 could be `"  %d %s"` or `"%6d %s"` or `"%6d%45s"` or even `"  1 %s"` by ignoring the first argument; all of these will generate text that matches the first part of the output exactly. The next time Macho sees PRN1 (the next line, in this case) it will use the same format string, and if the output does not match, the candidate format string will be discarded. If the entire output matches, each print statement in the program is replaced by a `System.format` call with the corresponding format string.

G. An Example

Figure 1 shows four candidate solutions Macho generated for "Print the names of files in a directory". Macho can do much more complicated programs, but the graphs get large very quickly. After running the Stanford Parser, Macho

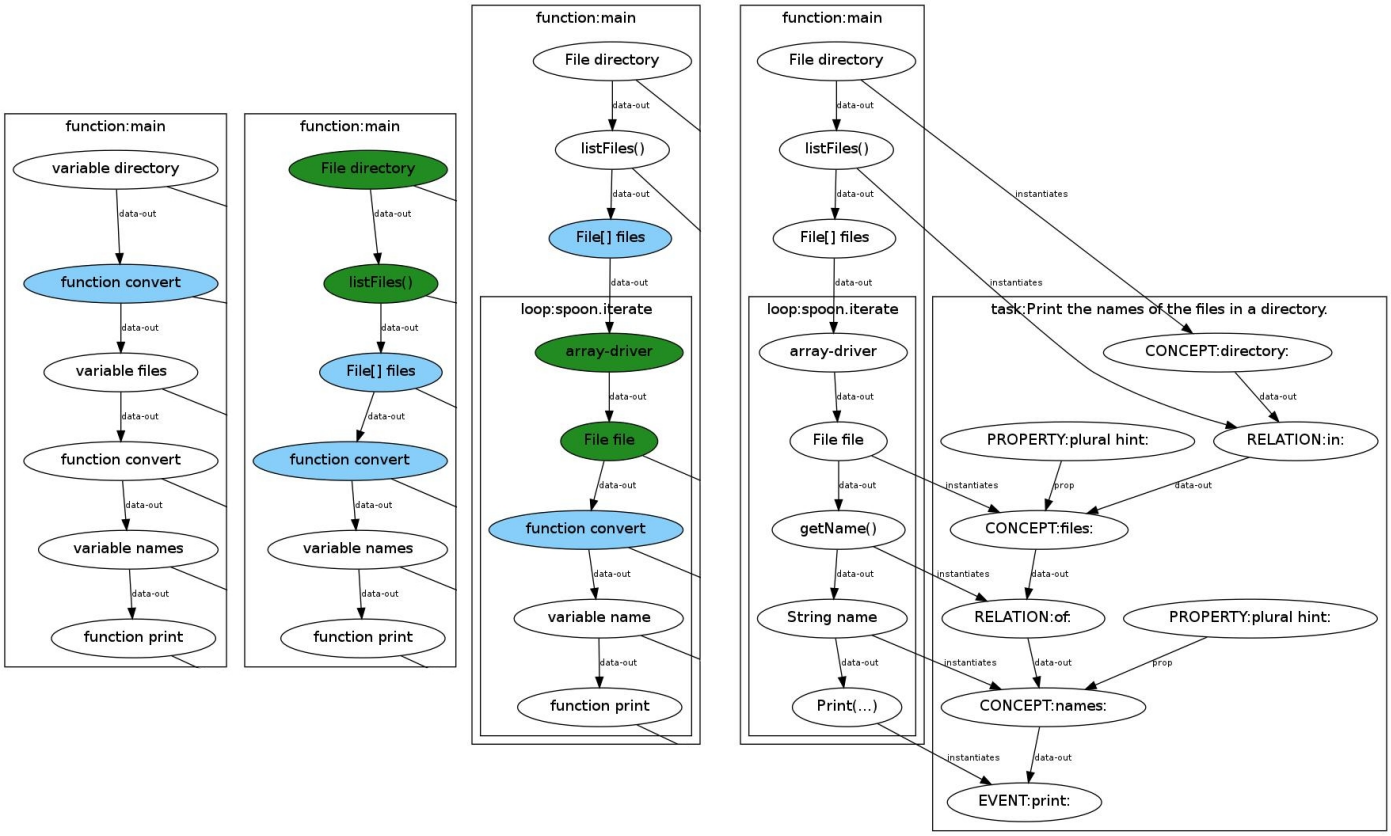

Fig. 1. Four of Macho's candidate solutions for "Print the files in a directory". The easiest way to understand this diagram is to start at the right, with the block labeled "Task:Print the names of the files in a directory". This contains the skeleton program generated from the parse tree; the verb 'print' has become an event, the nouns 'names', 'files', and 'directory' have become concepts, and the prepositions 'in' and 'of' have become relations. The box immediately to the left (second from the right) is the best solution. Each event has been mapped to a function and each noun to a variable. The three boxes on the left show intermediate solutions, starting with an empty program that matches the structure of the skeleton. All of them include a copy of the skeleton program (you can see the lines going off to the right for each node) that has been removed to save space. The second candidate is generated from the first one by the database pattern selecting the `java.io.File.listFiles()` function for the `directory` to `files` conversion, and the third candidate is generated from the second by Macho trying a for loop over the resulting array. For more details see section II-G.

gradually converts its parse tree to the skeleton program shown on the far right.

Once the parse tree has been completely converted to features, Macho attempts to generate programs which fit them; the three boxes on the left side of the figure show three such candidate programs.The first candidate is pseudocode representing a 1:1 feature-program mapping, which in this case is correct because our spec has only one sentence. The variable names reflect the concepts (and therefore the nouns) and the function names reflect the events (and therefore the verbs).

Among the patterns that matches the first candidate is the database pattern, which matches any function which is still pseudocode. It searches Macho's list of Java API calls for functions that would be a good fit based on the names and types of the surrounding variables and the label the programmer assigned. We will follow the correct selection `File.listFiles()`, but Macho tries among others `File.list()` and `JFileChooser.getSelectedFiles()`. It assigns the Java code to the pseudocode function and types to the input and output variables.

The third candidate shows Macho attempting to convert a File array to names. The database pattern will also trigger here, but there are not many functions which operate on an array of files. So we added a simple pattern which triggers on any function which operates on an array: try inserting a loop around the function, in this case a simple for each loop (spoon.iterate) over the elements. The array driver operation will return each successive element of the array. At this point the database pattern will again trigger and this time return several candidates for the new conversion, including `getName()`, `getPath()`, and `getAbsolutePath()`, eventually leading to the final program shown on the right side of the figure.

III. Evaluation

Objectively evaluating Macho is very difficult, because there is no standard benchmark suite that would allow us to compare its results against other systems. Therefore we built our own suite, and we chose to focus on generating standard Unix command-line utilities. Their outputs are much easier to check than GUI programs, but more importantly they provide a precise specification that prevents us from declaring the first

Task	Test	Text	Time	Pass	Best Prior	Rank	Result
CAT-1	cat	Print a file.	3:42	6	3.60e-6	1	success
CAT-10	catreve	Print a file. Show $ät the beginning of each line.	1:48	2	3.60e-7	1	success
CAT-11	catreve	Print a file.	3:11	8	3.60e-6	1	success
CAT-12	cat -T	Print a file. Display TAB characters as Î.		0			failure
CAT-2	cat	Read a file.	14:23	3	3.60e-6	1	success
CAT-3	cat	Display the contents of a file.	3:00	3	3.60e-6	1	success
CAT-4	cat	Print the lines of a file.	3:04	1	2.40e-6	5	success
CAT-5	cat -b	Print a file. Number non-blank lines.	5:57	2	4.06e-13	18	success
CAT-6	cat -n	Print the lines of a file. Show line numbers.	9:49	4	3.60e-8	11	success
CAT-7	nonblanklines	Print the number of non-blank lines in a file.	6:45	18	4.06e-12	1	success
CAT-8	whitespacechars	Count the whitespace characters in a file.	4:13	15	5.33e-12	2	success
CAT-9	cat -E	Print a file. Show $ät the end of each line.		0			failure
CP-1	cp	Copy src file to dst file	6:00	18	1.63e-8	2	success
CP-2	cp	Copy a file to a file.	5:41	3	1.63e-8	2	success
CP-3	cp	Copy a file.	6:39	1	1.63e-8	2	success
CP-4	cp	Duplicate a file.	6:30	1	1.63e-8	2	success
CP-5	cp -i	Copy a file. Prompt before overwrite.		0			failure
CP-6	cp -u	Copy a file. Only copy when the source file is newer.		0			failure
GREP-1	grep	Print the lines in a file that match a pattern.	2:40	6	3.27e-10	6	success
GREP-2	grep -i	Find a pattern in the lines of a file.	2:58	4	3.27e-13	6	success
GREP-3	grep	Search a file for a pattern.	3:25	7	3.27e-10	38	failure
GREP-4	grep -i	Find a pattern in the lines of a file. Ignore case.	3:14	4	3.27e-13	10	success
GREP-4	grep -i	Find a pattern in the lines of a file. Ignore case.	3:01	4	3.27e-13	10	success
HEAD-1	head	Print the first 10 lines of a file.	2:13	4	5.58e-13	1	success
HEAD-2	headsort	Print the first 10 lines of a file in sorted order.	3:12	2	1.12e-15	1	success
HEAD-3	sorthead	Sort a file. Print the first 10 lines.	7:56	2	1.12e-15	8	success
HEAD-4	head -n 7	Print the first 7 lines of a file.	1:26	6	5.58e-13	1	success
LS-1	ls	Print the names of the files in a directory.	10:39	5	1.20e-11	20	success
LS-2	ls	Display the entries in a folder.	2:10	2	1.20e-11	13	success
LS-3	ls -S	Print the files in a directory. Sort by size.	4:16	2	1.73e-16	16	success
LS-4	ls -Sr	Print the files in a directory. Sort by size.	3:24	2	8.65e-14	2	success
LS-5	ls -X	Print the files in a directory. Sort by extension. Ignore hidden files.	12:11	6	1.50e-13	2	success
LS-6	lsmed	Print all media files in a directory.	4:09	17	1.20e-11	2	success
LS-7	lsnamessizes	Print the names and sizes of files in a directory. Scale sizes by 1024.	3:05	1	3.20e-16	51	success
LS-8	ls -L	Print the files in a directory. Show more information.		0			failure
LS-9	lscount	Print the number of things in a directory.	3:08	8	3.47e-9	1	success
MATH-1	add	Add x and y		0			failure
MATH-2	add	Add value and value	2:09	24	1.27e-11	3	success
MATH-3	mult	Multiply value and value.	2:04	26	1.40e-4	24	failure
MATH-4	mult	Multiply bd and bd.	1:31	44	7.30e-13	14	success
MATH-5	mult	Multiply big decimal and big decimal.	1:25	8	7.30e-13	2	success
MATH-6	1024mult	Multiply value and value. Scale result by 1024	4:30	4	3.91e-13	70	success
MATH-7	1024mult	Multiply bd and bd. Divide by 1024.	2:57	4	3.51e-15	31	success
MATH-8	1024mult	Multiply big decimal and big decimal. Divide result by 1024.	2:23	27	4.56e-10	8	failure
PWD-1	pwd	Print the name of the current working directory.	2:05	9	1.78e-5	1	success
PWD-2	pwd	Print the user directory.	1:51	10	1.78e-4	1	success
PWD-3	pwd	Print the current directory.	1:33	10	1.78e-4	1	success
PWD-4	pwd	Print the working directory.	1:31	8	1.78e-4	1	success
PWD-5	pwd	Show the current working directory.	1:12	1	1.78e-4	1	success
PWD-6	pwd -P	Print name of the current directory. Avoid all symlinks.		0			failure
SORT-1	sort	Sort a file.	4:26	2	7.19e-9	6	success
SORT-2	sort	Sort the lines of a file.	3:50	1	7.19e-9	4	success
SORT-3	sort -r	Print the lines of a file in reverse sorted order.	11:34	1	1.44e-11	2	success
SORT-4	sort -n	Print the lines of a file in sorted order. Compare by numerical value.		0			failure
SORT-5	sort	Sort a file by line.	3:33	1	1.80e-8	2	success
SORT-6	sort	Sort the contents of a file.	4:21	1	7.19e-9	13	success
SORT-7	sort -R	Sort a file. Sort by random hash of keys.		0			failure
TAIL-1	tail	Print the last 10 lines of a file.	2:05	2	5.46e-14	1	success
TAIL-2	tailrev	Print the last 10 lines of a file. Reverse the lines.		0			failure
TAIL-3	tailtac	Print the last 10 lines of a file. Reverse the lines.	1:54	1	1.09e-16	1	success
TAIL-4	tailremblank	Print the last 10 lines of a file. Remove blank lines.	12:18	2	3.64e-21	1	success
UNIQ-1	uniq	Print a file. Filter adjacent matching lines.		0			failure
WGET-1	wget -q	Download a file.	6:03	1	3.13e-7	1	success
WGET-2	wget -q	Download remote file.	4:48	2	3.13e-7	1	success
WGET-3	wget -q	Open connection. Download file.	8:22	3	3.13e-7	1	success
WGET-4	wget -q	Open network connection. Download file.	10:34	1	3.13e-7	1	success
ZIP-1	zipinfo -1	Print the entries in a zipfile.	3:07	6	2.46e-9	1	success
ZIP-2	zipinfo -1	Show the stuff in a zipfile.	1:57	4	2.46e-9	1	success
ZIP-3	zipinfo -1	Print the files in a zipfile.	2:41	4	2.46e-9	8	success

Fig. 2. Macho's results for generating various programs. 'Pass' is the number of solutions that pass the unit test. 'Best Prior' is the prior probability of the solution, and a rough approximation of how long it would take Macho to generate the solution without the natural language. 'Rank' is the number of tries Macho needed to pass the test.

"reasonable" output to be correct. Originally we wanted to use the language directly from the man pages (hence the name Macho) but this turned out to be very difficult. Instead we picked a set of natural language/example inputs that are right on the border of Macho's capabilities.

A. Results

Figure 2 shows our results on a large selection of different natural language inputs. Any solution that passed the test was byte identical; it was judged correct or not after an additional manual inspection.

1) Coreutils: pwd: `pwd` doesn't pose many issues for Macho, with the exception being "Avoid all symlinks". Avoid is a difficult general-purpose word, and symlinks aren't well supported in Java.

2) Coreutils: cat: Macho is only one character away from `cat -E`; its best solution appends "$" after all lines, even those that don't end with a linefeed. In contrast, prepending "$" at the beginning of the line is easy, even without specific natural language.

3) Coreutils: count: This isn't a real core utility, but it shows off Macho's adjective filtering mechanism, which uses functions to select a subset of objects. This is very important for natural language code, because without formal structure much of the input text is often devoted to when and where the new code applies.

4) Coreutils: sort: `Sort` is basically `cat` with an extra function at the end. Macho infers most of the print statements here; any time the user requests a task which is implemented by a functional library, Macho tries printing the result rather than throwing it away.`Sort` on our linux machine does not put its arguments in ASCII order by default, so whenever Macho sorts an array of strings it tries both ASCII and dictionary order, which ignores whitespace and case.

5) Coreutils: grep: `Grep` is a great example of Macho's need for specialized knowledge - there is no real alternative for handling "Ignore case" other than knowing that it is an option to the `java.util.regex.Pattern` function. Macho is only stymied by another tricky word, context. `GREP-3` is almost correct, but Macho tries case insensitive regexes first and the unit test contains no upper case characters.

6) Coreutils: ls: `Ls` was our original example and fairly straightforward for Macho. `Ls -l`, "Print more information" is both extremely interesting and extremely hard.

7) Coreutils: wget and cp: Default `wget` and `cp` are both very simple programs (other than mimicking the extremely complicated output of `wget` under default options, which Macho does not generate) and were handled easily by Macho. `cp -i`, "Prompt before overwrite" looks very hard but is probably right on the border of Macho's capabilities.

8) Coreutils: head and tail: `head` and `tail` were handled by a pattern which takes the "first" and "last" part of an array, although Macho takes a subsequence rather than cutting the iteration short for `head`.

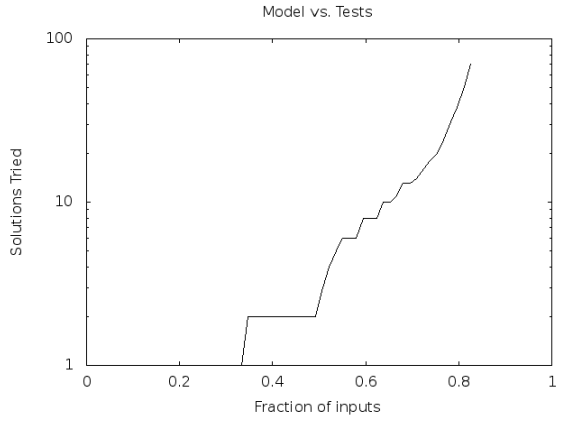

Fig. 3. The rank of the first solution to pass the unit test according to the probabilistic model. A rank of one implies that Macho can solve the problem without the unit test, while a larger rank is the number of solutions Macho had to try before finding one that would pass.

9) Custom: Math: The math programs are the easiest ones and the most similar to a traditional system. Interestingly Macho does not get 'Add x and y' because of a parser error. Macho also failed on `MATH-3` and `MATH-8` because we used a unit test with only one example. In both cases it generated a constant format string which passed the test.

10) Custom: Zip: `Zip` is almost identical to `ls`, except that we had to add support for iteration over java Enumerations.

B. Only Natural Language

Figure 3 shows the CDF of the number of candidate solutions Macho tried before finding one that would pass the test. Approximately half of the programs pass on the first attempt, which means that they could be correctly synthesized from only natural language. Nevertheless, adding unit tests hugely improves Macho's accuracy (from 35% to 80%) and those improvements come disproportionately on the more abstract programs.

But even more importantly, the unit test improved Macho's estimation of its accuracy. The first (and most highly ranked according to the probabilistic model) solution to pass the unit test was correct in all but three cases which means that Macho's confidence in its own correctness improves from 35% to 95% by adding tests.

C. Only Examples

Figure 4 shows the CDF of the prior probability of the best solution. This represents a rough estimate of the inverse of the number of potential programs that Macho would have to try without natural language to guide the search. Because Macho searches the entire Java standard library, the space of potential programs grows very quickly. Macho would require billions of guesses to synthesize an average program.

This graph is both optimistic and pessimistic. It is optimistic because Macho's prior does not include any terms regarding the shape of the program: which functions operate on which

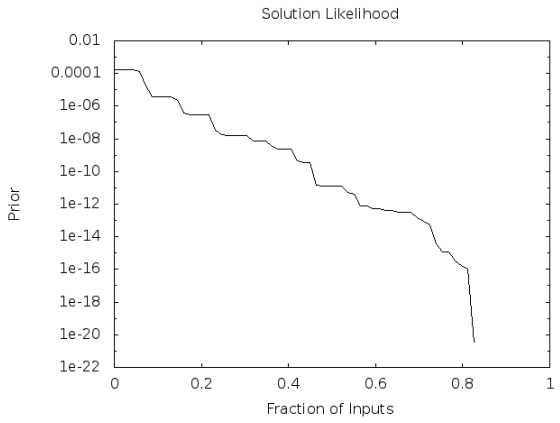

Fig. 4. Macho's prior score for the best solution that passed the test. This is a rough approximation of how long it would take Macho to find each program by brute force.

variables and which statements are inside which pieces of control flow. Any true enumeration of candidate programs would have to take this into account. It is pessimistic because it is probably possible to learn more efficient heuristics specifically for programming by example. Macho tries many candidate solutions that do not "make sense" to human programmers.

D. Macho vs abstraction

Most of the programs Macho writes are very simple and do not appear to contain much ambiguity. We would like to show you a few of the more interesting decisions that Macho must make.

1) Cat: bytes vs. lines: One of the first programs that Macho successfully wrote was cat. Its version opened the BufferedReader, read each line in a loop, and printed this line to the screen - Figure 5. This looks good, but when we went to actually test this version against the examples, we found a small but annoying bug: if the file does not end with a linefeed, an additional one will be created. Unlike the C routine `fgets()`, the `BufferedReader.readLine()` call strips trailing linefeeds, which requires the additional "\n" in the format call, which will then appear regardless of whether the last line in the file ended with a linefeed or not.

Depending on what the programmer is trying to do, this may be a convenient addition or a mission critical problem, and it is not clear that the Java version is actually wrong. Macho can avoid this problem by generating two different versions of `cat`; one reads the file by byte and the other by line. In the end we did implement our own version of `readLine` which mirrors the behavior of `fgets` and added it to the list of libraries, which is necessary for the versions of `cat` which do more than just write the file.

2) Ls: file length, "large" files: The natural language spec for `ls -S` includes "sort the files by size". Macho's first choice (by a wide margin) is the `File.length()` function, which returns the size of the file in bytes. However, it finds two other interesting solutions: one reads the file into an array of lines, and another into an array of bytes, and then both

```
public class CAT1 {  //read file by lines - wrong!
 public static void main(String[] args) {
  try  {
   File file = new File(args[0]);
   FileReader file_reader = new FileReader(file);
   BufferedReader reader =
     new BufferedReader(file_reader);
   while(true) {
    String line = reader.readLine();
    if(line == null) break;
    System.out.format("%1$s\n", line);
   }
  }
  catch (Exception e) {
   System.out.println(e.toString());
  }
 }
}

public class CAT2 { //read file by bytes - right!
 public static void main(String[] args) {
  try  {
   File file = new File(args[0]);
   FileInputStream reader =
     new FileInputStream(file);
   while(true) {
    int byteRead = reader.read();
    boolean eof = byteRead == -1;
    if(eof) break;
    System.out.format("%1$c", byteRead);
   }
  }
  catch (Exception e) {
   System.out.println(e.toString());
  }
 }
}
```

Fig. 5. Both of these programs were found by Macho during its attempts to implement cat from the text "Print a file." The first program has a very subtle bug uncovered by the examples - see III-D1 for details.

use the `array.length` variable to measure the size. We don't think the number of lines in a file is a better metric for size than byte count, but the point is that abstract qualities like size often have multiple definitions, like height, width, volume, weight, or bytes count. Macho can tolerate multiple definitions for these qualities by trying multiple solutions.

3) Formatting: Matching the output of the coreutils exactly requires a lot of attention to formatting. Consider `cat -b`, which prints the lines of a file and numbers only the blank lines. Macho must synthesize the format string "%6d ", including the trailing spaces, for the line number - nothing else will work. Macho also tries "%6d" for the line number and " %s" for the line, which moves the spaces from one format specifier to the other. Unfortunately this adds extra invisible whitespace before every empty line. This kind of formatting is extremely clumsy to describe in natural language.

E. Performance

We have been focusing on correctness rather than performance, but Macho is not that slow. Macho is not optimized at all: it is implemented in Lisp, its data structures are all lists which are traversed many times, its graph pattern matcher is extraordinarily simple and runs in exponential time, it runs on

one processor despite having a huge capacity for parallelism, many of its database lookups could be precomputed instead of cached, and it does not reuse java virtual machines across compiling and running its programs.

IV. Related work

A. Natural-Language Processing

Programming by natural-language specification is an ambitious goal. Largely, Natural-Language Processing (NLP) does not aim at this goal presently because it depends on open questions in artificial intelligence, such as commonsense reasoning [15], [16], [17], and knowledge-based natural-language understanding [18], [19], [20]. Instead, NLP works focus on fundamental semantics of natural language and computational applications.

The closest NLP works to ours, like *translation* and *question answering* [21], focus on using large corporas to yield reasonable results that are statistically reliable. Those applications rely on availability of simple solutions that do not require combination or deeper reasoning. For example, question answering looks for a sentence that includes the words that appear in the question.

Compared to NLP-based works, our work uses large corpora of possible interpretations of natural-language texts, applying combinatorial optimization to reach reasonable conclusions. In doing so, we use the type constraints available from our large repository of code segments together with available records of acceptable outputs (our unit tests). These enable combining results of natural-langauge-based segments in ways that are not possible in many other NLP applications.

B. Code snippet search

Programming languages like Java or C# are more difficult to learn than their less bulky counterparts. This is partly because of the explosion of APIs in those languages. A true Java programmer must be familiar with thousands of classes, many of which change between versions.

Many researchers have proprosed tools to make it easier to find chunks of code. Prospector [22] leverages the type system to answer questions like "How do I go from an ICompilationUnit to an IDebugWindow object". SNIFF [23] works more on the natural language using the documentation and intersecting examples to determine the really critical function calls. Other attempts [24], [25], [26] use both, as well as the context of the function around the requested snippet. While snippet search is a useful tool, especially for novice programmers, there will often not be a library call that perfectly matches the users requirements.

C. Programming by sketching

Natural language is not always the most suitable way to express a computation. Programming by sketching [27], [28] works by taking a simple, easy to understand version of a very complex computation (often the inner loop from an encryption or signal processing algorithm) and generating a faster one from a sketch, a partially specified program. Sketching tries to make hard problems simple, while Macho tries to make simple problems trivial.

D. Deductive program synthesis

An alternative to programming that is still under development is to tell the computer what you want in a formal language. For simple mathematical properties, there are mechanical rules that can transform them into programs [29]. The more complicated the system, the trickier the synthesis and the larger the specification. Amphion [30] can synthesize programs that calculate properties of solar system objects using a Fortran library and graphical input. Termite [31] generates device drivers automatically from a list of device and operating system state transitions. Like Macho, Bhansali *et. al* [32] tackle core utilities.

The problem with most of these is that because the user must still ultimately specify every detail, the formal description is not massively smaller than the code. For example, Termite's device-specific specification is approximately half the length of the corresponding Linux device driver C code, although probably somewhat less error-prone to describe. Macho allows the user to specify important details through examples while filling in the less important ones itself, thus achieving 90% of the results with 10% of the work.

V. Conclusions and Future Work

In this paper we have discussed Macho, a system that synthesizes simple Java programs from a combination of natural language and examples, and how combining natural language and examples makes it easier to generate correct solutions while also reducing the ambiguity in more abstract natural language. Macho can correctly generate 55 out of 69 test programs in 2-20 minutes each.

References

[1] R. Knöll and M. Mezini, "Pegasus: first steps toward a naturalistic programming language," in *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*. New York, NY, USA: ACM, 2006, pp. 542–559.

[2] E. W. Dijkstra, "On the foolishness of 'natural language programming'," http://userweb.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html.

[3] A. W. Biermann and B. W. Ballard, "Toward natural language computation," *Comput. Linguist.*, vol. 6, no. 2, pp. 71–86, 1980.

[4] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters, "Semantics-based composition for aspect-oriented requirements engineering," in *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*. New York, NY, USA: ACM, 2007, pp. 36–48.

[5] D. Price, E. Riloff, J. Zachary, and B. Harvey, "Naturaljava: a natural language interface for programming in java," in *IUI '00: Proceedings of the 5th international conference on Intelligent user interfaces*. New York, NY, USA: ACM, 2000, pp. 207–211.

[6] A. Cypher, "Eager: programming repetitive tasks by example," in *CHI '91: Proceedings of the SIGCHI conference on Human factors in computing systems*. New York, NY, USA: ACM, 1991, pp. 33–39.

[7] D. C. Halbert, "Programming by example," Ph.D. dissertation, University of California, Berkeley, 1984.

[8] P. Maes and R. Kozierok, "Learning interface agents," in *AAAI*, 1993, pp. 459–465.

[9] W. R. Harris and S. Gulwani, "Spreadsheet table transformations from examples," in *PLDI*, 2011, pp. 317–328.

[10] A. Cozzie, M. Finnicum, and S. T. King, "Macho: Programming with man pages," in *Proceedings of the 2011 Workshop on Hot Topics in Operating Systems (HotOS 2011)*, 2011.

[11] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, vol. 41, no. 6, pp. 391–407, 1990.

[12] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003. [Online]. Available: http://dl.acm.org/citation.cfm?id=944919.944937

[13] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proceedings of the 41st Annual Meeting on Association for Computational Linguistics - Volume 1*, ser. ACL '03. Stroudsburg, PA, USA: Association for Computational Linguistics, 2003, pp. 423–430. [Online]. Available: http://dx.doi.org/10.3115/1075096.1075150

[14] J. Pettibone, "Penn treebank tags," http://bulba.sdsu.edu/jeanette/thesis/PennTags.html.

[15] J. R. Hobbs and R. C. Moore, Eds., *Formal Theories of the Commonsense World*. Westport, CT, USA: Greenwood Publishing Group Inc., 1985.

[16] C. Matuszek, M. Witbrock, R. C. Kahlert, J. Cabral, D. Schneider, P. Shah, and D. Lenat, "Searching for common sense: Populating cyc from the web," in *In Proceedings of the Twentieth National Conference on Artificial Intelligence*, 2005, pp. 1430–1435.

[17] J. McCarthy, "Notes on formalizing context," in *IJCAI*, 1993, pp. 555–562.

[18] M. Connor, Y. Gerner, C. Fisher, and D. Roth, "Minimally supervised model of early language acquisition," in *Annual conference on computational natural language learning (CoNLL)*, 2009.

[19] J. R. Hobbs, "Deep lexical semantics," in *9th international conference on intelligence text processing and computational linguistics (CICLing-2008)*, 2009.

[20] K. Toutanova and C. D. Manning, "Enriching the knowledge sources used in a maximum entropy part-of-speech tagger," in *Joint SIGDAT conference on empirical methods in natural language processing and very large corpora (EMNLP/VLC-2000)*, 2000, pp. 63–70.

[21] R. de Salvo Braz, R. Girju, V. Punyakanok, D. Roth, and M. Sammons, "An inference model for semantic entailment in natural language," in *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI 2005)*, 2005.

[22] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *PLDI '05: Proceedings of the 2005 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2005, pp. 48–61.

[23] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for Java using free-form queries," in *FASE '09: Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 385–400.

[24] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A search engine for finding highly relevant applications," in *ICSE '10: Proceedings of the International Conference on Software Engineering 2010*, 2010.

[25] G. Little and R. C. Miller, "Keyword programming in java," in *ASE '07: Proceedings of the 22nd International Conference on Automated Software Engineering*. New York, NY, USA: ACM, 2007, pp. 84–93.

[26] N. Sahavechaphan and K. Claypool, "Xsnippet: mining for sample code," *SIGPLAN Not.*, vol. 41, no. 10, pp. 413–430, 2006.

[27] A. Solar-Lezama, R. Rabbah, R. Bodík, and K. Ebcioğlu, "Programming by sketching for bit-streaming programs," in *PLDI '05: Proceedings of the 2005 Conference on Programming Language Design and Implementation*. New York, NY, USA: ACM, 2005, pp. 281–294.

[28] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs," in *ASPLOS-XII: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2006, pp. 404–415.

[29] Z. Manna and R. Waldinger, "Fundamentals of deductive program synthesis," *IEEE Trans. Softw. Eng.*, vol. 18, no. 8, pp. 674–704, 1992.

[30] M. R. Lowry and J. V. Baalen, "Meta-amphion: Synthesis of efficient domain-specific program synthesis systems," *Autom. Softw. Eng.*, vol. 4, no. 2, pp. 199–241, 1997.

[31] L. Ryzhyk, P. Chubb, I. Kuz, E. L. Sueur, and G. Heiser, "Automatic device driver synthesis with Termite," in *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, Oct 2009.

[32] S. Bhansali and M. T. Harandi, "Synthesis of unix programs using derivational analogy," *Mach. Learn.*, vol. 10, no. 1, pp. 7–55, 1993.