Program Synthesis from Natural Language Using Recurrent Neural Networks

Xi Victoria Lin  
UW CSE  
Seattle, WA, USA  
xilin@cs.washington.edu

Chenglong Wang  
UW CSE  
Seattle, WA, USA  
clwang@cs.washington.edu

Deric Pang  
UW CSE  
Seattle, WA, USA  
dericp@cs.washington.edu

Kevin Vu  
UW CSE  
Seattle, WA, USA  
kevin.m.vu@gmail.com

Luke Zettlemoyer  
UW CSE  
Seattle, WA, USA  
lsz@cs.washington.edu

Michael D. Ernst  
UW CSE  
Seattle, WA, USA  
mernst@cs.washington.edu

ABSTRACT

Oftentimes, a programmer may have difficulty implementing a desired operation. Even when the programmer can describe her goal in English, it can be difficult to translate into code. Existing resources, such as question-and-answer websites, tabulate specific operations that someone has wanted to perform in the past, but they are not effective in generalizing to new tasks, to compound tasks that require combining previous questions, or sometimes even to variations of listed tasks.

Our goal is to make programming easier and more productive by letting programmers use their own words and concepts to express the intended operation, rather than forcing them to accommodate the machine by memorizing its grammar. We have built a system that lets a programmer describe a desired operation in natural language, then automatically translates it to a programming language for review and approval by the programmer. Our system, Tellina, does the translation using recurrent neural networks (RNNs), a state-of-the-art natural language processing technique that we augmented with slot (argument) filling and other enhancements.

We evaluated Tellina in the context of shell scripting. We trained Tellina’s RNNs on textual descriptions of file system operations and bash one-liners, scraped from the web. Although recovering completely correct commands is challenging, Tellina achieves top-3 accuracy of 80% for producing the correct command structure. In a controlled study, programmers who had access to Tellina outperformed those who did not, even when Tellina’s predictions were not completely correct, to a statistically significant degree.

1 INTRODUCTION

Even if a competent programmer knows what she wants to do and can describe it in English, it can still be difficult to write code to achieve her goal. Programmers increasingly work across libraries and programming languages, creating more complex systems than ever before, and cannot memorize every detail of all the systems that must be used.

An increasingly common practice is to seek help from websites such as Stack Overflow. Tutorial and question-answering websites are powerful resources with reams of specific examples of code snippets and explanations of their behavior. When a programmer finds her exact question has been asked before, the community-vetted answer is invariably useful. However, finding the correct answer may require the use of keywords the programmer does not know. Furthermore, sometimes outdated or incorrect answers persist even if the correct answer also appears. Finally, a programmer who wishes to do something new may waste time searching for it, and then must synthesize it on her own. Even a variation of task on the website may be difficult to find because of different constants and keywords.

Despite their limitations, tutorial and question-answering websites are valuable sources of information that tools should exploit in order to help developers solve programming tasks. We used these websites to gather training data for a novel natural language (NL) to code translation tool that allows the user to express their intent in English and automatically translates it into executable programs. Such synthesis methods have many advantages. The learned models can often generalize to new NL descriptions or synthesize novel code, and the programmer does not need to search through large websites to complete her task. However, they are also inherently error-prone. For the near future, all NL-driven synthesis methods will make mistakes and care must be taken to present their output to users in a way they can easily use, modify, or take inspiration from, without requiring that they blindly accept a single system output.

This paper presents a complete machine learning approach for natural language (NL) to code translation, along with a detailed user study that demonstrates the effectiveness of the overall approach.

Question 1. I have a bunch of “.zip” files in several directories “dir1/dir2”, “dir3”, “dir4/dir5”. How would I move them all to a common base folder? (http://unix.stackexchange.com/questions/67503)

Solution: find dir*/ -type f -name “*.zip” -exec mv {} “basedir” 

Question 2. I have one folder for log with 7 sub-folders. I want to delete all the files older than 15 days in all folders including sub-folders without touching folder structure. (http://unix.stackexchange.com/questions/155184)

Solution: find . -type f -mtime +15 | xargs rm -f

Figure 1: Linux command-line questions posted on the Unix Stack Exchange forum. The answer to each is a bash one-liner: a command that can be typed at the bash command line.
null
• Our source code and dataset will be released upon publication to support reproducibility in software engineering research.

2 PROBLEM DEFINITION AND FORMAL OVERVIEW

Problem Definition. Following Desai et al. [5], we define programming by natural language (PBNL) as synthesizing a ranked list of programs \([Y_1, Y_2, \ldots, Y_k]\) based on the specification expressed as a single natural language sentence (denoted as \(X\)). A natural language sentence is defined as a sequence of words \(X = (x_1, \ldots, x_m)\) and a program is defined as a sequence of tokens \(Y = (y_1, \ldots, y_n)\). This definition differs from prior work [5], which relies on a context-free grammar definition of the programming language.

Our Approach. We present a machine learning approach for PBNL, which trains the synthesizer using pairs of natural language and programs (denoted as \((X, Y)\)). Figure 1 shows two such training examples. We use a novel three-step deep-learning approach, as illustrated in Figure 2:

(1) A user provides a natural language sentence \(X\), which Tellina transforms to a template \(\hat{X}\) by recognizing and abstracting over domain-specific constants, such as file names or times (§4.1).
(2) A recurrent neural network (RNN) encoder-decoder model translates \(X\) into a ranked list of possible program templates \(\hat{Y}_i\), where each \(\hat{Y}_i\) contains argument slots to be filled by entities recognized in item 1 (§3).
(3) The slots in each \(\hat{Y}_i\) are replaced by program literals to produce an output program \(Y_j\), using a k-nearest neighbor classifier which identifies the corresponding source entities (§4.2).

We train this model on a new corpus of English paired with Bash commands that we collected (§5.1), which focuses on a complex subset of the language (see fig. 3). The complete system, Tellina, permits users to input an example queries. Tellina outputs a list of proposed bash commands that may solve the user queries.

Our user studies demonstrate that Tellina significantly outperforms existing alternatives, even when there are mistakes in the set of proposed commands (§7).

3 RNN ENCODER-DECODER MODEL FOR TRANSLATION

Inspired by recent progress in neural machine translation [2, 32, 35] and its successful application in the domain of formal languages [6, 19, 33], we adopt an RNN encoder-decoder (sequence-to-sequence) model to translate a natural language template into a list of command templates. This section first explains RNNs (§3.1) and encoder-decoders (§3.2). Next, it describes the attention mechanism, a common practice which significantly improves the performance of encoder-decoders (§3.3). Finally, it presents the implementation and training details of the Tellina model (§3.4).

3.1 Recurrent Neural Network (RNN)

A recurrent neural network [11] encodes an input sequence of vectors into a single vector or expands an input vector into an output sequence of vectors. In the most general case, it serves both purposes at the same time. In our context, the sequences of vectors represent sequences of words or tokens (English sentences or bash commands).

An RNN consists of a set of cells, each consisting of three layers: input, hidden and output (fig. 4). Each token/vector in a sequence is processed by a different cell in turn. The input layer maps a token in the input sequence to an input state vector:

\[ x_t = I(x_t). \] (1)

The hidden layer takes the input state and the previous hidden state as input, and generates the current hidden state as output:

\[ h_t = f(h_{t-1}, x_t). \] (2)

The output layer takes the current hidden state as input, and generates an output state vector:

\[ o_t = g(h_t). \] (3)

Discrete output symbols can be decoded from the output state. The standard practice is to project \(o_t\) into a \(|V|\)-dimensional space, where \(|V|\) is the size of the output vocabulary, and apply the softmax function:

\[ y_t = \text{softmax}(W_{o}o_t). \] (4)

The softmax function outputs a unit vector. Therefore \(y_t\) can be interpreted as a probability distribution of the output vocabulary conditioned on the partial input history \(x_1, \ldots, x_t\):

\[ y_t \sim p(y_t|x_1, \ldots, x_t). \] (5)

In general, \(I\) is a look-up table and \(f, g\) are non-linear functions. The Tellina model defines \(f\) and \(g\) to be gated recurrent units (GRUs) [4].

3.2 Encoder-decoder models

As described in §3.1, an RNN can be used to encode an input sequence or to generate an output sequence. For most translation problems, the input sequence and output sequence are of different lengths; hence, it is difficult to use a single RNN to model both. Such problems can be solved using the combination of an
encoder RNN and a decoder RNN, which is commonly referred to as encoder-decoder modeling (Seq2Seq, eq. 4).\textsuperscript{3}

As shown in fig. 4, the encoder RNN and decoder RNN are connected in the hidden layer. The source sequence is fed into the encoder RNN. The output sequence is decoded from the decoder RNN, and the generation is biased by the final hidden state of the encoder RNN. The input symbol at step $t$ is its output symbol at step $t-1$. Using eq. (4):

$$y_t \sim p(y_t | y_1, \ldots, y_{t-1}, h_{\text{final}}).$$

Applying the chain rule to eq. (6), the decoder RNN defines a conditional probability distribution of the target sequences given the (encoded) source sequence:

$$p(y'_1, \ldots, y'_T | h_{\text{final}}) = \prod_{t=1}^{T} p(y'_t | y'_{1:t-1}, h_{\text{final}}).$$

The translation problem is hence reduced to decoding the target sequence with the maximum conditional likelihood:

$$\hat{y'} = \arg \max_{y'_1, \ldots, y'_T} p(y'_1, \ldots, y'_T | h_{\text{final}}).$$

The optimization in eq. (8) cannot be computed efficiently since $p(y'_1, \ldots, y'_T | h_{\text{final}})$ does not factorize step wise. Effective approximations include beam search [32] (which Tellina uses) or greedily picking the token with the maximum local score at each step.

### 3.3 Attention

Recent work in neural machine translation has shown that the model can benefit from making the prediction of a target output token depend directly on the encodings of the input tokens [2, 33]. For example, in fig. 4 the choice of the output token “[regex]” might be triggered by the tokens “[filename]” and “files” in the source sequence.

An attention mechanism [33] can be used to find and encode the relevant inputs. Specifically, it computes an alignment model $a$ between the encoder and decoder hidden states, and a context vector $c_t$ which is a sum of all encoder hidden states weighted by $a$:

$$a_{ti} = a(h'_i, h_t)$$

$$c_t = \sum_{i=1}^{T} a_{ti} h_i.$$  

The context vector $c_t$ is then used as the input to the decoder output layer together with the current hidden state $h'_t$:

$$o'_t = g(h'_t, c_t).$$

In general, $a_{ti}$ is defined to be the probability that the output at step $t$ of the decoder is translated from the input at step $i$ of the encoder. Following Dong and Lapata [6], we define $a_{ti}$ as the inner product between the decoder hidden state and the encoder hidden state, normalized over all encoder steps.

$$a_{ti} = \frac{\exp(h'_i \cdot h_t)}{\sum_{j=1}^{T} \exp(h'_i \cdot h_j)}.$$  

### 3.4 Training and hyperparameter setting

The Tellina model uses a bi-directional RNN [31] encoder, which consists of a forward RNN that reads the input sequence in the usual order and a backward RNN that reads the input sequence in the reversed order. The hidden states of the two RNNs are concatenated to generate the output state. The rest of the construction is the same as the base model presented above.

We trained the encoder-decoder using pairs of natural language and program templates. We use the standard sequence-to-sequence training objective [32] which maximizes the likelihood of the ground-truth program template given the natural language template. We trained the neural network (consists of all token embeddings and layer parameters) end-to-end using mini-batch gradient descent with Adam [14].

We set the decoder RNN to be 400-dimensional, and the two RNNs in the bi-directional encoder to be 200-dimensional. We set the mini-batch size to 16, the initial learning rate to 0.0001.

---

\textsuperscript{3}Some recent work used tree-structured RNNs in the encoder-decoder framework [6, 27]. We tried both. A Seq2Tree network did not yield significant performance improvement compared to Seq2Seq, but was dependent on a specific grammar definition. Future work can investigate more sophisticated encoder-decoder architectures.
and the momentum hyperparameters of Adam to their default values [14]. We set the beam size to 100 for beam-search decoding. The hyperparameters were set based on the model’s performance on a development dataset (§5.3). We will release our trained model and source code to support reproducible research.

4 ARGUMENT FILLING

The neural encoder-decoder model (§3) takes a templated NL sentence as input and produces a ranked list of program templates, which contain empty argument slots to be filled. In this section, we describe how the NL templates are constructed and how the program slot values are filled, to complete the full processing pipeline.

4.1 Template Generation

We use a domain-specific heuristic approach for finding and abstracting over the textual entities in the input sentences and the corresponding argument values in the programs they are paired with.

In the domain of bash commands, we identified two categories of open-vocabulary entities, patterns and quantities, and several second-level semantic types (fig. 5). To recognize and assign types to the entity mentions in the natural language sentence, we manually defined a regular expression for each semantic type to match the entities of that type. The recognizers for quantities perform well in general, since numerical expressions are strong predictors for such entities. However, these recognizers perform poorly for a few semantic types. For example, instead of writing a date expression in standard format like "08/02/2017", the user may describe it verbally, e.g., "the 2nd of August of 2017", and our heuristic patterns cannot cover all possible cases. To generate a natural language template used as input to the sequence-to-sequence RNN from a sentence, we replace each open-vocabulary entity in the sentence with its semantic type.

We also generate command templates from commands. First, we type each argument in the command according to its type definition in man pages. Second, we map the man-page types to one of the 9 semantic types defined in fig. 5. For example, in Linux man pages, the target directory argument of find is of type path, and the argument of the option -name is of type pattern; our rules map path to Path and pattern to Regex. Finally, we replace each command argument with its semantic type.

4.2 Program Slot Filling

Entity mentions in an NL sentence often form a one-to-one mapping with a subset of the arguments in its corresponding program. Hence, Tellina performs argument filling in two steps. It first aligns the list of entity mentions with the program slots using the stable matching algorithm (§4.2.1, §4.2.2). Then, it extracts the argument values from the aligned entity mentions and fills the argument slots to form the complete program (§4.2.3).

Aligning entities with argument slots is challenging because of the following two types of ambiguity. First, there are usually multiple entity mentions and multiple argument slot. While type-matching provides a strong signal for the alignment, there may exist multiple entities of the same type. Such ambiguities need to be resolved based on the contextual information of the entity mentions. For example, in the task of “moving some files from the source directory to the target directory”, the algorithm needs to correctly identify the source and target directory without swapping their positions. Second, inferencing entity and argument types based on heuristics may be imprecise, and wrong types can lead to wrong alignment results. For example, heuristically, we cannot confidently determine whether the substring "2017" in a user’s input refers to a list of files, directories, or some other entity without looking into its context.

To tackle above challenges, our slot-filling algorithm makes use of the contextual information of the entities and the slots.

4.2.1 Global entity-slot alignment. Algorithm 1 computes the alignment between the entities and slots using a modified version of the stable matching algorithm [21]. Besides entities and slots, the algorithm also takes as input a local entity-slot match scoring function \(\gamma(i, j)\) whose output represents how likely the entity \(e_i\) is to be matched with the slot \(s_j\) based on local information. For example, such a local function may be type based, and "15 days" is more likely to be an argument of -mtime compared to "log files" since their types match.

Upon initialization, the algorithm first computes a preference list of slots for each entity, based on the local matching scores returned by \(\gamma(i, j)\). At each iteration of the while loop, the algorithm selects an unmatched entity mention \(e_i\) with a non-empty preference list \(S_{e_i}\), and attempts to match it with the highest-ranked slot \(s_j\) in \(S_{e_i}\). If \(s_j\) is not matched or \(s_j\) prefers \(e_i\) to the entity mention \(e_i\) it is currently matched to, \(e_i\) and \(s_j\) become matched. Otherwise \(e_i\) stays unmatched.

The while loop is guaranteed to terminate. Upon termination, it is guaranteed that every entity is aligned to at most one slot [21]. If every entity mention is aligned to exactly one argument slot, it returns the alignment. Otherwise, it returns the null value indicating not all entities can be aligned to a slot. In both cases, it is possible for some argument slots to be left unmatched.

5 In practice, we found this is an effective rule for detecting wrong program templates. Therefore Tellina filters out all command templates that do not have enough argument slots for the recognized entities.
We developed a novel k-nearest neighbor formulation for the local entity-slot compatibility function $\gamma(i,j)$. For some slots that are left empty, we automatically fill in default values based on heuristics. For example, we use "," as the default value for the target directory of find, since the users often omit the target directory in their NL description if it is the current directory. If a slot is not defined with a default value, we leave it empty when presenting to users.

4.2.2 K-nearest neighbor classifier for local entity-slot matching. We developed a novel k-nearest neighbor formulation for the local matching function $\gamma(i,j)$ introduced in alg. 1. It is based on the intuition that given the embeddings of the entity mentions and argument slots generated by the encoder-decoder (fig. 4), the embeddings of a matched entity-slot pair shall be closer to other matched entity-slot pairs in the embedding space and farther away from the unmatched entity-slot pairs.

Specifically, we sampled a set of positive and negative entity-slot pairs from our training data and use them as the training set of the k-NN classifier. We represent each entity-slot pair $(e_i, s_j)$ using the concatenation of the hidden state vectors $(h_i, h'_j)$ of the neural encoder-decoder model. For each test $(e_i, s_j)$ pair, we define $\gamma(i,j)$ as a voting score weighted by the distances between $(h_i, h'_j)$ and all the positive and negative training examples in the embeddings space:

$$\gamma(i,j) = \sum_{(c,d) \in NN(i,j,k)} d(i,j),(c,d) \cdot v(c,d),$$

where $NN(i,j,k)$ is the set of k-nearest neighbors of $(e_i, s_j)$ in the training set. $d(i,j),(c,d)$ is the distance function and $v(c,d)$ is the vote of neighbor example $(e_c, s_d)$, as defined below

$$d(i,j),(c,d) = \cos(h_i, h'_j, h_c, h'_d),$$

$$v(c,d) = \begin{cases} 1, & \text{if } (e_c, s_d) \text{ match } \\ 0, & \text{otherwise} \end{cases}.$$

This approach leverages the contextual information of both the entity mention and the argument slot, since the hidden state representations of $e_i$ and $s_j$ encode their context information. For example, the hidden states of the two occurrences of "css" in the sentence "find all css files in the folder css" are different, because they are surrounded by different words.

4.2.3 Post-processing. Given the aligned entities and argument slots, extracting the entity values and reformating them to complete the argument filling is easier and heuristics-based approach works well. Some entities such as file paths and string patterns can be directly copied into the slots. We define a small set of heuristic rules for reformating the entities that cannot be directly copied, such as adding wild-card symbol "*" in the front of a file extension and converting "x weeks" to "x days". These rules often work well in practice, but do introduce some errors. For example, if a user describes a regular expression pattern verbally as "all log files whose name starts with 2016", we do not have a rule which can infer the regular expression "2016.log". Further improving argument value extraction is an interesting area for future work.

5 DATA

We collected 8,000 pairs of NL description and bash command from the web (§5.1), from which 5,413 pairs remained after filtering (§5.1). We split this data into train, dev, and test sets, subject to the constraint that no NL description appears in more than one dataset (§5.3). Our dataset is publicly available for use by other researchers.

5.1 Data Collection

We hired freelancers who were familiar with shell scripting through Upwork\(^6\) to collect data. They searched for web pages that contain bash commands and recorded command-text pairs from them. Each text in a pair is a sentence that describes the command, either extracted from the webpage or described by the freelancer based on their background knowledge and web page contexts. We restricted the bash commands to be one-liners and the natural language description to be a single sentence. The source web pages include tutorials, tech blogs, question-answering forums, and course materials. The freelancers collected data through a web interface developed by us, which helps them with page searching, pair recording, and duplicate data elimination. On average, each freelancer collected 50 pairs per hour.

Filtering. After obtaining text-command pairs collected by the freelancer, we first filter the dataset with the following rules. First, we discarded all commands that cannot be parsed by the bash parser

\(^6\) Locascio et al. [19] uses RNN encoder-decoder to synthesize regular expressions based on a natural language description. Integrating a similar module into Tellina’s neural architecture to synthesize the argument content could be an interesting future direction.

\(^{http://www.upwork.com/}\)
Bashlex⁸. Second, we discarded all commands that contain out-of-scope bash operators, as shown in Figure 3. Finally, we discarded commands that contain an operator appeared less than 20 times in our dataset (e.g. cut, du, bzip2, etc.), since the model is unlikely to learn them from the data due to sparsity.

"Cleaning. After filtering, we clean both texts and commands in the data set. For texts, we use a probabilistic spell checker⁹ to correct spelling errors in the English descriptions. We also manually corrected a subset of the spelling errors that bypassed the spell checker in the English and in the bash commands.

For commands, we first remove sudo and shell input prompt characters such as "$" and "#" from the beginning of each command and replaced absolute command pathnames by their base names (e.g., we changed "/bin/find" to find). Then, we used a parser augmented from Bashlex to parse the command into an AST. The Bashlex AST handles nested command structures such as pipelines, command substitution, and process substitution. We augmented it to map each argument to the utility or utility flag it attaches to, using the command syntax defined in the Linux man pages. The augmented syntax is used to generate the bash command templates, as described in §4.1.

5.2 Data Statistics
After filtering and cleaning, our dataset contains 5,413 (NL, bash) pairs. These commands contain 17 different bash utilities and more than 200 flags. In descending order of frequency, the utilities are find, xargs, grep, egrep, fgrep, ls, rm, cp, mv, wc, chmod, chown, chgrp, sort, head, tail, tar.

Similar to other machine translation datasets [26], in our (NL, bash) dataset, one natural language description may have multiple corresponding correct bash command solutions, and one bash command may be phrased in multiple different NL descriptions. In general, higher number of such multiple-to-multiple correspondences between NL descriptions and bash commands implies bigger challenges for learning and evaluation. First, we are unlikely to collect all possible translations for an NL description, and the model could wrongly penalize correct predictions which it does not recognize during training. Second, at test time, the model may predict correct translations that are different from the ground-truth, and manual evaluation has to be done to compute the correct evaluation metrics (§6.1.2). We present the statistics of such correspondences in Table 1.

5.3 Data split
We split the filtered data into train, development (dev) and test sets. We first clustered the pairs by their NL templates — a cluster contains all pairs with the identical NL template. Then, we randomly split the clusters into 80% training, 10% dev, and 10% test. This prevents the model from testing a NL template that was included in the training set, which allows us to evaluate the model’s ability to generalize to new NL inputs. Table 1 shows the statistics of our data split.

Following the machine learning practice, we first trained our model on the training set and used the dev set to tune the hyperparameters (§3.4). Then, we trained our final model on the combination of both training and dev sets, with the hyperparameters achieving translation accuracy on the dev set (§6.1) obtained during parameter tuning phase.

6 MODEL EVALUATION
We report the end-to-end translation and argument filling accuracy (§6.1) for our new bash dataset, and a short discussion of qualitative results (§6.2).

6.1 Translation Accuracy
We report the end-to-end translation accuracy, both automatic and manually computed, of the Tellina model and a code retrieval baseline.

6.1.1 Baseline Model. We implement a code retrieval (CR) baseline using the tf-idf information retrieval (IR) technique [20]. The IR model encodes every NL description in the dataset into a bag-of-words feature vector using the tf-idf statistics. Given a test example, it computes the cosine-similarity between the test NL feature vector and the training NL features vectors, and returns the top-k most similar commands. We improved the IR model by first retrieving the command template using the NL template similarity, and then perform argument filling for the NL template using type-matching heuristics.¹⁰

6.1.2 Evaluation Methodology. We report two types of accuracy: top-k full-command accuracy (Acck) and top-k command-template accuracy (AccT). We define AccT for which a correct full command is ranked k or above in the model output, and AccT to be the percentage of test examples for which a correct command template is ranked k or above in the model output (i.e. ignoring errors in the constants).

As described in §5.2, many test examples have more than one correct commands and our collected data may not cover them all. Therefore, we asked three freelancers from Upwork who are familiar with shell scripting to evaluate the model output manually. The freelancers independently examined the top-3 translations of both the CR baseline and the Tellina model for all test examples, and

<table>
<thead>
<tr>
<th></th>
<th># pairs</th>
<th>cmd/nl</th>
<th>percentage</th>
<th>nl/cmd</th>
<th>percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Train</td>
<td>4330</td>
<td>1.21</td>
<td>13%</td>
<td>2.13</td>
<td>27%</td>
</tr>
<tr>
<td>Dev</td>
<td>559</td>
<td>1.13</td>
<td>9%</td>
<td>1.39</td>
<td>19%</td>
</tr>
<tr>
<td>Test</td>
<td>524</td>
<td>1.19</td>
<td>12%</td>
<td>1.40</td>
<td>15%</td>
</tr>
</tbody>
</table>

Table 1: Data statistics. "cmd/nl" is the average number of bash commands per NL description, and the following column is the percentage of NL with more than one bash command translations. Similarly, "nl/cmd" is the average number of NL descriptions per bash command, and the following column is the percentage of bash command with more than one NL descriptions.

¹⁰We found the retrieval results based on full NL description to be significantly worse, since the constants are likely to receive high idf weights while not being representative of the command semantics.

¹¹We treat the (bash, NL) pairs with the same NL description as a single test example.
evaluate correctness of the commands at both the full-command level and the command-template level. For each command translation, we use the majority vote of the three freelancers as the final evaluation.

6.1.3 Results. Table 2 shows the translation accuracies of both the Tellina model and the CR baseline. The Tellina model beats the CR baseline by a large margin on all accuracy metrics. It achieves strong template-based accuracy, up to 80% on the top-3 metric, indicating the effectiveness of the neural encoder-decoder model. On the other hand, both models struggle to generate full commands, often making mistakes with one or more of the command arguments. Nonetheless, as we will see in §7.6, users still found these predictions useful, even if the final output needs some corrections before it can be executed.

We also evaluate the argument filling component individually by using alg. 1 to fill in the arguments for ground truth templates. Table 3 shows the precision, recall, F1 measurement on the development set with varying k parameters. The model presents high accuracy in filling arguments to the templates (~86.7% F1 with optimal k). However, due to cascading errors from entity detection and template selection, the overall command correctness ratio is much lower than template correctness ratio, as we will discuss below.

6.2 Error Analysis

While the template correctness and argument filling accuracies are high (Table 2), we observed that complete command accuracy is much lower overall. To better understand this phenomena, we sampled 50 synthesized commands whose template is correct but full command is incorrect.

Among these 50 incorrect commands, 41 of them are caused by Tellina’s failure to recognize the argument from the NL description, 5 are caused by wrong command alignment, 2 caused by the fact that the NL description does not provide a concrete argument, and the last 2 are caused by predicting incorrect templates (which the freelancers failed to catch). The vast majority of the NL entity recognition failures involve missing idioms in the task description. For example, our algorithm is unable to extract the directory argument ‘/’ from the idioms “root directory” or “full file system”, and so is the case for extracting permission code “100” from idiom “read permission”.

However, while this type of error harms the full-command translation accuracy of Tellina model, it does not significantly harm the tool usability: we observed in our user study (§7) that many users can easily formulate arguments that our algorithm failed to recognize based on their knowledge of the file system; as a result, these errors can be easily fixed given correct template and alignment.

7 USER STUDY

We conducted a user study to determine whether Tellina helps programmers complete file system tasks using bash.

7.1 Experiment Design

We measured each participant’s performance under the following two treatment conditions.

- Control treatment: The participant may use any local resource (such as man pages and experimentation on the command line) and any Internet resource (such as tutorials, question-and-answer websites, and web search). This emulates how a programmer would normally solve a file system task.
- Experimental treatment: The participant may use any of the above resources, and also Tellina.

We adopted a counterbalanced factorial design in which each participant performs two sets of tasks, one with each treatment. This design prevents confounding due to order effects by randomly assigning the participants into four groups, which correspond to all four taskset and treatment combinations.

We recruited 39 students in the computer science major to participate in the experiment (24 graduate students, 15 undergraduates). None were familiar with Tellina. All of them were familiar with bash. We accepted only graduate students who self-reported to be bash users, and we accepted only undergraduates who had completed or were enrolled in our department’s Linux tools course.

We excluded data from 4 of the participants, because 3 of them forgot to switch treatment conditions between the tasksets and 1 of them did not complete the study.

7.2 Tasks

Each taskset is made up of 8 tasks. Each task consists of an English description of a file system operation, and a file system in a pre-defined initial state. The desired outcome is either a list of files (possibly with additional attributes such as modification time or number of lines) or a change in the file system, such as deleted/added/modified files. The user’s goal is to write a bash command that performs the desired operation without causing extra file system changes. All tasks have outcomes that are automatically verifiable.

The participant may attempt a task multiple times. The participant may reset the file system to its original state in order to start over from a fresh slate. Each task has a 10-minute timeout. If the participant gives up on a task, we count the participant as having spent 10 minutes. In addition, each taskset has a time limit of 40 minutes.\footnote{Eleven participants hit this time limit in the experiment (often only for the first taskset).}
7.2.1 Selection and filtering. The experiment uses real tasks selected from four websites offering programming help: Stack Overflow (http://stackoverflow.com/), Super User (http://superuser.com/), commandlinefu (http://www.commandlinefu.com/), and Bash One-Liners (http://www.bashoneliners.com/).

To obtain candidate tasks from the file system domain, we first extracted all questions from these websites tagged with "bash" and "find", and obtained 401 questions. We retained the 146 of them that can be answered using the 17 bash utilities that appear in Tellina’s training set. We did not filter tasks by the flags, and a task may require a flag that is not in Tellina’s training set. Finally, we randomly sampled 16 out of the 146 tasks. The answers to those 16 tasks use find, xargs, mv, cp, rm, grep, wc, ls, and tar.

7.2.2 Rewriting. We asked researchers who are not involved in this project to paraphrase the descriptions of the selected tasks. This prevents the original task from being trivially found on the web in case the user copy-and-paste the task description into a search engine. Eight researchers in total contributed to the rewriting, and each of them rewrote 1 to 3 tasks. This avoids biasing the participants’ phrasing of their natural language queries to be similar to a specific writer.

7.2.3 File system. We selected a code repository from GitHub and used it as the file system for our tasks. The file hierarchy is 4 levels deep and consists of 29 folders and 62 files. We changed all constant values in the task descriptions to match the contents of this file system.

7.3 Tool Interface

Tellina is a web application which can be accessed through a URL. It has an interface similar to the Google search engine. A user types a natural language sentence describing a task, then the website displays the RNN model’s top 20 bash command translations of the sentence.

To help users understand the output commands, a user can hover over a token, such as a program name or flag, and see the man page description of the token. To provide a level playing field and avoid conflating this explanation feature with use of Tellina, we gave the participants a third-party tool, explainshell (http://explainshell.com/), which provides exactly the same man-page explanation functionality in the control treatment.

To help users in all treatments understand the effects of their commands, we provided a visual file system comparison tool (similar to Araxis Merge, KDiff3, or Meld) that indicates which files and directories differ and permits the user to interactively navigate the file system. This enables a user to understand what is wrong with his or her command without interpreting voluminous, possibly confusing diff output.

7.4 Training

Before starting a taskset, each participant completes a training task with the appropriate treatment condition. The Tellina website includes a tutorial to explain the usage of the interface (e.g., the input should be a complete imperative sentence) and the output presentation. We assumed all participants were familiar with web search and man pages and did not provide training for them.

7.5 Quantitative results

Table 4 lists the independent and dependent variables. We performed a four-way analysis of variance (ANOVA) for each dependent variable. The statistically significant results ($p < 0.01$) are that all four independent variables predict the time spent, and subject and taskset also predict the success rate.

The effects of subject on time are expected, because the participants are at different bash proficiency levels. There was no statistically significant effect of subject on success rate because the generous time limits (10 minutes per task, 40 minutes per taskset) enabled most users to complete most tasks. The overall success rate was 88%, and we were more interested in how to help programmers become more efficient, because we know that programmers can manage to solve tasks if given enough time.

The effects of order are also expected. On average, the participants spent 20% more time on the first taskset they encountered than on the second (1767 seconds vs. 1414 seconds), but were less successful (84% vs. 92% success rate). This learning effect reflects increasing participant familiarity with the example file system, tools such as file system diff, and bash tricks they learned earlier in the experiment.

The effects of taskset indicate that we failed to create two tasksets of equal difficulty. Users spent 24% more time, but were 10% less successful, with taskset TS2.

Our counterbalanced factorial design enables the most interesting effects, those of treatment, to be accurately determined despite the effects of subject, order, and taskset. Participants in the Tellina treatment spent on average 22% less time (1397 seconds vs. 1784 seconds). This indicates that Tellina helps programmers to write bash commands in less time. The effect of treatment on success rate is significant only at the $p < 0.1$ level ($\mu_{Tellina} = 90\%, \mu_{Control} = 85\%$), for the reasons noted above when discussing the effect of subject on success rate.

7.6 Qualitative Results

Each participant filled out a questionnaire about their experience during the study.

The first part of the questionnaire consists of four Likert-scale questions (table 5). On average the participants wanted to use Tellina in the future (5.8/7). Tellina’s partially correct suggestions were helpful (5.2/7) and did not slow down the users (3.2/7), but

<table>
<thead>
<tr>
<th>Variable</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>TC</td>
<td>(1, . . . , 35)</td>
</tr>
<tr>
<td>TS</td>
<td>(TS1, TS2)</td>
</tr>
<tr>
<td>Order</td>
<td>(1st, 2nd)</td>
</tr>
<tr>
<td>Time spent (sec.)</td>
<td>[0, 2400]</td>
</tr>
<tr>
<td>Success rate</td>
<td>[0, 1]</td>
</tr>
</tbody>
</table>

Table 4: Experimental variables. Order indicates whether the taskset was the user’s first or second taskset. Success rate is the fraction of the 8 tasks in the taskset that the user completed successfully.
were difficult to correct (4.6/7). These results confirm our hypothesis that programmers are resilient to noise in the Tellina output and can use it as inspiration when formulating the correct command themselves. Anecdotally, while using Tellina ourselves we learned about command-line flags that we had not known about.

The second part of the questionnaire was four open-ended questions asking the participants to comment on specific features of Tellina and suggest future improvement (table 6). Below summarizes our findings.

**Useful features.** Many participants noted the usefulness of partially correct suggestions. Even when the full command was not correct, the web interface gave documentation and prompted the users to look up command options that they otherwise would not have remembered. Participants also liked the fact that Tellina suggests multiple solutions, which allows users to compare them and decide which one to try. Some participants liked Tellina’s argument-filling feature. Tellina returned an answer with constants appropriate to the user’s query, enabling the user to try out the command immediately without having to adjust the constants manually.

**Limitations.** Some participants noted that when Tellina suggested a wrong command that was close to a correct suggestion, it was difficult to trouble-shoot exactly what went wrong. Participants were also frustrated by subtle syntactic errors that prevented the commands from being run as is. (Tellina gives no guarantee that its output is a legal bash command.) Some participants were slowed down by incorrect argument formatting in Tellina’s output, such as a missing “+” sign in the argument to find’s -mtime flag. Sometimes the RNN model suggested unusually complex commands (e.g., long pipelines), and the participants found them distracting even if some of the commands in them were correct.

**Suggestions.** Many participants requested better explanations of the output commands, so that they can better decide which one to try. For example, Tellina could incorporate a bash→English translator (either hand-coded or a learned RNN model) to explain its bash command output. Some participants suggested flagging which commands are syntactically valid, or providing sample output. Some participants also suggest interactive features that would enable the user to correct the output or give hints such as “the task needs to be solved using the tar utility”.

<table>
<thead>
<tr>
<th>Question</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>Do you want to use Tellina in the future?</td>
<td>5.8 ± 1.2</td>
</tr>
<tr>
<td>How often did partially correct suggestions help you?</td>
<td>5.2 ± 1.5</td>
</tr>
<tr>
<td>How often were you slowed down by the incorrect suggestions made by Tellina?</td>
<td>3.2 ± 1.4</td>
</tr>
<tr>
<td>How easy was it for you to correct the incorrect suggestions made by Tellina?</td>
<td>4.6 ± 1.2</td>
</tr>
</tbody>
</table>

Table 5: Mean and standard deviation of the participant responses to the Likert-scale questions. All questions have scale 1–7.

<table>
<thead>
<tr>
<th>Question</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>What features of Tellina are the most helpful to you?</td>
<td></td>
</tr>
<tr>
<td>What mistakes made by Tellina affected you most?</td>
<td></td>
</tr>
<tr>
<td>Please list the features that you think we should add to Tellina.</td>
<td></td>
</tr>
<tr>
<td>Please give us any additional comments you have about Tellina.</td>
<td></td>
</tr>
</tbody>
</table>

Table 6: Open-ended questions in the post-study questionnaire.

8 RELATED WORK

**Programming by natural language.** There has been extensive research on synthesizing programs from natural language descriptions. Rule-based approaches have been developed to synthesize SQL queries [18], SmartPhone scripts [17], Java method specifications [25], and Spreadsheet programs [9]. These methods can produce complex programs, but may require non-trivial manual effort to build and maintain. Recently, machine learning techniques have been developed to build probabilistic models to represent the joint distribution of text and programs [5, 10, 13]. We extend this line of work by introducing an RNN encoder-decoder approach which can be applied to many different synthesis problems with relatively little domain-specific effort.

Our approach is also related to other deep-learning based PBNL approaches. DeepAPI [8] addresses the problem of retrieving API call sequences based on the user’s natural language queries, using the RNN encoder-decoder model. CodeMend [30] proposed encoder-decoder models which complete partial programs by jointly modeling user’s natural language input and the contextual programs. In comparison, Tellina directly synthesizes executable programs from NL descriptions and is able to handle open world arguments using our novel argument filling algorithm. Neural Programmer [23, 24] is a recently proposed deep learning architecture that allows direct encoding of discrete operators to improve learning efficiency. Since command languages cannot be succinctly represented using a few core operators, it is difficult to apply this approach to our domain. In addition, we present the first controlled user study demonstrating significant effectiveness of such techniques, despite their imperfect accuracy.

**Semantic parsing.** The problem of mapping natural language to programs or other formal representations has been extensively studied in the natural language processing community [1, 3, 6, 29, 36]. Earlier machine learning research in this area focus on learning formal grammars for mapping language to meaning representations [3, 36]. However, it has recently been shown that deep-learning based approaches work equally well, for example to produce regular expressions [16] and database queries [6, 23]. We expand the domain by studying deep learning for producing command languages, and evaluate the effect of such systems on programmers.

**Deep learning in software engineering.** Finally, deep-learning based approaches have also been applied to other software engineering problems, such as defect prediction [34], program feature extraction [22, 28], summarizing code using natural language [12], and program induction [7, 15]. In general, these applications leverage big data and design custom neural architectures for each application. In comparison, our focus is on studying the usefulness RNN encoder-decoder models, which have been shown to work well for a wide variety of program synthesis problems.
9 CONCLUSION
This paper presents an approach for program synthesis from natural language that leverages state-of-the-art neural machine translation techniques, augmented with slot (argument) filling and other techniques. We studied the complex domain of hash file system operations and conducted a controlled user study which shows that our tool, Tellina, significantly improves programmers’ efficiency despite being imperfect in its program predictions.

As future work, it would be interesting to extend our neural architecture so that the entire framework for entity recognition, template translation and argument filling can be learned end-to-end. It should also be possible to extend the approach to cover more programming languages.

REFERENCES