StriSynth: Synthesis for Live Programming

Sumit Gulwani*, Mikaël Mayer†, Filip Niksic‡, and Ruzica Piskac§
*Microsoft Research Redmond, USA, Email: sumitg@microsoft.com
†EPFL, Switzerland, Email: mikael.mayer@epfl.ch
‡MPI-SWS, Germany, Email: fniksic@mpi-sws.org
§Yale University, USA, Email: ruzica.piskac@yale.edu

***Abstract*—Motivated by applications in automating repetitive file manipulations, we present a tool called StriSynth, which allows end-users to perform transformations over data using examples. Based on provided examples, our tool automatically generates scripts for non-trivial file manipulations.**

Although the current focus of StriSynth are file manipulations, it implements a more general string transformation framework. This framework builds on and further extends the functionality of Flash Fill—a Microsoft Excel extension for string transformations.

An accompanying video to this paper is available at the following website http://youtu.be/kkDZphqIdFM.

I. Introduction

This paper describes an early prototype of a tool called StriSynth that synthesizes scripts based on input examples. Writing shell scripts and regular expressions can be a tedious task even for experienced computer users. Faulty scripts that lead to data loss and corrupted files are not uncommon. When a non-expert user seeks help in writing a script, she usually describes her intentions through examples: she illustrates what a script should do on the representative inputs that best convey her intentions. We mimic this scenario in StriSynth: the user demonstrates how chosen input strings should be manipulated, and we automatically synthesize a command that captures the user's intentions. If the user is unhappy with the suggested command, she can provide more examples to help StriSynth derive a better script.

While helping with writing scripts is an important problem to address, the majority of computer users cannot write scripts, but yet need them every day. For example, a user might want to organize their photos in a specific way. Every user has a different style for naming files, so it is infeasible to find a general script that works for everyone. Our additional motivation was to develop a tool that helps users automate file manipulations. StriSynth monitors the previous commands that have been executed by the user and then suggests, in an interactive manner, a script that will manipulate the remaining files that still need to be processed. The preliminary versions of our tool can be seen at [1], [2], [3].

We envision two groups of users benefiting from StriSynth:

- To the large community of end users, our tool represents an easy way to do complex file manipulations without actually writing scripts. To ensure that the tool always performs the operations that the user has in mind, the user also receives a description of the planned transformations in English.
- Programmers and experienced users can use StriSynth as a synthesis tool to help derive complex shell scripts from simple examples. Deriving scripts automatically increases programmer productivity and makes code less error-prone.

StriSynth builds upon the Programming by Example (PBE) paradigm [4], [5], [6]. PBE is a promising research direction that can enable rich, easy data manipulation even for non-programmers [7]. The success and impact of this line of work is witnessed by the fact that some of this technology [8] ships as part of the popular Flash Fill feature in Microsoft Excel 2013. Recent work in PBE has mostly focused on manipulating atomic data types such as strings [8], [9], [10] and numbers [11]. In contrast, StriSynth allows manipulating hierarchical data types such as nested lists of strings and numbers.

StriSynth can be seen as a live programming environment, with the key benefit being that it allows end-users to investigate the synthesis result without having to look at source code. Even more significantly, it allows the user to identify new examples that will accelerate convergence to the intended result/program.

The basis of StriSynth is an algorithm for automated synthesis of string transformation. The operations that StriSynth supports (cf. Sec. III) are, in summary, string manipulations. Although we list simple file manipulations as our main motivation, our tool can handle more general transformations. Our goal is to extend the functionality of StriSynth far beyond these initial intentions. For instance, we plan to generalize StriSynth to handle complicated manipulations of tabular data.

II. Motivating Examples

We illustrate the functionality of StriSynth on three different scenarios. The scenarios were chosen to show the range of capabilities of StriSynth.

A. Complex File Renaming

A typical user has a large number of files that she plans to organize "one day", but never does so as the work is tedious and repetitive. A common instance of such a task is organizing photographs based on the date when they were taken. StriSynth can help automate such tasks, as it supports complex file renamings involving both syntactic and semantic transformations. In particular, StriSynth supports digit-to-month (English) and month-to-digit transformations. To organize image files, the user only needs to provide a single example indicating her intentions; she renames a file, and asks StriSynth to rename the rest of the files accordingly. To do this, she gives the following command as an input:

```
mv 20141121-001.jpg 2014Nov21/001.jpg
```

Based on the given example, StriSynth detects a date in the file name, and converts it from the digit format into the YYYY-MM-DD format. StriSynth next generates a sequence of commands that correspond to renaming every image file in the directory according to the above pattern.

Even if there were different date formats in the file names, StriSynth would have detected them and followed the convention indicated by the user. While writing a script or renaming all the files manually can be a very tedious task, complex file manipulations using StriSynth usually require only 1-2 representative examples (see Table I).

B. Merging Parts of Books Together

Consider the following listing of a directory and the task of merging multiple chapters of each book into a corresponding file:

```
algorithms1.pdf
[...]
algorithms3.pdf
graphics1.pdf
[...]
graphics17.pdf
math1.pdf
[...]
math5.pdf
```

Since the files have different naming patterns, writing a script to do this task could be difficult for non-expert users. Using StriSynth, the user can semi-automate the process of creating the needed routine:

```
convert algorithms1.pdf [...] algorithms3.pdf
    algorithms-book.pdf;
convert graphics1.pdf [...] graphics17.pdf
    graphics-book.pdf;
convert math1.pdf [...] math5.pdf
    math-book.pdf;
```

The first step for the user is to group the files with similar names, and merge each group into a single file. To address the need for grouping strings, one of the main features of StriSynth is the *partition* operation. The user provides a small number of example strings that should be grouped together, and based on these examples, StriSynth partitions the rest.

In our scenario, a good way for the user to indicate her intention is to declare the following two groups:

Group 1	Group 2
algorithms1.pdf	graphics1.pdf
algorithms2.pdf	graphics2.pdf

Based on these examples, StriSynth recognizes how the file names should be partitioned, and displays the following message:

> Groups files by file name until the beginning of the first number. The name of the category is the constant string 'Group ' + a 1-digit counter starting at 1.

As this message correctly reflects the user's intentions, she does not need to provide any further examples, and StriSynth partitions the files accordingly.

Note that even though there are three groups, it was sufficient for the user to provide examples for only two groups. This feature is particularly useful when there are many groups—indicating good representatives for even two groups can help StriSynth categorize a large number of files.

Once the partitioning is finished, the user sees three groups of file names (three different directories). Let us, for a moment, consider Group 1. The user can write the following command to combine the files into a single file:

```
convert algorithms1.pdf algorithms2.pdf ...
    algorithms-book.pdf
```

So far we have used the ellipsis (...) as a meta-symbol, to denote that some parts of a string are missing. However, in the previous command, the user literally enters the ellipsis, and StriSynth automatically expands the command to:

```
convert algorithms1.pdf algorithms2.pdf
    algorithms3.pdf algorithms-book.pdf
```

After validating the command, the user asks StriSynth to synthesize the corresponding complete commands for all other partitions. In particular, the following command is generated for the third directory:

```
convert math1.pdf math2.pdf math3.pdf
    math4.pdf math5.pdf math-book.pdf
```

At this point, the user could simply copy/paste each of the three commands to the console and execute them. However, we can go one step further and synthesize a single script that will execute all three commands. For this purpose, the user invokes the *reduce* operation in StriSynth. To indicate an example reduction, the user copies and pastes the first and the second command separated by a semicolon:

```
convert algorithms1.pdf [...]
    algorithms-book.pdf;
convert graphics1.pdf [...]
    graphics-book.pdf; ...
```

In this command, the first two occurrences of the ellipsis are meta-symbols, but the last one is literal. Based on the provided specification in the form of a partial command, StriSynth is able to correctly complete the command.

C. Renaming and Printing Documents

In this subsection, we demonstrate two additional operations provided by StriSynth: *filter* and *split*. The filter operation can be seen as a variant of the partition operation—it partitions a list of strings based on a predicate, and it keeps those parts for which the predicate is true. The split operation splits a string into a list of strings based on a few examples of the initial elements of the list. At the end of the subsection we also demonstrate that the tool can learn counters by example.

To illustrate the split and filter operations, consider the following scenario: the user has a list of files and wants to send all the DOC files to a printer. The files are listed in a comma-separated string:

```
img1.jpg, img2.jpg, reportA.doc,
reportA.pdf, reportB.doc, reportB.pdf,
reportC.doc, reportC.pdf, report4.doc,
report4.pdf, summary.doc, conference.mp3
```

The user first needs to split the string to get a list of file names. She gives examples of the first two elements of the list: `img1.jpg` and `img2.jpg`. Based on these examples, StriSynth detects the delimiter and performs the split.

Next, the user wants to process only the DOC files. She gives `reportA.doc` and `reportB.doc` as positive examples:

```
reportA.doc -> OK
reportB.doc -> OK
```

It turns out this is not enough. Based on these two examples, StriSynth concludes that the filtering needs to be done based on the substring `report`, which incorrectly classifies `reportA.pdf` as a positive example. Therefore, the user provides `reportA.pdf` as a negative example:

```
reportA.doc -> OK
reportB.doc -> OK
reportA.pdf -> notOK
```

Now StriSynth correctly performs the filtering, and outputs the following list:

```
reportA.doc
reportB.doc
reportC.doc
summary.doc
```

Finally, to illustrate the use of counters, let us assume that the user wants to print those DOC files, but she also wants to rename the files to add a counter that indicates the order in which the files were printed. She inputs an example command for `reportA.doc`:

```
lpr reportA.doc;
    mv reportA.doc printed01_reportA.doc
```

For `reportB.doc`, StriSynth generates the following command:

```
lpr reportB.doc;
    mv reportB.doc printed01_reportB.doc
```

This is almost correct, but the new name should contain “02” instead of “01”. The user corrects this issue manually and runs StriSynth again. It now detects the increasing counter and synthesizes the correct list of commands. After the final reduction, the synthesized command is:

```
lpr reportA.doc;
    mv reportA.doc printed01_reportA.doc;
lpr reportB.doc;
    mv reportB.doc printed02_reportB.doc;
lpr reportC.doc;
    mv reportC.doc printed03_reportC.doc;
lpr summary.doc;
    mv summary.doc printed04_summary.doc;
```

III. StriSynth: A System Overview

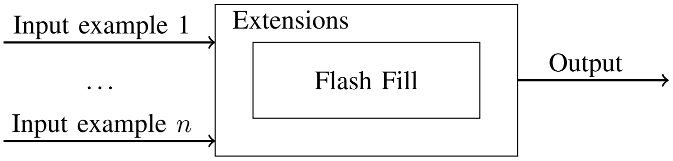

Fig. 1. System architecture of StriSynth: it uses Flash Fill as a black box and implements additional functionality on top.

An overview of StriSynth is given in Figure 1. StriSynth implements the operations discussed in the previous section. These operations leverage the Flash Fill [8] algorithm as a black box, which has the added advantage of being able to plug in any string transformation learner in place of Flash Fill.

In order to understand how that works, we can think of Flash Fill's learning algorithm as a function which takes a list of examples of how n-tuples of strings should be transformed into strings, and produces a *transformer* from n-tuples into strings that works correctly on the provided examples. Flash Fill then applies the obtained transformer on a list of n-tuples, thus essentially performing our *map* operation. Indeed, if we were supporting only simple file renamings, we could easily use Flash Fill alone for that purpose.

We now describe the other operations in StriSynth that extend the functionality of Flash Fill.

Partition divides a list of strings into groups based on the string the transformer produces. The transformer is in this case a classifier. The learning algorithm extracts common substrings for each group, and pairs up the example strings with their corresponding substrings. The result of StriSynth is a common transformer.

Filter is a special case of Partition where the output of the classifier is tested against an expected string. To learn the expected string, StriSynth looks for a common substring among the positive examples. If all negative examples are mapped to another string, the resulting transformer is returned.

Reduce: The transformer with unbounded number of inputs becomes a flattener. A natural way for the user to specify the outcome of the reduce operator is by providing a prefix of the output. To force the reduce behavior while learning the transformer, the user enters the meta-character '...' at the desired point in the output. It is up to StriSynth to find the delimiter as well as necessary transformations.

Split does the opposite of Reduce—it splits a string into a list of strings. To learn the transformer, the user provides an input string, and the corresponding substrings. The transformer returns a list of strings.

A. Implementation

Our initial prototype of StriSynth was calling an online Windows-hosted version of the proprietary Flash Fill. To deal with unbounded number of inputs, counters, number transformations and the operations described above,, we implemented our own version of the Flash Fill algorithms [6].

We wrote StriSynth in Scala, a language running on the Java Virtual Machine. Hence, StriSynth runs on both Unix and Windows. We packed StriSynth into a runnable JAR with commands in arguments to manipulate files. We also deployed this JAR to sonatype.org for reuse as a library.

We also added a server in the JAR to be able to run it as a service listening on a given port. Using this service, we wrote a Powershell script to listen to file renamings, and to offer a suggestion in English about the generalized operation that the user is about to perform.

IV. Preliminary Evaluation

For the preliminary evaluation of StriSynth, we were interested in how many examples the user needs to provide to accomplish a complex file manipulation. To find interesting file manipulations, we searched various web sources for users asking for help in generating a complex command or a script. We collected 60 scenarios: 12 come from stackoverflow.com, 18 are from unix.com, and 30 come from various other websites. Based on some specificities, e.g. the operations that need to be performed, we manually categorized the scenarios into several classes. Some scenarios fall into two or more different classes. The results of this preliminary evaluation are summarized in Table I. We ran our benchmarks on a JVM with 1Gb of RAM on an Intel Core 2.60 GHz running Windows 7. Most of the scenarios were solved in less than 1s. We bounded the learning to 23s per added example.

TABLE I
Preliminary evaluation summary.

Class	N	#	Sample transformation
Filter	**43**	3	report.doc→Ok, report.pdf→NotOk
Include	**16**	1	bar.txt → lpr bar.txt &
Ending	**8**	1	file.JPEG → file.jpg
Extraction	**12**	1	__a.txt → _a.txt
	3	2	Gof483_HD.avi→EP483.avi
Counter	**8**	2	AB123.gif→AB-0111-1.gif B3245.gif →B-0111-2.gif
Number	**3**	1	fl**1**.dat→fl1_**21001**.dat
	2	2	Img**401**.jpg→Img**001**.jpg
Properties	**10**	1	abc.log→abc.2011120706:54.log
Reduce	**9**	1	doc1.pdf doc2.pdf... doc.pdf
	1	2	s**1**_1.mp3 s1_2.mp3... song**1**.mp3
Split	**3**	2	”fa,fb,fc” 2 → fb
Date	**2**	1	110214-01.jpg→11Feb14\01.jpg
Partition	**7**	4	Godfather_CD1.avi → Godfather

The first column in the table shows the name of the scenario class. The number in the bold font in the second column denotes the number of scenarios belonging to this class. The third column denotes the number of input/output examples that we needed to derive the correct transformation. The last column illustrates a sample transformation in the corresponding scenario class.

We can see that in the majority of cases we needed only one or two examples.

A. Further Evaluation Plans

As StriSynth reaches a satisfying level of maturity, our goal is to empirically evaluate it in more detail. We plan to conduct experiments on two types of users: programmers and end users. The experiments for those two groups will be similar, but the set of benchmarks used will be different. For expert users we will collect representative tasks on technical forums and mailing lists, while the benchmark for non-expert users will correspond to their everyday tasks. The experiment will ask test subjects to first accomplish a given problem without our tool support. After completing the task, they will be asked try to solve it using StriSynth.

In addition to measuring the usability of our tool, we will also collect user feedback on how the features provided by StriSynth can be further improved.

V. Conclusion

In this paper we described StriSynth. The goal of our tool is to empower end users and help them to easily perform tasks that they could not have done otherwise. At the same time, StriSynth helps programmers execute complex tasks more efficiently, thus making them more productive and their code less error-prone. We believe that StriSynth's type of interactive synthesis could be a highly-desired feature that could improve tomorrow's use of computers.

References

[1] M. Mayer, “Video,” http://youtu.be/F9mUIPK7h-I.

[2] ——, “Video,” http://youtu.be/yaNr-JDc8tA.

[3] ——, “Video,” http://youtu.be/SRFC-Hi08-I.

[4] A. Cypher and D. Halbert, *Watch what I Do: Programming by Demonstration*. MIT Press, 1993.

[5] H. Lieberman, *Your Wish Is My Command: Programming by Example*. Morgan Kaufmann, 2001.

[6] S. Gulwani, “Synthesis from examples: Interaction models and algorithms,” *14th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, 2012, Invited talk paper.

[7] S. Gulwani, W. R. Harris, and R. Singh, “Spreadsheet data manipulation using examples,” *Commun. ACM*, vol. 55, no. 8, pp. 97–105, 2012.

[8] S. Gulwani, “Automating string processing in spreadsheets using input-output examples,” in *POPL*, 2011, pp. 317–330.

[9] R. Singh and S. Gulwani, “Learning semantic string transformations from examples,” *PVLDB*, vol. 5, 2012.

[10] A. K. Menon, O. Tamuz, S. Gulwani, B. W. Lampson, and A. Kalai, “A machine learning framework for programming by example,” in *ICML (1)*, 2013, pp. 187–195.

[11] R. Singh and S. Gulwani, “Synthesizing number transformations from input-output examples,” in *CAV*, 2012, pp. 634–651.