SmartSynth: Synthesizing Smartphone Automation Scripts from Natural Language

Vu Le
Univ. of California at Davis
vmle@ucdavis.edu

Sumit Gulwani
Microsoft Research, Redmond
sumitg@microsoft.com

Zhendong Su
Univ. of California at Davis
su@cs.ucdavis.edu

ABSTRACT

This paper presents SmartSynth, a novel end-to-end programming system for synthesizing smartphone automation scripts from natural language descriptions. Our approach is unique in two key aspects. First, it involves a carefully designed domain-specific language that incorporates standard constructs from smartphone programming platforms to balance its expressivity and the ability to synthesize scripts from natural language. Second, our synthesis algorithm integrates techniques from two research areas: (1) It infers the set of components and their partial dataflow relations from the natural language description using techniques from the Natural Language Processing community; and (2) It uses techniques from the Program Synthesis community to infer missing dataflow relations via type-based synthesis and constructs scripts in a process akin to reverse parsing. SmartSynth also performs conversational interactions with the user when multiple top-ranked scripts exist or it cannot map part of the description to any component. Evaluated on 50 tasks collected from smartphone help forums, our system produces the intended scripts in real time for over 90% of the 640 natural language descriptions obtained from a user study for those tasks. SmartSynth has also been adapted to TouchDevelop, an end user-targeted programming environment on mobile platforms, with very promising results (see `http://www.cs.ucdavis.edu/~su/smartsynth.mp4` for a video demo). We believe that SmartSynth is a step toward fully personalized use of smartphones' increasingly rich functionalities.

Categories and Subject Descriptors

D.1.2 [**Programming Techniques**]: Automatic Programming; D.3.2 [**Programming Languages**]: Language Classifications—*Design languages*; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces—*Natural language*

General Terms

Algorithms, Design, Languages, Human Factors

Keywords

Program synthesis, natural language processing, smartphone, automation script

MobiSys'13, June 25-28, 2013, Taipei, Taiwan

1. INTRODUCTION

We have entered a new era for computing in recent years with the sharp increase in availability and popularity of multi-functional mobile devices. These devices are equipped with more and more sensors, which radically multiplies the ways people interact with them. Indeed, they have led to an enormous number of novel applications and interesting challenges for programming [34] and finding personalized applications [42].

In this paper, we study the problem of developing smartphone automation scripts, those that execute some tasks automatically under certain conditions (*e.g.*, the phone turns off network connections automatically when its battery is low, or the phone sends the user's spouse a text saying "Kids are dropped" when she is at the location of their kids' school). The abundance of such scripts on popular smartphone platforms like Locale [39], Tasker [7], and On{x} [29] motivate our problem domain.

The current state of the art in end-user programming of smartphones is far from satisfactory. General-purpose languages such as Java, C#, and Objective C are clearly not suitable for most end user. The other alternative is visual programming systems such as App Inventor [30] and Tasker [7], which visualize programming constructs into building blocks. Users program by identifying blocks and composing them visually. Although these systems are more user-friendly, they still require end users to consciously think about programming like typical programmers, such as introducing variables and deciding when/how to use conditionals, loops, or events.

Because of the deluge of mobile devices and their underlying systems, we believe in the promising direction toward *general* programming systems with *natural* interfaces for meeting end users' needs. Two notable examples are VoiceActions [10] and Siri [5] for controlling smartphones via speech. However, they only provide support for limited smartphone features. Siri, for example, only matches speech to pre-defined patterns. Our goal is to be (1) *natural* so that users can interact with the system through natural language (NL) and (2) *general* so that, unlike Siri, our system does not simply match NL descriptions to pre-defined patterns.

SmartSynth To this end, we introduce SmartSynth, a new end-to-end *NL-based programming system* for synthesizing smartphone scripts. The key ***conceptual novelty*** of SmartSynth is the methodology to decompose the problem into: (1) designing a domain-specific language (DSL) to capture automation scripts users commonly need, and (2) combining natural language processing (NLP) and program synthesis [11] techniques to synthesize scripts in the DSL from NL. Our ***technical novelties*** include: (1) a carefully designed DSL that incorporates standard constructs from smartphone programming platforms and balances its expressivity and the feasibility for automatic script synthesis from NL descriptions, (2) techniques adapted from the NLP community to translate English descriptions into rele-

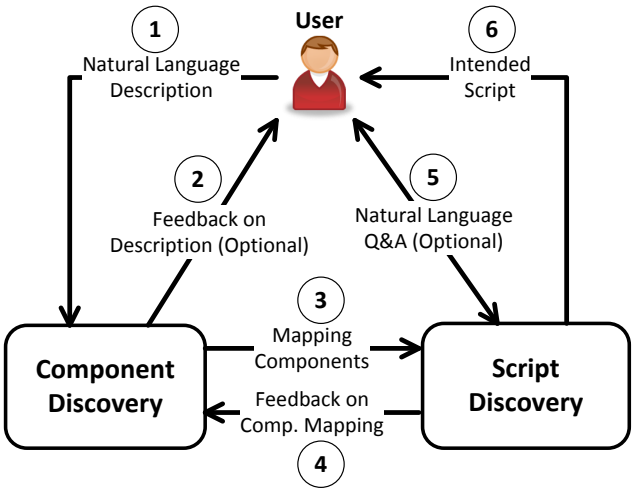

Figure 1: SmartSynth's system architecture.

vant components (*i.e.* script constructs) and a partial set of dataflow relations among the components, and (3) techniques inspired by the program synthesis community to complete the partial set of dataflow relations via type-based program synthesis [15,27,32] and construct the desired scripts via reverse parsing [2].

Indeed, we combine and refine recent advances in both the NLP and program synthesis areas. A recent, emerging trend in NLP is semantic understanding of natural languages. There have been recent, although limited, successes in translating structured English into database queries [4] and translating NL statements into logic in specific domains [9,20]. As for program synthesis, the traditional goal has been to discover new or complicated algorithms from *complete logical specifications*. There is recent, renewed interest in this area of synthesis because of (1) interesting applications (such as end-user programming [14] and intelligent tutoring systems [13]), and (2) the ability to deal with under-specifications (such as examples [14], and a set of APIs or keywords [15,24]). Our combination of NLP and program synthesis is synergistic since NLP is used to "partially understand" natural language and program synthesis is then used to refine this understanding and generate the intended script.

Figure 1 illustrates SmartSynth's process of synthesizing scripts. The user communicates her intent via natural language (Step 1). The "Component Discovery" box contains two algorithms that are inspired by the NLP area: (i) a mapping algorithm that maps the description into script components such as APIs and literals (Step 3), and (ii) an algorithm that asks the user to refine parts of the description that SmartSynth does not understand (Step 2). The "Script Discovery" box first uses an NLP-based algorithm to detect dataflow relations among the identified components. If these relations do not fully specify the dataflow relationships between components, SmartSynth invokes our algorithm (inspired by type-based synthesis) to complete the missing dataflow relations. If there are multiple equally high-ranked relations, SmartSynth initiates an interactive conversation with the user to resolve the ambiguities (Step 5). As the final step, it constructs the intended script from the identified components and relations using an algorithm akin to reverse parsing.

An interesting feature of SmartSynth is the NLP-Synthesis feedback loop in Steps 3-4. Although our NLP algorithm has its own mapping feature set to perform component mapping, it might not precisely capture the mapping in some cases, due to irregularities of natural language. SmartSynth uses program synthesis technique to gain confidence about the quality of the generated script and as

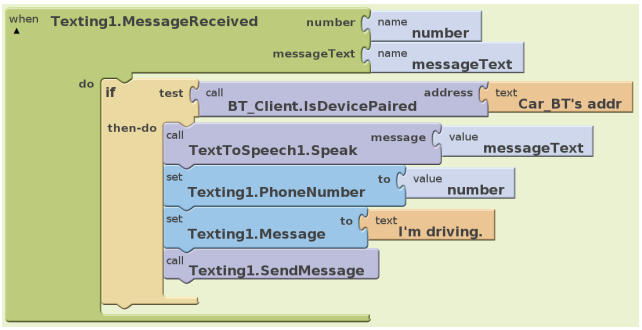

Figure 2: A visual program for Example 1 written in AppInventor.

another metric for the quality of NLP mapping. In particular, SmartSynth repeatedly requests the NLP-based algorithm for the next likely interpretation of the description, if the current interpretation is deemed unlikely (*i.e.*, the interpretation does not translate to a high-confidence script).

Contributions We make the following contributions:

- We introduce an automation language `SmartScript` that is expressive enough to represent a wide variety of automation scripts discussed on online forums, while at the same time restrictive enough to enable efficient search over the program space (Section 3).
- We present an algorithm for generating likely `SmartScript` scripts from natural language descriptions (Section 4). Our algorithm involves three key technical steps: (1) generation of script components, (2) generation of dataflow relations among those components, and (3) constructing the script from the components and dataflow relations. The first and second steps leverage techniques from NLP, while the second and third steps leverage techniques from program synthesis.
- We have implemented our technique and evaluated it on 50 different tasks collected from smartphone help forums. Our results show that SmartSynth is effective — it can generate the intended scripts in real time for over 90% of the 640 NL descriptions collected via a user study (Section 5). We have also extended SmartSynth to TouchDevelop, which has a much larger grammar and API set, with similar positive results. A video demo of the extended system is available at `http://www.cs.ucdavis.edu/~su/smartsynth.mp4`.

Paper Organization The rest of the paper is structured as follows. We first use a concrete example to motivate and illustrate SmartSynth (Section 2). Section 3 introduces our DSL `SmartScript` and defines the notion of script components. Sections 4 presents three key steps of SmartSynth— mapping NL descriptions to components, detecting dataflow relations among the identified components, and synthesizing scripts from these components and relations. We then present our detailed evaluation of SmartSynth (Section 5). We discuss related work in Section 6 and conclude in Section 7.

2. EXAMPLE

We motivate our system via the following running example, taken from a help forum for Tasker [7].

EXAMPLE 1. *The user wants to create a script to do the following when she receives an SMS while driving: (1) read the content*

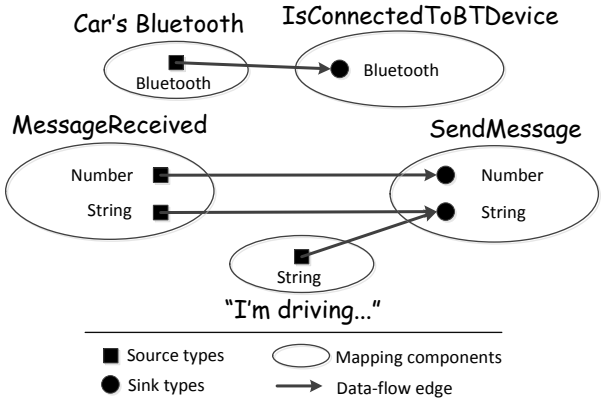

Figure 3: The graph showing all possible dataflow relations among components identified by the Component Discovery algorithms.

Description	Possible Mappings
When I receive a new SMS	`MessageReceived`

if the phone is connected to	`IsConnectedToBTDevice`
	`IsConnectedToWifiNetwork`
	`IsConnectedToDataService`

my car's bluetooth	`Car_BT`
reads out loud	`Speak`
the message content	`MessageReceived.Text`$_O$
replies	`SendMessage` `SendEmail`
the sender	`MessageReceived.Number`$_O$
"I'm driving"	`"I'm driving"`

Table 1: Possible mappings from text chunks to components in Ex. 1. `Text`$_O$ and `Number`$_O$ are return values of `MessageReceived`.

of the message and (2) send a message "I'm driving" to the sender. Since the user always connects her car's bluetooth to the phone when she is in the car, she uses this fact to denote that she is driving[1]*. One possible description of this script is:*

> "When I receive a new SMS, if the phone is connected to my car's bluetooth, it reads out loud the message content and replies the sender *"I'm driving."*."

Users of both conventional and visual programming systems have to deal with all the low-level details and make several decisions, often unintuitive to them, during the process of creating a new program. In contrast, users of SmartSynth only need to give the system their problem description in NL, and interact with it, if necessary, in natural language. Below we compare traditional programming systems with SmartSynth using our example.

Identifying Components Users of conventional and visual programming systems have to conceptualize their ideas into script components (*i.e.*, script constructs such as APIs), potentially under several refinement steps [40]. This is a non-trivial task as it assumes that users should understand all script constructs' specifications. As shown in Figure 2, users of AppInventor have to conceptualize and identify various components while transforming the idea in Example 1 into a running script.

In comparison, SmartSynth automatically decomposes the description into disjoint text chunks and matches each chunk to its supported components (details in Section 4.1). For the description in Example 1, SmartSynth is able to decompose and match the description to the components shown in Table 1. When any chunk of the description cannot be mapped to a component, SmartSynth will ask the user to refine the chunk.

However, as is often the case in NLP, mapping a chunk to components can be ambiguous. For example, we can map *"if the phone is connected to"* to any of the three candidates shown in Table 1. Similarly, we can map *"replies"* to either `SendMessage` or `SendEmail`. One can look at API arguments to resolve these ambiguities. In the first case, since the argument is a bluetooth device, it is very likely that the mapping component is `IsConnectedToBTDevice`. But this approach does not work as well for the second case, where the ambiguity can only be resolved by considering the global context that may indicate receipt or sending of an SMS.

SmartSynth takes a different approach. Instead of manually encoding many disambiguation rules, it relies on the techniques inspired by type-based synthesis to automatically disambiguate mapping candidates. Specifically, from each mapping candidate that needs to be resolved, SmartSynth generates a script and assigns it a score indicating how likely the script is. The best mapping is associated with the script that has the highest score. In Example 1, SmartSynth is able to select the right mapping set (`IsConnectedToBTDevice`, `SendMessage` and the others) because together they form the highest ranked script.

Synthesizing Scripts Besides identifying all necessary components, users of conventional systems also need to understand the components' low-level details in order to assemble them meaningfully.

To create the script in Figure 2, the users must understand that `MessageReceived` is an event and returns a phone number and a message content, which may be stored in two temporary variables. They need to understand that `IsDevicePaired` is the guard of the conditional and it must be linked with the car's bluetooth. Also, the APIs `Speak` and `SendMessage` must be configured with arguments of types `Text` and `Number` & `Text` respectively. Finally, the users need to pass the temporary variables/literals to those APIs and arrange them in the correct order.

In contrast, users of SmartSynth are not required to understand those low-level details because the system knows the signatures of all those APIs. The challenge is to generate additional script constructs such as loops, conditionals and assignments to combine these components together into a script reflecting the user's intent. SmartSynth solves this challenge in the following steps (details in Section 4.2). First, it builds a special data structure that represents all possible dataflow relations among the components. We call each of them a (dataflow) *relation*. Figure 3 shows the data structure for the right component mapping set. An edge in this figure represents a relation, which specifies a possible dataflow from a value (source) to an API's parameter (sink). The value from a source might be assigned to multiple sinks (if they have the same type), and a sink might receive a value from different sources (also of the same type). For example, the sink denoting the message content in `SendMessage` can be assigned to the argument from either the received message of `MessageReceived`, or the string literal "I'm driving" because they have the same type `String`.

Next, SmartSynth uses classic NLP techniques [19] to detect likely relations from the NL description. These relations must be derived from the set of all possible relations embedded in the graph. Table 2 shows the relations that SmartSynth has detected from the description. A row in this table represents a relation defining which source is assigned to a sink. For example, the last row states that

[1] In fact, this is a clever workaround to avoid using the GPS sensor, which drains the battery power very quickly.

Return value or Literal	API Parameter
`Car_BT`	`IsConnectedToBT-Device.Bluetooth`$_I$
`MessageReceived.Text`$_O$	`ReadText.Text`$_I$
`MessageReceived.Number`$_O$	`SendMessage.Number`$_I$
`"I'm driving"`	`SendMessage.Text`$_I$

Table 2: Relations detected from Example 1's description. Subscripts I/O characterize the field of an API as a parameter or a return value.

the message content to be sent by `SendMessage` is the string literal "I'm driving". SmartSynth generates and returns the final script to the user if it is able to detect all necessary relations from the NL description (Figure 4).

```
when (number, content) := MessageReceived()
   if (IsConnectedToBTDevice(Car_BT) then
      Speak(content);
      SendMessage(number, "I'm␣driving");
```

Figure 4: The script for Example 1.

However, since users can give free-form descriptions, SmartSynth may often detect only a *subset* of the necessary relations. In these cases where the intent is under-specified, SmartSynth uses techniques inspired by type-based synthesis to find the missing relations. It performs searching over the dataflow graph for missing relations and uses a special ranking scheme to prioritize more likely relations. SmartSynth then uses these newly discovered relations to generate the most likely script and returns it to the user.

As an example, suppose that the user had given a slightly different description *"... and send back..."* (instead of *"... and replies the sender..."*) and also suppose that SmartSynth does not model this mapping and thus is unable to extract the third relation between `MessageReceived.Number`$_O$ and `SendMessage.Number`$_I$ in Table 2. Nonetheless, SmartSynth can discover this missing relation and generate the same script as in Figure 4.

3. AUTOMATION SCRIPT LANGUAGE

In this paper, we use a scripting language `SmartScript` (Figure 5) with representative features of the current smartphone generation. Since `SmartScript` is an intermediate language, we can port it to other mobile platforms via simple syntax-directed translation [2].

3.1 Language Features

We have designed `SmartScript` from an extensive study of the scripts from various smartphone help forums. This design process is an interesting exercise to balance the trade-offs between the expressiveness of `SmartScript` and the effectiveness of SmartSynth. The restrictions that we place in `SmartScript` (event and conditionals only appear at the top of the script) allow SmartSynth to perform type-based synthesis more effectively under the uncertainties in NL processing.

A script P in `SmartScript` represents a task that executes a sequence of actions under a certain condition. It has some parameters I (which will be entered by users when the script runs) and may be triggered by an event E. When the event occurs, it generates some variables. The conversions T converts these variables into new types that are used in the condition C. The condition is then evaluated and if it holds, the main body M is executed.

$$
\begin{array}{rcl}
\text{Script } P & ::= & I\ E\ T\ C\ M \\
\text{Parameter } I & ::= & \textbf{input}(\mathbf{i_1}, \ldots, \mathbf{i_n}) \mid \epsilon \\
\text{Event } E & ::= & (\underline{r_1}, \ldots, \underline{r_n}) := \textbf{when}\ \underline{\texttt{Event}}() \mid \epsilon \\
\text{Conversions } T & ::= & F_1; \ldots; F_n; \\
\text{Condition } C & ::= & \textbf{if}\ (\Pi_1 \wedge \cdots \wedge \Pi_n)\ \textbf{then} \\
\text{Body } M & ::= & Stmt_1; \ldots; Stmt_n; \\
\text{Conversion } F & ::= & \mathbf{x} := \texttt{Convert}(a) \\
\text{Predicate } \Pi & ::= & \underline{\texttt{Predicate}}(a_1, \ldots, a_n) \\
\text{Stmt. } Stmt & ::= & S \\
 & \mid & \textbf{foreach}\ \mathbf{x} \in a \\
 & & \textbf{do}\ S_1; \ldots; S_n;\ \textbf{od} \\
\text{Atom. Stmt. } S & ::= & A \mid F \\
\text{Action } A & ::= & (\underline{r_1}, \ldots, \underline{r_n}) := \underline{\texttt{Action}}(a_1, \ldots, a_n) \\
\text{Argument } a & ::= & \mathbf{x} \mid \mathbf{i} \mid \underline{r} \mid \underline{l}
\end{array}
$$

Figure 5: The syntax of automation language: $\mathbf{x}$, $\mathbf{i}$, $\underline{r}$, $\underline{l}$ refer to a temporary variable, an argument of the script, a return value, and a literal, respectively. The essential components identified by the NLP techniques are underlined.

We use a few examples, each of which contains an NL description and the synthesized script, to illustrate our key language constructs.

Event and Conditional

EXAMPLE 2. [Phone Locator] *When the phone receive a new text message, reply with my current location if the message content is "Secret code".*

```
when (number, text) := MessageReceived
   if (text = "Secret␣code") then
      text2 := LocationToString(CurrentLocation);
      SendMessage(number, text2);
```

API Composition

EXAMPLE 3. [Picture Uploader] *Take a picture, add to it the current location and upload to Facebook.*

```
pic := TakePhoto();
text := LocationToString(CurrentLocation);
pic2 := AddTextToPhoto(pic, text);
UploadPhotoToFacebook(pic2);
```

Loops

EXAMPLE 4. [Group Texting] *Send my current location to 111-1111 and 222-2222.*

```
text := LocationToString(CurrentLocation);
foreach number in {111-1111, 222-2222} do
   SendMessage(number, text);
od
```

3.2 Essential Components

Script components (constructs) in `SmartScript` are not equally important. A component is essential to the semantics of the script if we cannot reconstruct the script once we have removed the component from it. On the other hand, a component is not essential if we are still able to reconstruct the script if the component is removed. In order to successfully generate the intended script, SmartSynth must be able to identify all essential components from an NL description.

In Figure 5, we underline all `SmartScript`'s essential components. An essential component in `SmartScript` is either an API or an entity. While APIs are pre-defined and related to smartphone functionalities, entities correspond to personal data. This mixture enables the easy creation of personalized automation scripts.

APIs We have created a representative set of 106 APIs that cover most of the available functionalities of the current generation of smartphones. We annotate each API with a set of weighted tags (used by the mapping algorithm to map text chunks to appropriate components), and a type signature (used by the synthesis algorithm to generate only type-safe scripts). The tag set is weighted since some tags are more indicative of a component than others.

We classify the APIs into the following categories to match the constructs in `SmartScript`:

- **Events:** These represent events that occur on the phone. For example, the event of receiving a new text message (`MessageReceived`), or the event of receiving a phone call (`IncomingCall`). These events trigger script execution.
- **Predicates:** These denote conditions on the state of the phone. For example, there are text messages that the user has not read (`HasUnreadMessages`), or the phone is in landscape orientation (`IsInLandscapeMode`). They also include comparison operators (*e.g.*, $<$, $>$, and $=$). Predicates restrict the condition under which the script is executed.
- **Actions:** These are normal phone operations, such as turning off wifi or bluetooth connection (`TurnWifiOff`, `TurnBluetoothOff`), muting the phone (`MutePhone`), taking a picture (`TakePhoto`), or sending a text message (`SendMessage`). SmartSynth executes a sequence of actions in a script's body when its event is triggered and the predicates are satisfied.

Entities Entities represent values that are passed among APIs inside a script. An entity is one of the following:

- **Return values:** These are return values of our representative APIs. For example, the content of the received message ($\texttt{MessageReceived.Text}_O$), the caller of incoming call ($\texttt{IncomingCall.Number}_O$).
- **Literals:** These are personal data items from the user's phone (*e.g.*, a contact in her address book, her bluetooth headset, her music collection, her current location) or generic values (*e.g.*, a string "I'm driving", the time 10pm).

The purpose of identifying APIs is to form the script's skeleton, while that of capturing entities is to prepare for building the dataflow relations among these identified APIs in a later step. We discuss the details of our algorithm next.

4. SYNTHESIS ALGORITHM

The key observation behind our synthesis algorithm is that a `SmartScript` script can be constructed from its constituent set of essential components and the dataflow relations among those components. This observation allows us to reduce our original problem of synthesizing a `SmartScript` script from an NL description to the following three sub-problems: (a) identifying essential components (Section 4.1), (b) identifying their dataflow relations (Section 4.2.1), and (c) constructing the script from the components and their dataflow relations (Section 4.2.2). We use techniques inspired from both the NLP and program synthesis communities to solve these sub-problems.

We first use an algorithm inspired by NLP techniques to map the given NL description to a set of essential components. We then use rule-based relation detection algorithm (a classic NLP technique) to extract dataflow relations among the identified components. Since this algorithm may not be able to extract all relations, we use a technique inspired by type-based synthesis to find the likely missing relations. Finally, we use the components and their relations to construct the script and return it to the user. We explain these algorithms in detail next.

4.1 Component Discovery

We assume that for every NL description, there exists a decomposition of that description into disjoint text chunks, where each of the text chunks can be mapped to a component. It is expected that there are many possible ways to decompose a given description into chunks. Furthermore, each of the chunks can be mapped to many possible components. Thus, the space of all possible mappings from an NL description to a set of components is large. ***The key high-level insight*** of our component discovery algorithm is to consider all possible mappings (to gain high recall) in a ranked order (to gain high precision).

To this end, we have developed an effective data structure that efficiently stores all possible description decompositions, and the supportive and refutative features used to map these decompositions into components. We then find the best component mapping from this data structure.

Mapping Data Structure Our data structure is a weighted directed graph $\langle V, E \rangle$ where

- V is a set of vertices. Each vertex represents a *cut* between two consecutive words in the description.
- E is a set of edges. An edge (v_i, v_j) denotes a chunk of text that starts from the vertex v_i and ends at the vertex v_j.

For each edge in E, we need to calculate the confidence of mapping its text chunk to a particular component. We use mapping features to calculate this score.

Mapping Features A *feature* h is a function that returns the confidence score of mapping an edge in our data structure to a component. In other words, it indicates how likely a text chunk (represented by the edge) is mapped to a certain component, among all possible components.

Currently, our system uses the following features:

- **Regular expressions:** We use a set of regular expressions to extract entities. A chunk is likely mapped to an entity if it matches the corresponding regular expression. For each entity that is an API's return value, we create a set of common terms that people might call it. For example, we associate $\texttt{MessageReceived.Text}_O$ with "message content", "received content", and "received message". To extract literal entities (*e.g.*, the time "10pm", the number "123-4567"), we define regular expressions for all data types used in SmartSynth.

- **Bags-of-words:** We use the bags-of-words model [19] to categorize a chunk as representing some APIs. A chunk is more likely mapped to an API if it contains more words that are related to the API's tags. For example, it is more likely to map "send a text" to `SendMessage` than to map the single word "send" to the same API, because the tag set of `SendMessage` contains both "send" and "text".

- **Phrase length:** This feature gives a negative score to a mapping (between a chunk and a component) if the chunk is either too long or too short.

- **Punctuation:** A punctuation is an indication of the start or end of a chunk. Therefore, chunks that start or end with punctuations have higher scores.

- **Parse tree:** Although NLP parsers may not provide precise parse trees, they are still useful because these parse trees are partially correct, and we can exploit them to gain higher mapping confidence. For example, if a parse tree indicates that a text chunk is a noun phrase, it increases the confidence of mapping this chunk to entities. Also, if the main verb in a chunk matches a verb in an action API, it is more likely to map the chunk to this API. In our implementation, we use the Stanford NLP parser [21].

The above features give different scores for mapping a text chunk to a component. The *aggregate function* combines these scores into a single score. In our implementation, the aggregate function is a weighted sum of all feature scores. It is formally defined as follows:

$$f(e, c) = \sum_{h_i \in F} w_i * h_i(e, c)$$

where e is the edge that contains the text chunk, c is the component that e maps to, F is the feature set, w_i is the feature weight.

For each edge, we select the mapped component whose aggregate score is the highest. This score becomes the edge's weight in the data structure.

$$mapping(e) = \{\langle c, f(e, c)\rangle \mid c \in \operatorname*{argmax}_{c \in C} f(e, c)\}$$

where argmax returns the components that maximize the value of the function f.

We store all related information in the data structure to make a *global* decision based on all the features. The next step is to extract the best mappings for the whole NL description from the data structure.

Finding the Best Mapping A path from the start node to the end node decomposes the descriptions into several disjoint text chunks. Each chunk is mapped to a component with a confidence score represented by the weight of its corresponding edge. The total confidence score of mapping the whole description to a component set is the total weight of the corresponding path. Thus, the best mapping (which has the highest total confidence score) corresponds to the longest path in the graph.

4.2 Script Discovery

Once SmartSynth has discovered the set of essential components, it uses the following ingredients to synthesize the intended script: (1) type signature of the components, (2) structural constraints imposed by the automation script language, (3) spatial locality of the components in the natural language description, and (4) an effective ranking scheme.

C_1	**typeof** (C_2)	**typeof** (C_3)	**Relations**
`IsConnected-ToBTDevice`	`BT`		$\langle C_2, C_1.\texttt{BT}_I\rangle$
`ReadText`	`Text`		$\langle C_2, C_1.\texttt{Text}_I\rangle$
`SendMessage`	`Number`	`Text`	$\langle C_2, C_1.\texttt{Number}_I\rangle$ $\langle C_3, C_1.\texttt{Text}_I\rangle$

Table 3: Some rules to detect relations in Example 1. C_1, C_2, C_3 is a sequence of identified components.

The overall process works as follows. Using typing information, SmartSynth builds a dataflow graph to capture all possible dataflow relations among the provided components. It uses a classic NLP technique to extract dataflow relations from the relative orders of the identified components. If the set of extracted relations is *incomplete*, SmartSynth utilizes the dataflow graph and a ranking scheme to identify the most likely missing relations. SmartSynth then constructs the intended script from the identified components and relations via a process akin to reverse parsing, and returns the script to the user. Next, we discuss these steps in detail.

4.2.1 Dataflow Relations Discovery

Dataflows are essential in determining the intended script. If components are building bricks, dataflows are the cement to glue the bricks together to form scripts. ***The key insight*** in this process is to use an NLP technique to detect a (partial) set of (high-confidence) dataflow relations followed by a type-based synthesis technique to complete this set using type signatures of components and a ranking scheme.

DEFINITION 1 (DATAFLOW RELATION). *A* dataflow relation *is an ordered pair of a* source *(*i.e.*, an API return value or a literal) and a* sink *(*i.e.*, an API parameter), representing a value assignment from the source to the sink.*

Dataflow Graph We build the graph $G_{\mathcal{C}} = \langle V, E\rangle$ that concisely captures all dataflow relations among the identified components $\mathcal{C}$ as follows:

- V is the set of all sources and sinks in $\mathcal{C}$.

- E is the set of all directed dataflow edges from sources to sinks, such that they are type-compatible. Each edge in E represents a dataflow relation between a source and a sink.

A source type τ_1 is type-compatible with a sink type τ_2 if they have the same type or τ_1 is a list of type τ_2 (we use the later case to generate loops).

The task of SmartSynth is to determine which relations are most relevant in $G_{\mathcal{C}}$. It does this in two steps. First, it detects dataflow relations among components from their relative locations in the description. When necessary, it uses type-based synthesis techniques to infer additional relations from the dataflow graph.

Detecting Partial Dataflow Relations We use rule-based relation detection algorithm, a standard NLP technique, to detect dataflow relations among the identified components. Since all of our APIs are pre-defined and structured (each API is annotated with its type signature), we are able to manually compile a set of rules to extract relations based on the order of how components appear in the sentence. Specifically, if an API component is followed by entities having types that match its signature, those entities are likely to be the API's arguments.

Algorithm 1: Discovering the most likely complete dataflow set

Input : component set $\mathcal{C}$, partial relation set $\tilde{R}$
Output: the most likely complete dataflow set R_{top}

1 **begin**
2 $G_{\mathcal{C}} \leftarrow$ `BuildDataFlowGraph`$(\mathcal{C})$
3 $T_{all} \leftarrow \{t \mid t$ is sink in $G_{\mathcal{C}}.V\}$
4 $T_{conc} \leftarrow \{t \mid \langle s,t\rangle \in \tilde{R}\}$
5 $\tilde{T} \leftarrow T_{all} \setminus T_{conc}$ `// to be concret'd sinks`
6 $\mathcal{R} \leftarrow \{$`ArbitraryRelationSet`$(\tilde{R})\}$
7 $\mathcal{Q} \leftarrow \{\langle \tilde{R}, \tilde{T}\rangle\}$
8 **while** $\mathcal{Q} \neq \{\}$ **do**
9 $\langle R, T\rangle \leftarrow \mathcal{Q}$.`Dequeue()`
10 **if** `MinCost`$(R) \leq$ `MaxCost`$(\mathcal{R})$ **then**
11 **if** $T = \emptyset$ & `GenCode`$(R) \in$ *SmartScript* **then**
12 **if** `Cost`$(R) <$ `MaxCost`$(\mathcal{R})$ **then**
13 $\mathcal{R} \leftarrow \{R\}$
14 **else if** `Cost`$(R) =$ `MaxCost`$(\mathcal{R})$ **then**
15 $\mathcal{R} \leftarrow \mathcal{R} \cup \{R\}$
16 **else** `/* explore child spaces */`
17 $t \leftarrow$ `choose`(T)
18 **foreach** $(s,t) \in G_{\mathcal{C}}.E_v$ **do**
19 $\mathcal{Q}$.`Enqueue`$(\langle R \cup \langle s,t\rangle, T \setminus t\rangle)$
20 **else** ; `/* prune this branch */`
21 **if** $\mathcal{R} = \emptyset$ **then** $R_{top} \leftarrow \perp$
22 **else if** $\mathcal{R} = \{R\}$ **then** $R_{top} \leftarrow R$
23 **else** $R_{top} \leftarrow$ `PerformConversation`$(\mathcal{R})$
24 **return** R_{top}

Table 3 shows three rules for detecting all relations in Example 1. The first rule states that if the entity following `IsConnectedToBT-Device` has type `BT`, there is a relation between those two.

We remark that these rules can be learned automatically using supervised learning [19, 20]. However, in a manageable domain like ours, expert rules work quite well. The Stanford NLP parser also uses hand-written rules to extract typed dependencies (grammatical relations between words) [28].

Even though our algorithm works fairly effectively, it is unable to detect all relations within a description, as is typically the case with NLP algorithms. In these cases, SmartSynth relies on recent advances in modern program synthesis to complete the relation set and generates the final script. Without the synthesis algorithm, SmartSynth would halt here and ask the user to refine the description despite that it might be entirely correct.

Completing the Relation Set The goal here is to discover missing relations to form a complete *dataflow set* that can be turned into a script in a later phase. We use a specialized ranking scheme to find the most likely complete dataflow set among all possible sets.

DEFINITION 2 (COMPLETE DATAFLOW SET). *A set $G_{\mathcal{C}}$ of dataflow relations among components in $\mathcal{C}$ is* complete *if every sink of a component in $\mathcal{C}$ has one and only one relation associated with it. This dataflow set fully specifies how arguments are passed to all the sinks in $\mathcal{C}$.*

Algorithm 1 describes the process of completing the dataflow set. At the high-level, SmartSynth performs search over the dataflow graph to find the best dataflow sets $\mathcal{R}$ for the sinks $\tilde{T}$ that have not been assigned any values (line 5). SmartSynth uses the ranking scheme (represented by `Cost`) to guide the search towards more likely relations (lines 12, 14). It enforces structural constraints imposed by `SmartScript` by eliminating dataflow sets that do not conform to the language (line 11). To speed up the discovery process, our algorithm implements the general branch-and-bound algorithm. It ignores a search sub-space if the minimum cost (`MinCost`) of all dataflow sets in that space is greater than that of the current set (lines 10 and 20).

One important aspect of Algorithm 1 is our specialized ranking scheme, which helps prioritize the search process toward the script that most likely matches the user's intention. We next discuss the ranking scheme.

Algorithm 2: Script Construction

Input : component set $\mathcal{C}$, dataflow set R
Output: an automation script

1 **begin**
`/* generate temporary variables */`
2 **foreach** *source* $\tau \in \mathcal{C}$ **do**
3 `SymTable`$[\tau] \leftarrow$ new variable
`/* assign sources' vars to sinks */`
4 **foreach** $(\tau_s, \tau_t) \in \mathcal{C}.E$ **do**
5 `SymTable`$[\tau_t] \leftarrow$ `SymTable`$[\tau_s]$
6 $I \leftarrow$ live input variables in $\mathcal{C}$
7 $E \leftarrow \perp$
8 **if** $\exists e \in \mathcal{C} : e$ *is event* **then** $E \leftarrow e$
9 $C \leftarrow \bigwedge_i \Pi_i$ where Π_i is a predicate in $\mathcal{C}$
10 $T \leftarrow$ conversion functions used by C
11 $M \leftarrow$ the rest of actions and conversion functions in g
`/* generate the foreach loop */`
12 **foreach** $(\tau_s, \tau_t) \in g.E_v$ s.t. $\tau_s = List\langle\tau_t\rangle$ **do**
13 $s \leftarrow$ the subgraph reachable from τ_t
14 `SymTable`$[\tau_t] \leftarrow$ new variable
`/* replace` s `by foreach stmt */`
15 $M \leftarrow M[$`(foreach SymTable`$[\tau_t] \in$ `SymTable`$[\tau_s]$
16 `do` s `od)` $/ s]$
17 $P \leftarrow \langle I, E, T, C, M\rangle$
18 **return** *P*

Ranking Scheme Our high-level insight behind the ranking scheme is to give preferences to those scripts (1) that have greater coupling among the components, (2) that are more generally applicable, or (3) that require fewer user inputs.

We define a set of (negative) cost metrics to realize the above insight, each of which defines a less likely or less preferred feature of a script. The `Cost` function is simply the sum of the costs given by each of them:

- **Unused Variable Cost**: This metric gives a unit cost to each unused variable generated by SmartSynth. It promotes scripts that have better coupling among components.
- **Parameter Cost**: This metric gives a unit cost for each of parameters the script uses. This is because users normally do not want to configure the script when it runs. They often want fully automatic scripts.
- **Domain-Specific Cost**: This metric contains expert rules that capture what people usually do in practice. For example, in the messaging domain, we punish the script if the replied message is the same as the content of the received message.

Algorithm 3: The overall algorithm of SmartSynth.

```
   Input  : NL description d
   Output : script P, or request to refine (parts of) d
1  begin
2      G_d ← BuildDataFlowGraph(d)
3      if G_d is connected then
4          C ← BestMapping(G_d)
5          R̃ ← DetectRelations(C)
6          while MappingScore(⟨C, R̃⟩) < δ do
               /* mapping score too low */
7              C ← NextBestMapping()
8              R̃ ← DetectRelations(C)
9          if C ≠ ∅ then                /* mapping exists */
10             R ← CompleteRelations(⟨C, R̃⟩)
11             if R > 1 then
                   /* equally ranked sets */
12                 R ← QuestionAnswering(R)
13             else R ← R.First()
14             return GenerateCode(⟨C, R⟩)
           /* refine the whole desc.  */
15         return AskForRefinement(d)
16     else          /* refine nonmappable phrases */
17         l ← DisconnectedSubGraphs(G_d)
18         return AskForRefinement(l)
```

This ranking scheme work quite effectively for our problem domain. However, in the presence of a large amount of user data, we can build a probabilistic ranking model. We are investigating this option for the TouchDevelop domain from its thousands of scripts contributed by its users.

4.2.2 Script Construction

At this point, SmartSynth has found a set of components and a complete set of dataflow relations. It now uses Algorithm 2 to construct the intended script. This process is the reverse of the normal parsing process in a compiler. A compiler breaks down a program into components and dataflow relations to enable optimizations and low-level code generation. In contrast, the SmartSynth synthesizes the script structure from its building pieces, namely the components and their dataflow relations. During this synthesis process, SmartSynth needs to: (1) generate code to pass arguments to APIs, (2) generate conditionals, and (3) generate loops.

SmartSynth generates a temporary variable for each API return value, and, based on the dataflow set identified in the previous steps, passes these variables as arguments to other APIs (line 2-5). Next, it extracts the event and conditionals from the event and predicate components to form the guard of the script (lines 8-9). The remaining action APIs form the body.

The ***most interesting aspect*** of the script construction process is loop generation. SmartSynth generates a looping construct if there is a relation from a collection to an API parameter (lines 12-16). For example, the user may say "send Joe and John my current location." Although the user does not explicitly mention the word "loop", it would be her intention to use a loop. In particular, SmartSynth detects that there is a relation between the group {Joe, John} and `SendMessage.Number`$_I$, and generates a loop to send messages to both Joe and John.

4.3 SmartSynth

Having described the two technical components (namely Component Discovery and Script Discovery), we now ready to formally describe the rest of SmartSynth's architecture in Figure 1.

Algorithm 3 implements all the components in the architecture. It realizes our observation that a `SmartScript` script can be identified from its essential components and their dataflow relations. Specifically, it uses NLP techniques to find components from the provided NL description (line 4) and to detect some of relations (line 5). It then invokes the synthesis algorithm to complete the dataflow relations (line 10), and to synthesize the final script in line 14. If nothing goes wrong, those steps correspond to the simplest possible workflow in SmartSynth.

However, since it is dealing with end users and natural language, SmartSynth faces many uncertainties. For example, users might give irrelevant and/or ambiguous descriptions. We discuss SmartSynth's error handling mechanism next.

Handling Descriptions with Unsupported Functionalities Due to their high expectation (and sometimes ignorance), users might ask for functionalities that are not supported by SmartSynth.

If the given description is in-scope, there exists a meaningful decomposition of the description into several chunks, and SmartSynth is able to map each of these chunks to a component. The graph in this case is a *connected* graph. On the other hand, if one or more parts of the description are out-of-scope, these parts cannot be mapped to any component(s). The graph in this case is *disconnected*. SmartSynth asks the user to refine the phrases corresponding to the disconnected subgraphs that cannot be mapped to anything (line 17 in Algorithm 3, step 2 in Figure 1). Figure 6 visualizes this case.

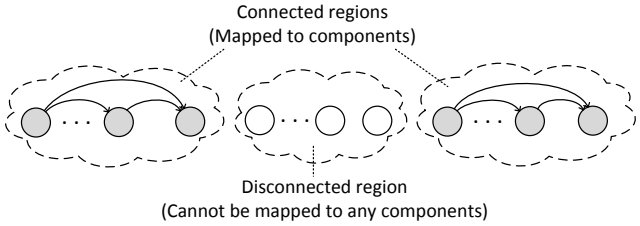

Figure 6: A region in the description that cannot be mapped to any component. SmartSynth reports its phrase to the user to clarify.

Handling Ambiguous NL Descriptions Although SmartSynth uses multiple mapping features to increase the precision of the component mapping process, there are cases where these features cannot promote the best component mapping properly. This is unavoidable because we cannot precisely model natural language (due to its irregular nature).

SmartSynth resolves such cases via the feedback loop on lines 6-8 in Algorithm 3 (and Steps 3-4 in Figure 1). SmartSynth evaluates the quality of the component mapping provided by the mapping algorithm and repeatedly generates other interpretations of the NL description, if the current one is unsatisfactory.

Handling Equally Likely Scripts When the ranking scheme cannot differentiate the top dataflow sets, SmartSynth has multiple equally ranked scripts at the top. In such cases, SmartSynth performs interactive conversation with the user to eliminate undesired scripts (line 12 in Algorithm 3, and Step 5 in Figure 1). It generates a sequence of *distinguishing* multiple-choice questions based on these candidates and presents them to the user, one question at a time. Her

answers help SmartSynth capture more constraints and eliminate unintended dataflow sets.

The intuition behind the algorithm is that each question corresponds to a sink that receives different values in those dataflow sets, and the answer choices are those values that can be assigned to the sink. When the user selects an answer, SmartSynth eliminates those dataflow sets that do not correspond to the selected value.

Because the sink set is finite and known *a priori* (sinks are parameters of pre-defined APIs), we are able to build a database that stores an NL question for each of the sinks. Similarly, we define NL representations for all the sources. SmartSynth generates questions and answers by looking up the values of the corresponding sinks and sources in the database. For example, for the sink `SendMessage.Number`$_I$, SmartSynth generates the question *"Who do you want to send the SMS to?"*. Suppose this sink is related to two sources, `MessageReceived.Number`$_O$ and `111-1111`. They will be presented as two answer choices *"(A) The sender of the received SMS"* and *"(B) The phone number 111-1111"*.

5. EVALUATION

In this section, we evaluate the effectiveness of SmartSynth in discovering the right set of components and their dataflow relations. We also compare the overall performance of SmartSynth and the system that employs only NLP techniques. We implemented SmartSynth in C#. All our experiments were run on a machine with an Intel® i7 2.66 GHz CPU and 4 GB of RAM. We use the Stanford NLP parser [21] wrapped under a public web service [1].

We compile the benchmark from task descriptions found in help forums of Tasker [7], App Inventor [30], and TouchDevelop [38]. Among the 70 found descriptions, seven cannot be expressed in our language `SmartScript`. Therefore, we remark that `SmartScript` was able to express 90% of the kind of real-life examples that it was designed for. All the inexpressible descriptions that we found required synchronization between different tasks, possibly through the use of some shared global variables or control-flow among events. To illustrate, we describe below one of them; the other ones are similar.

EXAMPLE 5. [Inexpressible in `SmartScript`] *If there is an unread SMS and the user has not turned the screen on, remind her every 5 minutes.*

This script has two synchronized tasks, which may be synchronized using a global status variable denoting whether or not the screen has been turned on. We plan to extend our language to allow task synchronization as future work.

Of the remaining 63 task descriptions that are expressible in `SmartScript`, 13 are similar to another one in the set. Table 4 shows the distinct 50 tasks. To collect NL descriptions for those benchmark tasks, we conducted an online user study involving freshman students. We provided a one page document describing their task. We divided the benchmark into two sets that are roughly equivalent in terms of complexity. The system randomly assigned a student to one set. For each problem in an assigned set, we gave the student its desired script in `SmartScript` as the specification, and asked the student to give several NL descriptions that match the given script.

Eleven students participated in the study, giving a total of 725 distinct descriptions for our 50 benchmark tasks. Among these given descriptions, 640 match their respective specifications. We used these 640 descriptions to evaluate SmartSynth.

In Table 4, column "Compnts" shows, for each task, the total number of components that are identified from the description. The next two columns divide these components into APIs and entities. Column "Relations" corresponds to the number of dataflow relations for each task, and "Detected Relations" shows how many relations on average the NLP-based algorithm can extract from various descriptions. The last two columns show the average parsing time by the Stanford NLP parser and the average total synthesis time under ten runs. Note that the parsing time includes the communication overhead to the web service provider. It dominates the total time by SmartSynth.

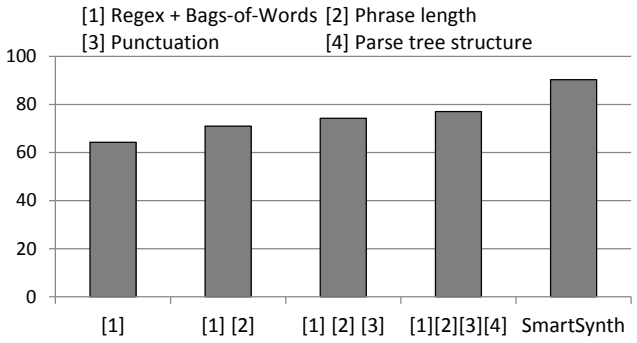

Figure 7: The result of mapping phrases to components under increasingly sophisticated configurations.

Component Discovery We first evaluate the effectiveness of SmartSynth in mapping descriptions into components. Figure 7 shows the precision of this process under increasingly sophisticated configurations. The baseline algorithm (regular expression for entities and bag-of-words for APIs) produces the right component set 64.3% of the time. The algorithm combining all features achieves a 77.1% precision (the fourth column). These results indicate that the considered features work harmoniously to improve the overall precision.

Component Discovery (with Feedback Loop) Since mapping from descriptions to components tends to be ambiguous, SmartSynth utilizes the feedback from the synthesis algorithm (see Section 4.3) to gain higher mapping confidence. The last column of Figure 7 shows that the aid from the feedback loop helps improve the mapping precision to 90.3%. This proves that the synthesis feedback is very effective in disambiguating mapping candidates. Note that we measure precision by counting the number of sets that are mapped *entirely* correctly, while in other settings, *individual* components are counted. Their performance might be reduced when measured in our context.

Dataflow Relation Detection (with NLP Technique) We now evaluate the capability of SmartSynth in finding dataflow relations among APIs and entities under the ideal case, *i.e.* when the components are mapped correctly. Figure 8 shows some statistics on the number of dataflow relations that the rule-based relation detection algorithm can detect on benchmark tasks that are grouped by the number of relations.

These results indicate that the rule-based relation detection algorithm works quite effectively for tasks that have few relations. Specifically, it found almost all relations in the 1-relation tasks and nearly 86% relations in those having four. However, the rule-based algorithm faces problems when the tasks contain more relations. In particular, it could not detect nearly half of the relations in tasks having 7 or more relations. In order to resolve this, SmartSynth needs further support. Next, we measure the effectiveness of the

Test	Brief description of the benchmark	Compnts	APIs	Entities	Relations	Detected Relations	t_{NLP} (ms)	t_{total} (ms)
1	Reply current location, read sender's number and msg's content	12	6	6	7	4.13	3,151	3,325
2	Reply current loc. and batt. level upon receiving a secret code	11	5	6	7	3.75	2,705	3,206
3	Phone locator by SMS	10	5	5	8	4.6	4,376	5,737
4	Text a friend when come nearby	8	3	5	4	2	1,102	1,137
5	Silent at night but ring for important contacts	8	3	5	4	3.14	2,544	2,617
6	Speak weather in the morning	7	3	4	3	1.67	388	405
7	Take a picture, add the location and upload to Facebook	7	4	3	4	3	837	875
8	Auto response to SMS at night	7	3	4	4	3.61	2,071	2,134
9	Alert and turn off connections when battery is low	7	6	1	1	1	2,402	2,878
10	Text wife when nearly finish a long commute	6	2	4	3	2.94	1,439	1,481
11	Send SMS at a particular time	6	2	4	3	2.60	1,173	1,210
12	Send an email when leaving the office	6	2	4	3	2.67	1,077	1,108
13	Send a message when kids are dropped at school	6	2	4	3	2.88	1,644	1,689
14	Send to a list of friends current location every 15 minutes	6	3	3	4	4	788	817
15	Reply busy message if connected to car's bluetooth	6	3	3	3	2.19	2,723	2,811
16	Turn off GPS when connected to home wifi	5	2	3	1	1	737	779
17	Increase display timeout while running some apps	5	2	3	3	2	948	983
18	Auto answer calls when headset is connected	5	3	2	1	1	1,339	1,395
19	Block calls from a group of users	5	2	3	2	1.42	715	744
20	Connect to home wifi when disconnected from car's bluetooth	5	2	3	2	2	1,234	1,292
21	Find direction from current loc. to the place a photo is taken	5	3	2	3	2.07	759	787
22	Turn off wifi and GPS at night	5	3	2	2	2	653	688
23	Ring loudly for important contacts	5	2	3	3	3	1,090	1,128
24	Lower volume when a loud friend is calling	5	2	3	3	3	1,539	1,584
25	Take a picture and add current time to it	5	3	2	3	1.88	486	507
26	Handsfree texting	5	3	2	2	1.36	1,246	1,290
27	Broadcast the received message from a group	5	2	3	4	3	827	859
28	Read the received message while connected to car's bluetooth	5	3	2	2	1.90	1,122	1,176
29	Mute the phone if is in the theater	4	2	2	1	0.89	467	485
30	Disable screen rotation at night	4	2	2	2	2	413	436
31	Mute the phone if is in meeting	4	2	2	1	0.92	344	359
32	Turn on data service for apps that require data	4	2	2	2	1.76	1,501	1,547
33	Turn on GPS when apps requiried GPS are run	4	2	2	2	1	752	781
34	Send current location to a friend via SMS	4	2	2	3	2	272	284
35	Send an SMS with a secret code to trigger an alarm	4	2	2	1	1	912	947
36	Repeat caller name	4	3	1	2	1	441	459
37	Alert when receive an email with important subject	4	2	2	1	1	759	788
38	No text while driving	4	2	2	2	1.55	1,824	1,879
39	Ask to take the call if connected to car's bluetooth	4	3	1	1	1	1,702	1,776
40	Launch Pandora when the headphone is plugged	3	2	1	1	1	310	327
41	Reduce the volume when headset is plugged in	3	2	1	1	1	361	379
42	Keep screen awake when using the keyboard	3	2	1	1	1	414	434
43	Maximize screen brightness for calls	3	2	1	1	1	406	424
44	Turn off ringer by turning the phone down	3	3	0	0	0	718	757
45	Lay the phone down to switch to speaker	3	3	0	0	0	792	827
46	Show direction from current loc. to previously saved loc.	3	2	1	2	2	661	696
47	Open the keyboard and start texting	2	2	0	0	0	199	211
48	Unplug headset to pause media player	2	2	0	0	0	219	231
49	Play alert sound when battery is full	2	2	0	0	0	221	233
50	Send a text message to a group	2	1	1	3	2	387	402

Table 4: Characteristics of the benchmark set extracted from smartphone help forums.

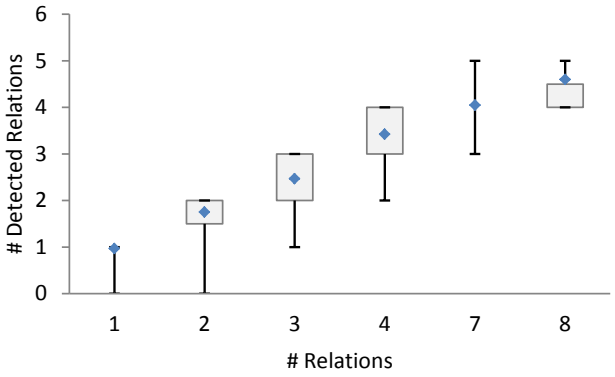

Figure 8: The number of dataflow relations detected by the rule-based algorithm for various tasks grouped by relation number.

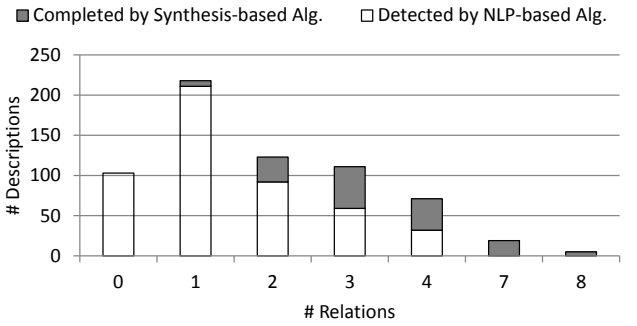

Figure 9: The number of descriptions whose dataflow relations are entirely detected by the rule-based algorithm (total 487) vs. those that require completion by the completion algorithm (total 153), grouped by the number of relations.

synthesis-based algorithm in assisting the rule-based algorithm to complete those missing relations.

Dataflow Relation Completion (with Synthesis Technique) Figure 9 shows that in nearly a quarter of the cases, the rule-based algorithm failed to detect all the dataflow relations in the solution set. Furthermore, the failure rate is proportional to the number of dataflow relations in the task descriptions. These indicates that pure NLP techniques do not scale well with complicated tasks. By employing the relation completion algorithm inspired by program synthesis area, SmartSynth was able to complete the dataflow sets and synthesize the desired scripts. In general, the synthesis-inspired algorithm is more useful with more complicated scripts, where the failure rate of detecting all relations using only NLP techniques is close to 100%.

Overall Performance Should SmartSynth only use NLP techniques and not employ the synthesis-inspired algorithm, it would suffer from the ambiguity of mapping descriptions to components and the uncertainty of extracting their relations. As a result, it might not be able to generate the user's intended script. Evaluated on our data, such system only returns the intended script 58.7% of the time (mostly with simple descriptions).

With the appearance of techniques inspired by program synthesis area, SmartSynth both improves its component identification (from the feedback loop) as well as its relation discovery (from additional relations found using the relation completion algorithm). SmartSynth achieves the 90% accuracy by combining the strength of the two research areas.

SmartSynth works quite well once it combines the strength of both NLP and synthesis communities. However, it might fail to generate the intended script sometimes. We discuss such cases next.

Discussion Given the NL description is grammatically correct and in-scope, there are two possibilities for SmartSynth to fail in generating the intended script. The first possibility is when users ask for a script that is not expressible in `SmartScript`. When designing `SmartScript`, we carefully balanced the trade-offs between language expressiveness and synthesis effectiveness. We can extend `SmartScript` to cover more scripts, but this also expands the search space, and may make the completion algorithm less scalable and the ranking scheme less effective. Nonetheless, `SmartScript` is able to express 90% of automation scripts that users care about (see Section 5).

The second possibility is when SmartSynth captures the intent from the NL description incorrectly. This is a common problem for any system that handles NL. Although it cannot completely avoid incorrect mappings, SmartSynth alleviates this problem by empowering the feedback loop, which estimates script likelihood and repeatedly generates alternative interpretations of the provided description, if the current script is not likely.

As with any programming paradigms that use examples or natural language descriptions, the correctness of the synthesized script in SmartSynth needs to be checked by the user. Another limitation of SmartSynth is its tight integration with the API set and the English language. To support any new APIs (or new languages), we have to add new rules and possibly new ranking metrics. Interesting future work is to learn these rules and ranking metrics automatically from training data. Currently, we only support small tasks that can be described in one-liners. We are considering expanding `SmartScript` to support task descriptions that use multiple sentences.

6. RELATED WORK

We discuss related work from both NL understanding and program synthesis.

Natural Language Interfaces (NLIs) There have been numerous attempts to build NLIs for many other application domains such as controlling robots [9], performing navigation [36], processing XML [22], and most notably for querying databases (NLIDBs) [4]. To solve the NL ambiguity problem, NLIDBs normally accept only a restricted subset of natural language [8, 18]. For example, in [8], relative clauses must follow their noun phrases. In contrast, users of SmartSynth can give free-form descriptions.

Another challenge in implementing NLIDBs is the capability to perform conversations with users, where they can give anaphora (referring to previous objects) and elliptical (incomplete) sentences based on the context and previous query results [4]. Because of its nature, SmartSynth does not have this problem. However, it does provide feedback on the scope of queries and performs interactive conversations with the user to resolve query ambiguities.

The main difference between SmartSynth and other NLIs is its use of type-based synthesis algorithms. While other systems have difficulties in extracting *all* necessary constraints, SmartSynth overcomes this problem by exploiting the capabilities of working with under-specifications of synthesis algorithms to complete the missing dataflow relations. We believe that our approach is applicable for building natural language interfaces for other domains as well.

Specification Extraction From NL Extracting specifications automatically from NL has long been a research dream. Kate *et al.*

propose an approach to transform NL to formal languages [20]. Xiao *et al.* develop a template-based method to extract security policies from NL software documentations [41]. Pandita *et al.* analyze API descriptions to infer their method specifications [31]. Our approach complements these existing approaches on specification extraction. SmartSynth can also leverage advances in this area to improve its NLP engine's capability of identifying components and detecting dataflow relations, which may reduce the burden on its synthesis engine to complete the relations.

General-purpose Programming Using NL and Keywords NaturalJava [33] is an attempt to bridge the gap between NL and general-purpose programming languages. It performs translation from a restricted form of NL, which is centralized around Java's programming concepts, to Java statements. It requires the user to think and give descriptions at the syntactical level of Java. Little and Miller propose a code completion tool that synthesizes the most likely Java expression in a code context from a set of keywords [23]. SmartSynth is different, in that it synthesizes a complete script and does not require extra contextual information.

Metafor [25, 26] considers the programming task as telling a story. It automatically generates the program structure from a given NL description. It can extract classes and method names from the description, but the paper is unclear about how to extract the constraints to form sequences of statements inside the methods.

Digital Assistants Siri [5] is a virtual personal assistant that allows users to ask questions and give verbal commands. It originated from the CALO project [37], the "largest known artificial intelligence project in U.S. history" [2]. Siri can handle a wide range of queries, but only those that are simple and relate to a single phone functionality [6]. In other words, users cannot give complicated commands that combine different phone features. Voice Action [10] is another personal assistant application targeting Android phones. However, it requires its users to give queries that fit pre-defined patterns. It also does not allow compositions of supported functionalities.

In contrast, users of SmartSynth can give commands containing events, conditions, and actions without any restrictions. Thus, SmartSynth provides users with greater flexibility and control over their smartphones.

Program Synthesis Program synthesis is the task of automatically synthesizing a program in some underlying domain-specific language from a given specification using some search techniques [11]. It has been applied to various domains [11], including bit-vector algorithms [15], graph algorithms, mutual exclusion algorithms, intellisense for auto completion [17, 27], and spreadsheet macros [12, 14, 35]. In this paper, we apply program synthesis to a new end-user programming domain of smartphone scripts. The traditional intent specification mechanisms based on logical specifications or examples are not useful here: Logical specifications are beyond the expertise of end-users and not worth the effort for small scripts. The by-example paradigm [13] does not apply since smartphone scripts are not functional programs: they are event-based and have side-effects. In contrast, we use natural language.

Our synthesis algorithm performs *type-based synthesis* that refers to a class of search techniques that use typing information/abstraction to perform search and produces a ranked set of results. Type-based synthesis has been used for various applications: synthesizing snippets that can convert a given source type into a given target type [17, 27], completing partial expressions [32]. , assembling a given set of APIs into a program [15]. In each of these cases, the programmer starts out with (incomplete) program structures and the underlying synthesis engine generates ranked completions/assemblies of these structures. SmartSynth is different from these systems in two key ways: (i) SmartSynth does not require the user to start out with program structures — these are automatically inferred from NL descriptions; (ii) SmartSynth extracts relational information from NL descriptions, which helps disambiguate between multiple solutions and significantly improves the effectiveness of ranking.

[2] `http://sri.com/about/siri.html`

7. CONCLUSION

There is a proliferation of programmable mobile systems, thanks to the market for devices with various form factors and the competition. Natural user interfaces for programming are important for two reasons. First, standard programming is beyond the expertise of hundreds of millions of end-users or non-programmers. Second, even programmers find it difficult to keep pace with new syntax and APIs supported by new systems. To this end, we have proposed SmartSynth for synthesizing smartphone automation scripts from natural language descriptions. SmartSynth combines and refines techniques from two different research areas, namely Natural Language Processing and Program Synthesis. It is able to synthesize the intended scripts, in real time, for over 90% of the 640 NL descriptions obtained from a user study (involving 50 different tasks collected from smartphone help forums).

We believe that our NL translation techniques can also be useful for structured logical domains in education, such as elementary geometry [16] and automata theory [3]. This can pave the way for exciting advances in computer-aided education, such as automated grading [3] and solution generation [16] given NL descriptions of word problems.

Acknowledgments

We would like to thank our shepherd, Deepak Ganesan, and the anonymous MobiSys reviewers for valuable feedback on earlier drafts of this paper. This research was supported in part by NSF grants 0917392 and 1117603, and a Microsoft SEIF award. The information presented here does not necessarily reflect the position or the policy of the government and no official endorsement should be inferred.

8. REFERENCES

[1] `http://nlp.naturalparsing.com/`.

[2] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[3] R. Alur, L. D'Antoni, S. Gulwani, D. Kini, and M. Viswanathan. Automated grading of dfa constructions. In *IJCAI*, 2013.

[4] L. Androutsopoulos. Natural language interfaces to databases - an introduction. *Journal of Natural Language Engineering*, 1, 1995.

[5] Apple. Siri for iPhone. `http://www.apple.com/iphone/features/siri.html`.

[6] J. Aron. How innovative is Apple's new voice assistant, Siri? *The New Scientist*, 2011.

[7] Crafty Apps. Tasker for Android. `http://tasker.dinglisch.net/`.

[8] S. S. Epstein. Transportable natural language processing through simplicity–the PRE system. *ACM Transactions on Information Systems (TOIS)*, 3(2), 1985.

[9] C. Finucane, G. Jing, and H. Kress-Gazit. LTLMoP: Experimenting with language, temporal logic and robot control. In *IEEE/RSJ ICIRS*, 2010.

[10] Google. Voice Actions for Android. http://www.google.com/mobile/voice-actions/.

[11] S. Gulwani. Dimensions in program synthesis. In *PPDP*, 2010.

[12] S. Gulwani. Automating string processing in spreadsheets using input-output examples. In *POPL*, 2011.

[13] S. Gulwani. Synthesis from examples: Interaction models and algorithms. *SYNASC*, 2012. Invited talk paper.

[14] S. Gulwani, W. Harris, and R. Singh. Spreadsheet data manipulation using examples. *Communication of the ACM*, 2012.

[15] S. Gulwani, S. Jha, A. Tiwari, and R. Venkatesan. Synthesis of loop-free programs. In *PLDI*, 2011.

[16] S. Gulwani, V. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In *PLDI*, 2011.

[17] T. Gvero, V. Kuncak, and R. Piskac. Interactive Synthesis of Code Snippets. In *CAV*, 2011.

[18] G. Hendrix, E. Sacerdoti, D. Sagalowicz, and J. Slocum. Developing a natural language interface to complex data. *ACM Transactions on Database Systems (TODS)*, 3(2), 1978.

[19] D. Jurafsky and J. H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics and Speech Recognition. Prentice Hall, 2nd edition, 2008.

[20] R. J. Kate, Y. W. Wong, and R. J. Mooney. Learning to transform natural to formal languages. In *AAAI*, 2005.

[21] D. Klein and C. D. Manning. Accurate unlexicalized parsing. In *ACL*, 2003.

[22] Y. Li, H. Yang, and H. V. Jagadish. NaLIX: an interactive natural language interface for querying XML. In *ICMD*, 2005.

[23] G. Little and R. C. Miller. Keyword programming in Java. In *ASE*, 2007.

[24] G. Little and R. C. Miller. Keyword programming in Java. *ASE*, 2009.

[25] H. Liu and H. Lieberman. Metafor: visualizing stories as code. In *IEEE IUI*, 2005.

[26] H. Liu and H. Lieberman. Programmatic semantics for natural language interfaces. In *Extended Abstracts on Human Factors in Computing Systems*, 2005.

[27] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *PLDI*, 2005.

[28] M. D. Marneffe, B. MacCartney, and C. D. Manning. Generating typed dependency parses from phrase structure parses. In *LREC*, 2006.

[29] Microsoft Research. on{X}. http://onx.ms/.

[30] MIT Center for Mobile Learning. MIT App Inventor for Android. http://appinventor.mit.edu/.

[31] R. Pandita, X. Xiao, H. Zhong, T. Xie, S. Oney, and A. Paradkar. Inferring method specifications from natural language api descriptions. In *ICSE*, 2012.

[32] D. Perelman, S. Gulwani, T. Ball, and D. Grossman. Type-directed completion of partial expressions. In *PLDI*, 2012.

[33] D. Price, E. Riloff, J. Zachary, and B. Harvey. NaturalJava: a natural language interface for programming in Java. In *IEEE IUI*, 2000.

[34] M.-R. Ra, B. Liu, T. L. Porta, and R. Govindan. Medusa: A Programming Framework for Crowd-Sensing Applications. In *MobiSys*, 2012.

[35] R. Singh and S. Gulwani. Learning semantic string transformations from examples. *VLDB*, 2012.

[36] I. Song, F. Guedea, F. Karray, Y. Dai, and I. El Khalil. Natural language interface for mobile robot navigation control. In *ISIC*, 2004.

[37] SRI International. Cognitive Assistant that Learns and Organizes. http://www.ai.sri.com/project/CALO.

[38] N. Tillmann, M. Moskal, J. de Halleux, and M. Fahndrich. TouchDevelop: programming cloud-connected mobile devices via touchscreen. In *ONWARD*, 2011.

[39] Two forty four a.m. LLC. Locale. http://www.twofortyfouram.com/.

[40] N. Wirth. Program development by stepwise refinement. *Communication of the ACM*, 14(4), 1971.

[41] X. Xiao, A. Paradkar, S. Thummalapenta, and T. Xie. Automated extraction of security policies from natural-language software documents. In *FSE*, 2012.

[42] B. Yan and G. Chen. AppJoy: Appjoy: Personalized mobile application discovery. In *MobiSys*, 2011.