Spring 2012 LTI Colloquium Report

Programming by Demonstrations and Verbal Commands

Ni Lao

1. Introduction

This is a response to Pedro Moreno's talk "Google's Speech Internationalization Project: From 1 to 300 Languages and Beyond". In this talk, Pedro described how speech recognition is made possible for hundreds of languages on mobile devices. This technology reminds me the great potential of enabling end users to do programming. In this report I will first briefly survey the current state of programming by demonstration (PbD) research, and then I will describe a PbD framework called Verbal Programming Architecture (VPA), which allows end users to do programming through verbal commands. An important feature of VPA is to use Path Ranking Algorithm as (PRA) as the engine to generalize users' action sequences.

2. Programming by Demonstration

Long before the existence of personal computers, people (Teitelman, 1966) have got the idea of users constantly coaching computers in order to achieve functions so complex that are hard to express by programming. However, it was until the advent of personal computers that many such attempts have been made (Lieberman, 2001; Cypher, 1993). Naturally many of these attempts reflect the diverse need of different users---i.e. personalization of computer programs. Generally, **Programming by Demonstration (PbD)** is an end-user development technique for teaching a computer or a robot new behaviors by demonstrating the task to transfer directly instead of programming it through machine commands.

The tasks people try to personalize have a wide range including

- controlling software (e.g. CAD, GIS, games, word processing, graphical editing etc.)
- information extraction (e.g. from emails)
- information gathering (e.g. from web browsers)
- creation of simple applications
- robotic control

Although none of these systems is popular or commercially successful today, I will assume in this report that some form of highly personalized application (e.g. information gathering) will be significant enough in the future, and this report will focus on solving an abstract PbD problem.

The motivations behind these PbD systems can be any one of the following

- personalization
- repeated tasks
    - e.g. "send invoices to all California customers on the 1st of each month, and to all other customers on the 15th"
- tasks which need fast than human response
- tasks which need automatic response
    - e.g. automatically invoke a macro whenever a particular dialog box appears.
- the programming bottleneck
    - most users are unable to take advantage of the computers' capabilities because they do not know how to program.

From a technical perspective there are several challenges for PbD

- Determine when to record an action
    - For most existing systems, users can explicitly invoke the recording of actions. Some systems detect repeated actions sequences by the users (e.g. the Eager system).
- Determine when to invoke an action
    - For most existing systems, programs are invoked by user actions--e.g. voice command, or system events--e.g. time of day, keystrokes, etc. Some systems learn signals which trigger the actions.
- Generalize users' action sequences into program.
    - Many approaches have been proposed, which I will discuss in detail later.
- Specify the flow of control
    - There may be conditions in the control flow--e.g. to handle failures, or loops. Most systems automatically detect such constructs, and ask the user to verify.
- Access internal data of existing applications
    - To avoid this problem, most systems are built from scratch. Several exceptions are systems which provide APIs --e.g. Windows, FireFox.
- Make sure the user is in control
    - Some systems allow the user to execute the program one step at a time. Some systems allow the user to rollback a stack of actions taken by the program.

A central problem of PbD is to generalize users' action sequences into programs which are applicable to new contexts. Whenever the user selects an object in a demonstration, the PBD system must determine why that particular object was selected, so that when the program is invoked in the future -- in a different context -- the appropriate object will be selected. A data description specifies how to select the appropriate object. The description might be "the rectangle created in the previous step", "the first word in the current document", or "a button named SEND". Generally, most systems use search or pattern matching techniques to come up with several hypothesis, visualize then to the user, and let him/her choice the most appropriate one. In some cases, the hypothesis is showing as a high level script, and the user is allowed to directly edit it. Some patterns that have been inferred by different systems are

- **Condition-action rules**: Some systems (e.g. Peridot, Metamouse) predefine a set of condition-action rules, if the condition applies to the current context, then the user is asked if the rule should be applied.
- **Repeated sequences**: Some systems (e.g. Predictive Calculator, Eager) keep a history of all k actions taken by the user, and whenever the user does k-1 actions that match a previously recorded sequence, then the k-th action is predicted.
- **Conjunctions of features**: Sometimes the PbD need the ability to infer conjuncts of features, e.g. select "all New England customers with small orders".
- **Iteration**: which can be classified as set iteration, iteration with a counter, and iteration until a condition is satisfied
- **Conditional branches:** allow a program to execute different code in different situations. They have traditionally been difficult to specify by demonstration, since it can be laborious to demonstrate each complete path through a program. Pygmalion, Tinker and Metamouse address this issue by allowing the user to postpone recording the "else" part of a conditional until that situation arises during the use of the program.
- **Arguments binding:** When a recorded procedure is invoked in a new context, it must establish bindings for the various objects in the program. Most systems let the users explicitly specified the arguments in a new context.

This report mainly considers the argument binding problem, which is very close to the relational learning problem that PRA is trying to solve.

3. Verbal Programming Architecture

This section describes a PbD framework called Verbal Programming Architecture (VPA), which allows end users to do programming through verbal commands. The key characteristics of VPA are

- **Graph representation**: the environment of a problem is represented as edge-typed graphs, which may include information such as attributes of objects and users, global state variables, adjacency of lines or word tokens in a text, or general world knowledge. This representation allows us to develop a general purpose PbB algorithm.
- **PRA as inference engine**: PRA is a relational learning algorithm which is efficient for large scale application, and expressive enough to capture complex patterns.
- **Recursive definition of functions:** one limitation of existing PbD systems is the lack of ability to combine lower level commands to form higher level commands. For example solving a linear equation consists of several steps, but itself can be a sub-routine of a higher level command.

3.1. Problem Definition

Formally, we define the problem of **verbal programming** as the following.

- The **environment** E={O, T, R} consists of a set of **objects** O={o_i}, a set of object **types** T={t_j} and a set of **relations** R={r_k}. The nodes are typed, for example a node can be an UI object in an operating system, a file in storage system, or a text span. A relation r_j:O→2^O maps each object in its range to a set of objects (pointers). For example a file in windows is a node in the environment, and a size_of relation maps this node to a node of double value.
- There is an **agent** which can execute actions to this environment, and also interact with a user in order to figure out what the user want. Each **agent action** has the form of a(t_1,…,t_n), where t_i is the type of the i-th argument. For example a *delete_file* command has one argument of type *file*.
- There is a set of **commands** C={c_i} that can be invoked by the user. By default, each action a(t_1,…,t_n) correspond to an atomic command c(t_1,…,t_n), but the agent can also learn compositional commands which invoke two or more commands.

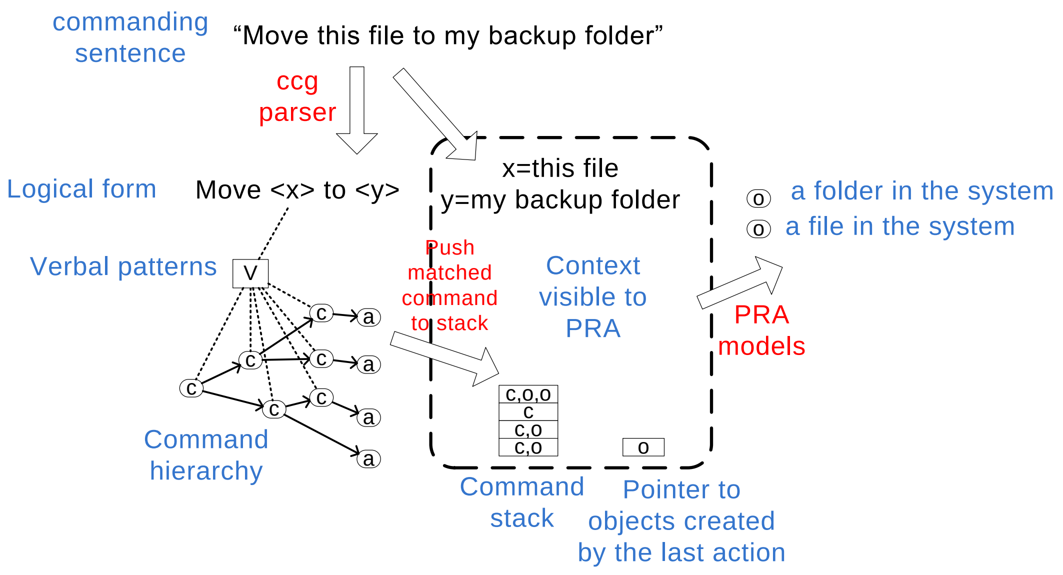
There is a single user interacting with the agent. The user can speak three types of sentences.

- A **commanding sentence** invokes a predefined or learned command of the agent--e.g. "delete this file". Since the meaning of vocabularies can be learned through HCI, we assume an unbounded vocabulary size and also pronouns are acceptable.
- A **programming sentence** is supposed to define/redefine a command--e.g. "the open control panel command has two steps--left click the start bottom, and left lick the control panel link".
- A **feedback sentence** is supposed to give feedback to the current behavior of the agent--e.g. "stop", "correct", "wrong".

3.2. Architecture

Here we assume that the agent can already differentiate commanding, programming, and feedback sentences through some sort of speech recognition and template matching techniques. The figure below demonstrates how a commanding sentence s (e.g. "Move file 001 to folder 001") is processed.

- s is first parsed by a ccg parser into **verbal patterns**-- "Move<x> to <y>", and **NP arguments** --"this file" and "my backup folder".
- The verbal pattern is matched to an existing command c. Note that different patterns might map to the same command.
- The NP arguments of c are translated into objects in the environment by PRA models.
    - Each argument of a command comes with *a set of weighted relation paths* which can retrieve objects in the environment to be used as arguments. These paths can be learned from demonstrations by the user in the past.
- Pushed the fully translated command and arguments to the top of a command stack
- If the top of the stack is a composite command, then it is replaced by a sequence of lower level command, otherwise it is executed.



References

[1] Cypher, Allen (1993), Watch What I Do: Programming by Demonstration, Daniel C. Halbert, MIT Press, ISBN 0-262-03213-9

[2] Lieberman, Henry (2001), Your Wish is My Command: Programming By Example, Ben Shneiderman, Morgan Kaufmann, ISBN 1-55860-688-2

[3] Warren Teitelman (1966), PILOT: A Step Toward Man-Computer Symbiosis, September, PhD Thesis, MIT