DIY Assistant: A Multi-Modal End-User Programmable Virtual Assistant

Michael H. Fischer∗
Giovanni Campagna∗
Euirim Choi
Monica S. Lam

Computer Science Department
Stanford University
Stanford, California, United States

Visit a couplecooks.com. “Start recording price.” “Return this.” “Run price on this.” Result returned to user.

Figure 1. Creating a virtual assistant skill that returns the cost of ingredients in a list using DIYA (DIY Assistant). (a) A user sees a cookie recipe on a popular food blog and wants to see how much the ingredients are. (b) They enter DIYA’s recording mode using their voice and search for one of the ingredients on Walmart’s website. (c) They click on the first search result and highlight the price, telling DIYA via voice that it should be returned. (d) A few days later, they are interested in the “Spaghetti Carbonara” recipe on another food blog. They highlight the ingredients and ask DIYA to run the previously defined program with them. (e) DIYA returns the prices of the items.

Abstract

While Alexa can perform over 100,000 skills, its capability covers only a fraction of what is possible on the web. Individuals need and want to automate a long tail of web-based tasks which often involve visiting different websites and require programming concepts such as function composition, conditional, and iterative evaluation. This paper presents DIYA (Do-It-Yourself Assistant), a new system that empowers users to create personalized web-based virtual assistant skills that require the full generality of composable control constructs, without having to learn a formal programming language.

With DIYA, the user demonstrates their task of interest in the browser and issues a few simple voice commands, such as naming the skills and adding conditions on the action. DIYA turns these multi-modal specifications into voice-invocable skills written in the ThingTalk 2.0 programming language we designed for this purpose. DIYA is a prototype that works in the Chrome browser. Our user studies show that 81% of the proposed routines can be expressed using DIYA. DIYA is easy to learn, and 80% of users surveyed find DIYA useful.

CCS Concepts: • Software and its engineering → Domain specific languages; Programming by example; • Human-centered computing → Natural language interfaces; Personal digital assistants.

ACM Reference Format:

1 Introduction

Many enterprises today are improving the cost efficiency of their businesses with Robotic Process Automation (RPA), the use of AI bots to automate routine digital tasks. This market has been projected to reach over 1.5B USD in 2020. Today, process automation is performed mainly by developers in RPA service companies. This paper explores enabling end users to automate their own workflows, making automation affordable for individuals and small companies. As established by the trend of consumerization of IT, technology
shown to be useful for consumers may also be adopted in business workflows.

This paper proposes **DIYA** (Do-It-Yourself Assistant), a multi-modal system that lets end users apply conventional programming concepts to the task of web automation, without having to learn a formal language.

### 1.1 End-User Programmable Virtual Assistants

We look to the virtual assistant as a new software architecture for end-user process automation. Today, commercial assistants like Alexa offer 100,000 skills, which are APIs that users can invoke simply by voice. Recent projects [18, 19, 22, 31] allow users to create new skills by program-by-demonstration on mobile apps, such as ordering coffees or finding the closest restaurant. Instead of defining primitive skills on single apps, IFTTT [14] supports composition of APIs in “if-this-then-that” constructs, using a graphical user interface. The Almond virtual assistant [6] generalizes the if-this-then-that construct to “when-get-do”, and uses a neural semantic parser to translate natural language into a formal programming language called ThingTalk 1.0. End users can now specify simple event-driven programs in natural language.

As our goal is to automate consumers’ repetitive work flows, we conduct a formative user study to understand the nature of these tasks. Of the 71 tasks suggested by the users in our study, 99% of the skills are intended for the web. Here are some representative examples of tasks that consumers or workers would like to automate:

* Buy these concert tickets as soon as they are available.
* Send Happy Holidays to all my friends on Facebook.
* Translate all non-English emails in my inbox to English.
* Order food for a recurrent employee lunch meeting.
* Compile a weekly report of sales.
* Send a personally-addressed newsletter to all people in a list.
* Check the price of a list of stocks.

We find that users do not want to just replace a few clicks with a verbal command. But rather, the tasks they wish to automate require operating across multiple pages, where the result of one page is used as input in another; the tasks may be repeated periodically or applied conditionally and to multiple elements in a data set. To accomplish such tasks, it is insufficient to let users specify just single-statement or straight-line programs. End users need to bring all standard programming language concepts to bear to create tasks of arbitrary complexity. These concepts include function abstraction, composition of control constructs, and carrying states across statements with variables. This paper asks if it is possible to give the full-power of programming to end-users in webtask automation, without requiring them to learn a formal language.

### 1.2 The Design of DIYA

We propose **DIYA**, short for Do-It-Yourself Assistant, a multi-modal end-user programmable virtual assistant of web-based tasks. A **DIYA** user can define new skills involving GUI interactions on the web; they can invoke skills by voice, on parameters given verbally or by pointing to them with the mouse. **DIYA** is designed to be powerful, yet easy to use and learn, without requiring the user to learn a formal programming language.

Figure 1 shows how a user defines a skill to research the cost of a recipe using **DIYA**. They take a recipe on a website, define a “price” function that returns the price of an ingredient in Walmart, and run “price” on the list of ingredients. This simple routine combines information from two different websites, making it unlikely a dedicated API combining both exists. It involves iteration and aggregation, which current virtual assistants are unable to do.

**Web-Automation Tradeoff.** Virtual assistant skills today are implemented by developers connecting voice interfaces to APIs. Not only are APIs unavailable for most web services, end users do not know how to use APIs. Consumers know their task as visiting certain pages, choosing from the menus, entering words in the appropriate textboxes, and clicking the sequence of buttons. They cannot even verbalize their tasks in detail without referring to the GUI interface. Thus, the simplest way for end users to specify new functionality is to automate their web operations. Automating operations via the GUI interface takes advantage of the generality of the web and minimizes the learning overhead. However, web pages are heterogeneous and dynamic in nature. A web page is updated more often than an API, and skills defined by web page navigation operations are more fragile.

**DIYA** is like a lightweight scripting tool that lets end users automate their repetitive, long-tail tasks. It is useful for “off-the-cuff automation of one-off tasks” on the web. When faced with repeated tasks, such as writing personally addressed emails for a long mailing list, consumers would find **DIYA** handy, provided we keep the automation process quick and easy to learn. This approach complements the more robustness API-based implementations which exist only for the most frequently used skills. It is also possible that once we capture the intent of the end users, GUI operations can be substituted with API calls by professionals in the future.

**Multi-Modal Specification.** To give the user full power of a programming language in web automation, **DIYA** introduces a multi-modal interface that combines the advantages of programming by demonstration (PBD) and natural-language specification. A traditional PBD system passively observes a user’s browsing behavior and then replays a straight-line sequence of steps. More advanced systems use synthesis techniques to generalize the action sequences to other data.

---

1A video showing how **DIYA** is used on a similar example is available at [https://almond-static.stanford.edu/papers/diya-pldi21.mp4](https://almond-static.stanford.edu/papers/diya-pldi21.mp4).
elements in iterative execution. Having the system intuit what the user wants can yield only the simplest of programs. On the linguistic front, neural semantic parsers have been applied to translate individual natural-language sentences into single-statement programs. Not only is it hard for users to verbalize complex programs, it is hard also to train an accurate enough semantic parser.

In our multimodal system, the PBD modality lets users enter multiple steps naturally with specific concrete input parameter values. Voice commands allow users to create abstraction by naming functions, and to generalize from a program trace, such as why a certain element is chosen to enable conditional execution, and which functions are to be applied to a data set.

**ThingTalk 2.0 Design.** To give end users the full power of programming, we need a formal programming language as the target of the multimodal specification. We cannot simply adopt a conventional programming language. Take the design of ThingTalk 1.0 for example. It was designed to support natural-language specification of single-statement programs [7]; as such it does not even have variables, as end users do not know anything about variables. To go from single statements to a full-blown programming language, we introduce ThingTalk 2.0, which supports function abstraction, composability of statements, and the use of variables to carry state across statements. The language is designed carefully to support multimodal specification with the goal of making it as natural as possible for the end users. For example, we do not ask end users to indicate looping with conventional nested “begin” and “end” constructs.

Our insight is to support composability just with function definitions. As end users have learned to invoke skills in virtual assistants, they are familiar with function abstraction. By enabling users to define functions which can themselves invoke and compose functions, we support composition of all control constructs to create programs with arbitrarily complex functionality. Instead of teaching users to declare function signatures formally, the user only needs to learn the pair of “start recording” and “stop recording” commands, which is obvious. The user only works with concrete values; DIYA automatically substitutes it with parameter references. The user does not have to think abstractly. We infer the function signatures in a PBD as the user says the function name and uses the mouse to indicate the input parameter value, etc. Function composition follows naturally as the user selects the result of a function and invokes another. Thus, the user gets the power of nesting control constructs and function composition without having to be taught.

**1.3 Contributions**

Our paper makes the following contributions:

1. Our need-finding survey shows that consumers are interested in automating their tasks on the web, many of which require control structures: iterations, conditionals, and function composition.
2. **DIYA:** the first multi-modal virtual assistant that lets end users define new web-based skills with control constructs, without learning a formal language. DIYA supports 81% of the tasks collected in our need-finding user survey.
3. ThingTalk 2.0: the first virtual-assistant programming language for automating web-based tasks with a multi-modal PBD specification. The language supports full composability of functions, iterative and conditional statements while providing a natural easy-to-learn specification interface.
4. We developed an end-to-end prototype of the **DIYA** design. In a user study involving five tasks in a controlled environment, we show that the system is easy to learn and use. In a user study with four real-world scenarios, 80% of the users find our prototype useful.

## 2 Overview of **DIYA**

The goal of **DIYA** is to support users on their day-to-day work and personal tasks. These tasks include monitoring data and receiving alerts, generating reports including summary statistics, submitting repeated forms over each element of a list (sending emails, placing orders). The tasks can span multiple websites, and can require complex logic and computation.

As a PBD system, **DIYA** allows users to perform their routine as usual, clicking on buttons and typing text into textboxes. **DIYA** requires the user to add only a few commands by voice to turn each task into a virtual assistant skill they can invoke. A **DIYA** specification is thus multi-modal, consisting of **web primitives**, which capture the mouse and keyboard operations, and **constructs**, which are the verbal statements to describe the control flow. The **DIYA** system translates the specification into virtual assistant skills written in ThingTalk, which can then be invoked either by pure voice or in a multimodal fashion combining voice and GUI.

### 2.1 Example

We introduce how **DIYA** works by way of the recipe pricing example in Table 1. The first column shows what Bob, our user, says and does; the second column shows the corresponding ThingTalk statements. The @ sign denotes a call to a function in the library; parameters are passed by keyword. Web primitives are mapped to library calls, the constructs are mapped to ThingTalk control constructs.

Bob’s first task is to build a virtual assistant skills that lets him query the price of any ingredient. Bob copies the name of an ingredient, opens Walmart.com, and starts recording the “price” function. He pastes the name of the ingredient in the search, searches the product, selects the price and returns it. This completes the “price” function (lines 1 to 7).

With the “price” function in hand, Bob can now build a virtual assistant skill that computes the price of a whole recipe. Bob visits a recipe website and starts recording the
“recipe_cost” function. He types “Grandma’s chocolate cookies” into the search box. He then indicates that this is a parameter to be called “recipe”, and clicks Search (lines 10 to 12). He then clicks on the first recipe (line 13), and sees the full list of ingredients. He needs to compute the price of each ingredient, so he selects all the ingredients in the list, then says “run price with this” (line 15). Because he has selected multiple elements, the selection is iterated and the “price” function is called on each element, returning a list of prices. Bob is shown the list of prices computed immediately. He then says “calculate the sum of the result”, and “return the sum”. He finishes recording the top level function by saying “stop recording”.

Subsequently, when he encounters a different cookie, such as “white chocolate macadamia nut cookie”, he can say “run recipe on white chocolate macadamia nut cookie”, or select the name with his mouse and say “run recipe on this”.

This example shows that Bob is performing his task as usual and needs to issue only a few extra verbal commands. The user is responsible for delineating the start and end of each function, naming the function, and identifying the parameters. The rest of the commands, such as running the “price” function and computing the sum, provide meaningful functionality to the end user.

### 2.2 Co-Design of ThingTalk 2.0 and its Multi-Modal Specification

Here we discuss the principles behind the design of ThingTalk 2.0 as well as how the language is specified. Each construct in the language will be formally covered in Sections 3 and 4.
Conditionals do not have an “else” clause. In PBD where users are operating with concrete values, the users can only perform actions that follow from conditions the concrete values satisfy. In the future, we can add “else” clauses by letting sophisticated users refine a defined function with additional demonstrations using alternate concrete values.

**Parametrization and variables.** Many tasks require passing parameters and carrying state across multiple statements, for example to pass the result of one function call to the next. We design ThingTalk so that many programs can be written without explicitly declaring variables and parameters. We bind the current selection in the GUI to the implicit variable “this”, which can be referred to naturally by voice. We bind the “copied” value in the implicit variable “copy”, which can be used in subsequent “paste” operations. If the user pastes a value that is copied before the current function definition, the user is implicitly defining input parameters. This design works well as it mirrors the standard GUI design to have only one clipboard value and one latest selection.

Whereas conventional programmers typically go back to add variable declarations as needed, program-by-demonstration is inherently sequential. We let the user declare input parameters when they are first used, and they are automatically added to the function signature being defined, as in the case of the “recipe” parameter in the example (Table 1, line 11).

**DIYA** also supports user-defined variables, but they are expected to be used only by expert users. Whereas programming languages are typically parsimonious in feature design, **DIYA** is designed to make the common case easy for the sake of learnability. Note that user-defined variables allow explicit tracking of multiple parameters and variables in the same function. This increases the expressiveness of the language beyond what is allowed by pure PBD, which only carries state implicitly in the web page itself or with copy-paste.

### 2.3 **DIYA** System Overview

The **DIYA** accepts multimodal commands to (1) apply virtual assistant skills to information on the website and (2) specify new skills via the programming-by-demonstration paradigm. The system captures the specification in ThingTalk, a formal language with well-defined semantics. This facilitates building other tools, such as allowing the programs to be read back to the user and conversational editing of the programs; these tools, however, are outside the scope of this paper. The architecture of the system is shown in Fig. 2. The system consists of (1) a translator that maps the multimodal specification into ThingTalk, and (2) a ThingTalk runtime system that executes the code. Note that because **DIYA** is a program-by-demonstration system, execution of a **partial function** is necessary while a function is being defined.

The specification translator consists of two modules: a **GUI Abstracter** that converts the mouse and keyboard operations into web primitives in ThingTalk, and a **natural language processing module** that translates the natural language sentences into ThingTalk constructs. The latter consists of an automatic speech recognition (ASR) component which translates the voice input into text, and a semantic parser that translates the text into code.

The ThingTalk runtime system consists of a JIT compiler and a library of browser-automation APIs running on a separate browser. This allows users to issue a virtual assistant command while not running a browser themselves. Moreover, this is necessary even in the specification of a function that calls another function, so as to show the results to the user for demonstration, without affecting the current page. Details on the system are discussed in Section 5.

### 3 Web Primitives

As the user demonstrates the program, the GUI interactions are translated into built-in functions in ThingTalk so they can be replayed, as shown in Table 2. We must record all keyboard inputs, mouse clicks on buttons and links, as well as select, cut/copy, and paste. We do not need to record operations such as scrolling or moving the mouse, as those operations only affect the view of the user. Drawing with the mouse, which involves a click and a drag, is not currently supported.

#### 3.1 Parameters and Variables

To support parameterization of skills, as well as computation on results, **DIYA** must track and distinguish between literals and variables. **DIYA** needs the variable names and literals to generate the code, and the actual values to run the code and generate results, so as to support the user’s demonstration.
<table>
<thead>
<tr>
<th>DIYA Web Primitives</th>
<th>ThingTalk Web Primitives</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open page (url)</td>
<td>@load(value)</td>
<td>Navigate the browser to the given URL.</td>
</tr>
<tr>
<td>Click (element)</td>
<td>@click(selector)</td>
<td>Click on the element matching the CSS selector.</td>
</tr>
<tr>
<td>Cut/Copy (element)</td>
<td>let copy = @query_selector(selector)</td>
<td>Read the text in each element matching the CSS selector and bind it to variable “copy”.</td>
</tr>
<tr>
<td>Select (element)</td>
<td>let var = @query_selector(selector)</td>
<td>Read the text in each element matching the CSS selector and bind it to variable “this” and a local variable (var-name) if given.</td>
</tr>
<tr>
<td>“Start selection”</td>
<td>let var = @query_selector(selector)</td>
<td>While in selection mode, add the clicked elements to the CSS selector and bind it to variable “this” and a local variable (var-name) if given.</td>
</tr>
<tr>
<td>“Stop selection”</td>
<td>@set_input(selector, value)</td>
<td>Set the input elements matching the CSS selector to the value of the “copy” variable or the first parameter if the “copy” variable is not defined in the function.</td>
</tr>
</tbody>
</table>

Table 2. Web primitives that the user can perform in DIYA, and the corresponding ThingTalk statements. “CSS selector” refers to the selector derived from the element used during demonstration.

<table>
<thead>
<tr>
<th>DIYA Constructs</th>
<th>ThingTalk Constructs</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Start recording (func-name)”</td>
<td>function (func-name)();</td>
<td>Begin recording a new function, with the given name.</td>
</tr>
<tr>
<td>“Stop recording”</td>
<td></td>
<td>Complete the current function definition and save it for later invocation.</td>
</tr>
<tr>
<td>“Run (func-name) [with (var-name)] [if (cond)]”</td>
<td>[let result =] [((var-name)[]).,,(cond)] ⇒ (func-name)($(var-name).text))</td>
<td>Execute a previously-defined function, optionally passing the content of the named variable (or “this” to refer to the current selection), and optionally filtering based on a predicate on the content of the variable. The result, if present, is stored in the “result” variable.</td>
</tr>
<tr>
<td>“Run (func-name) [with (var-name)] at (time)”</td>
<td>timer($(time)) ⇒ (func-name)()</td>
<td>Execute the function every day at the given time. Optionally, the function is applied over each element of the named variable.</td>
</tr>
<tr>
<td>“Return (var-name) [if (cond)]”</td>
<td>return (var-name)</td>
<td>Use the named variable as the return value of the current function. The variable can be “this” to refer to the current selection. If a condition is specified, only the elements matching the condition are returned.</td>
</tr>
<tr>
<td>“Calculate the (agg-op) of (var-name)”</td>
<td>let (agg-op) = (agg-op)(number of (var-name))</td>
<td>Compute the given aggregation operator based on the numeric values in the named variable, and save it as a new variable.</td>
</tr>
</tbody>
</table>

Table 3. Constructs that DIYA understands and corresponding ThingTalk statements. The user issues each construct vocally. The table includes only the canonical form of each utterance; users can use different words to convey the same meaning.

of the program. As each statement is generated, DIYA also remembers the values of the variables in the program.

In ThingTalk, we distinguish between input parameters and local variables. Input parameters are always scalar string values. Local variables are used to represent the content of the user selection on the page and contain a list of HTML elements; a scalar variable is a degenerate list with one element. Each entry in the list records a unique ID of the HTML element, the text content, and the number value, if any. Applying a scalar operation on a variable indicates that the operation is applied iteratively to each element. This avoids additional terminology for iterations.

To minimize the need for users to declare and manage variables, the run-time system has two implicit variables: the copy variable to save the copied or cut value and a this variable to remember the selected variable. Users can refer to the variable being selected as “this” in the immediate verbal command. The copy variable is only referred to implicitly in paste operations. The user can also define named variables, by issuing the command “this is a (variable-name)” after selecting a value. Parameters and variables can be referred to by name in ThingTalk constructs discussed in section 4.

Input parameters are also inferred by DIYA. Users can input a value during a demonstration by pasting in a buffer, typing
in a textbox, or selecting from an HTML drop down box. For the former, any time a paste operation refers to the "copy" variable defined outside the function, it is considered an input parameter. For the latter two cases, the user indicates that the value they just entered is an input parameter by saying that "this is a (variable-name)" (Table 1, line 11). Otherwise the value is considered a literal (Table 1, line 10). DIVA will augment the signature of the function being defined upon encountering a new parameter.

To allow the user to select complex lists of elements, and elements in tables or in pages with complex layouts, in addition to the plain browser selection, DIVA also supports an explicit selection mode. The user enters the selection mode with the voice command "start selection". While in selection mode, the page is not interactive: instead, clicks add or remove the clicked elements to the current selection. Selection mode is exited with "stop selection". Once exited, selection mode is treated equivalently to a native browser selection operation.

### 3.2 Selector References

ThingTalk uses CSS (Cascading Style Sheets) selectors [35] to refer to the HTML element of interest. CSS selectors are a language for describing a subset of HTML elements in a page, originally designed for styling. CSS selectors use semantic information to identify the elements (HTML tag name, author-specified ID and class on each element), positional and structural information (ordinal position in document order, parent-child relationship), and content information.

When recording the action, DIVA records which element the user is interacting with, and generates a CSS selector that identifies that element uniquely. When available, DIVA uses ID and class information to construct the selector, falling back to positional selectors when those identifiers are insufficient to uniquely identify the element. As such, the CSS selectors DIVA generate are robust to changes in the content of the page and small changes in layout. DIVA selectors work best with static HTML pages where IDs and classes are assigned according to the semantic meaning of each element. Examples of selectors are shown in Table 1.

### 3.3 Web Primitive Statements

Each GUI interaction is directly translated into a ThingTalk statement, as shown in Table 2. For each statement, the buttons, output texts, or input boxes accessed are represented as CSS selectors. The "open page" operation is immediately added based on the current URL when the user starts recording, and also when the user navigates explicitly by typing in the address bar. The "click" action is faithfully recorded to be replayed. A "copy" operation binds the selected text to the "copy" variable. The "select" operation maps to the "query_selector" web primitive, which binds the selected values to the "this" variable, and to a local variable if a variable name is added. The "paste" operation is mapped to a "set_input" web primitive, which may refer to either the "copy" variable or the first parameter depending on whether a copy operation is issued in the same function. The "type" is also mapped to "set_input", and refers to the literal unless parameterized by the user.

### 4 Control Constructs

DIVA supports function composition, iteration, and conditional execution. While the control constructs, as shown in Table 3 are limited, DIVA has rich functionality because it can invoke and combine any of the public or custom virtual assistant skills.

Functions in DIVA can be invoked by voice as skills outside of the browser. Functions should only depend on the input parameters and not the state of the browser, such as its history or the content of any filled form before recording is started. That is, functions are not just macros that are to be replayed in the current calling context. The definition of a function should start immediately after loading a webpage. The function can depend on the persistent state (cookies, server-side state) and can perform side effects.

The user delimits a function definition with a pair of "start recording (func-name)" and "stop recording". At most one return statement can appear in the function, but the return statement need not be the last. It can be followed by additional web primitives, which do not affect the return value. This allows the function to perform "clean up" actions, such as logging out, before returning the result. The returned value can be the "this" variable, or a named variable.

Functions can be run with the "run" construct. If the function has one parameter, the user can simply say "run (func-name)" with either "this" or a named parameter. If the function has multiple parameters, the parameter passing convention is based on key value pairs. The user must name the actual parameters with the names of the formal parameters in the function, and the user can simply say "run (func-name)". All functions are defined with scalar parameters. When functions are applied on list variables, the functions are applied individually to each element. If a function has a result, the result is bound to the "result" variable. Outside of a demonstration, functions can be set to run at a certain time, such as "at 9am".

Function invocations, return statements, and iterative statements can be conditionally executed. The computation is performed on each element of the current selection or named variable that satisfies the given predicate. For example, the invocation "run alert with this if this is greater than 98.6", where "this" is a list of temperatures, generates an alert for each temperature above 98.6. The ThingTalk syntax is

```plaintext
this, number > 98.6 ⇒ alert(param = this.text);
number is a field of the currently selected HTML elements (in the "this" variable) and it is computed by extracting any numeric value in the elements.
```
Our current system currently only supports a single predicate, which can be equality, inequality, or comparison between the current selection and a constant. As the natural language technology improves, we expect to support arbitrary logical operators (and, or, not) in the future.

Finally, diyA lets users perform aggregate operations on the current selection or a named variable. The supported operations are those used in database engines: sum, count, average, max, and min. The user can issue the voice command “calculate the ⟨operation⟩ on ⟨var⟩”. The result is stored in a named variable with the same name as the operation.

5 The diyA System

In this section, we describe how diyA transforms the multimodal specification into a ThingTalk function, and how that ThingTalk is later executed. The high-level flow of the diyA system is shown in Fig. 2.

5.1 Translating the Specification to ThingTalk

diyA contains a GUI abstractor component that records all the actions the user performs on a page, and maps them to the corresponding ThingTalk statements. The GUI abstractor uses a browser extension to inject JavaScript code that intercepts the actions on each page. The browser extension clearly displays to the user a prominent indicator when it is recording the user’s actions.

When the user starts recording a function, diyA must first record the context in which the function is recorded. diyA records the current URL, and maps it to a @load web primitive. diyA’s injected JavaScript code listens to all interaction events (keyboard, mouse, clipboard) from the browser on all parts of the page. When an event is intercepted, the injected JavaScript code considers the HTML element that is the target of the event, and constructs a new CSS selector that identifies that element uniquely. The CSS selector is used to construct the corresponding ThingTalk web primitive.

diyA continuously listens for the user’s voice, and reacts to the commands that map to ThingTalk control constructs (Table 3). Each utterance is passed to automatic speech recognition (ASR), followed by a semantic parser that translates it into a ThingTalk fragment. The ThingTalk code is then passed to the ThingTalk runtime, and diyA acknowledges the user’s command by speech.

5.2 ThingTalk Runtime

The ThingTalk runtime must support the user in (1) defining the function by demonstration, (2) invoking a pre-defined function while browsing, (3) executing the function itself. In the following, we describe these three run-time contexts in reverse order: execution, browsing, and demonstration.

5.2.1 The Execution Context. Once a ThingTalk specification is complete, it is compiled to native JavaScript code using a ThingTalk compiler, which is built by extending the ThingTalk compiler [6] to support user-defined functions.

The Javascript code is executed on an automated browser: a form of browser that is driven with an automated API rather than interactively by the user. This means that the function can be invoked as a skill by voice, for example, on a smart speaker. Every function invocation occurs in a new session in the browser, starting with a @load web operation. That is, each function executes in a separate, fresh copy of a webpage. This ensures that the callee does not affect the calling function, except via returned results. Nested function invocations are managed with a stack; a new invocation pushes a new execution context on the stack, which is popped when the function terminates.

The environment of the execution consists of all the explicitly and implicitly declared variables and parameters. The semantics of each web primitive and construct is informally introduced in Tables 2 and 3, respectively. For each reference of a CSS selector, the runtime extracts the HTML elements as specified from the page. The @load, @click, and @set_input functions are mapped to the corresponding web automation APIs to manipulate the webpage. @query_selector evaluates the value of the specified selector. Note that the browser selection and clipboard are not affected in an execution context. Iteration, conditional execution, and aggregation operations are implemented in straightforward JavaScript code.

5.2.2 The Browsing Context. A user can invoke a predefined function while browsing, whenever the diyA browser extension is enabled. The browsing context keeps track of the latest values bound to the implicit variables, “this” and “copy”, and any other explicitly named variables. There is a single browsing context shared by all pages in a running browser, and all variables named in the browsing context are global. The values are derived from the HTML elements in the webpages visited. As an optimization, we bind the values of the implicit variables lazily when they are used, because their values are available as the “clipboard” and “selection” in the browser. When running a pre-defined diyA function, the values in the browsing context are passed into the execution context. As discussed, the execution of any diyA function does not alter the state of the browsing context.

5.2.3 The Demonstration Context. When a user starts recording a function, the user enters the demonstration context. Here the user is specifying the code to be generated and executing it at the same time as demonstration.

The demonstration requires the computation to be performed collaboratively between the user and the diyA runtime. The user performs web operations directly with their mouse and keyboard in the normal browser (not an automated one). The runtime is responsible for executing all function calls and aggregation operations in a separate automated browser, so that the results can be returned for
the user to continue the demonstration. This requires the run-time system to

1. generate the code as the user defines it, as shown in Tables 2 and 3,
2. track the browsing context as the user demonstrates, as discussed in Section 5.2.2.
3. run the functions and aggregation operations as they are added to the function definition. Each function and computation is run in a new browser session as discussed in Section 5.2.1, and the result is returned to the user in the user’s normal browser.

6 Implementation

We implemented an end-to-end prototype for DIYA, written in JavaScript. The implementation consists of a Google Chrome browser extension that injects the DIYA recording code in every page the user visits and a standalone Node.js application containing the ThingTalk execution code. The standalone application is based on Almond [6]; it spawns the automated browser and communicates with it.

To handle the user’s speech, we use the Web Speech API, a native speech-to-text and text-to-speech API available in Google Chrome. We use the annyang library [2] to understand the user’s commands. This library uses a template-based NLU algorithm, requiring the user to speak exactly the supported words. At the same time, it supports open-domain understanding of arbitrary words, which is necessary to let the user choose their own function names. We include multiple variations of the same phrase to increase robustness.

CSS Selectors are generated using the finder library [25]. Event recording code is based off the Puppeteer Recorder Chrome Extension [27], and event replaying uses the Puppeteer API [12] to automatically control Google Chrome. The automated browser controlled by Puppeteer shares the profile with the normal browser, including cookies, local storage, certificates, saved passwords, etc. Automated actions are executed at a reduced speed (slower than typical automation but faster than human execution) to improve robustness to dynamic page conditions and reduce anti-spam measures.

7 Experimentation

To evaluate our system, we performed four experiments: (1) We conduct a need-finding survey to learn what types of web flows users would like to automated. (2) We evaluate whether users can learn the DIYA specification constructs. (3) We evaluate the design choice of supporting implicit variables. (4) We collect user feedback on DIYA in user-suggested scenarios on real-world websites.

7.1 What Do Users Need To Automate?

Our first study is a need-finding online survey to learn what users are interested in automating and whether the primitives in DIYA are adequate. We recruit 37 participants on
Figure 5. Number of skills organized by domains that users were interested in creating using DIYA.

Figure 6. Results of our user studies. “Exp. A” refers to the construct learning study, while “Exp. B” refers to the real world evaluation study.

<table>
<thead>
<tr>
<th>Construct</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic</td>
<td>Automated the clicking of a button.</td>
</tr>
<tr>
<td>Iteration</td>
<td>Send an email to a list of email addresses.</td>
</tr>
<tr>
<td>Conditional</td>
<td>Reserve a restaurant conditioned on rating.</td>
</tr>
<tr>
<td>Timer</td>
<td>Buy a stock at a certain time.</td>
</tr>
<tr>
<td>Filter</td>
<td>Show restaurants above a certain rating.</td>
</tr>
</tbody>
</table>

Table 5. Tasks performed by the participants in the programming construct study.

Table 4. Representative tasks that users wanted to automate. All but the last one can be automated with DIYA. The last task requires computer vision to process the image and is out of scope.

The tasks were unsupervised and performed on custom demo websites in order of increasing complexity, to emulate the learning experience on the system. Before each task, users watched a video demonstrating how the control construct worked. They then repeated the task. Lastly, they were asked to do a different task that requires the same construct. The five tasks they performed on their own are shown in Table 5. Note that because the “Iteration” task requires two parameters, the recipient name and their email address, the users have to name the parameters explicitly, instead of relying on the copied data as the implicit parameter.

Quantitative Results. Participants successfully completed the new tasks assigned using DIYA 94% of the time. After the tasks, users were asked to complete a survey, rating a number of questions on a 5-point Likert scale from “strongly disagree” to “strongly agree”. The results are shown in Figure 6 as “Exp. A”. We notice that users consistently found the system easy to learn (72%), and easy to use (75%). 91% are satisfied with the experience of testing it. The multimodal interface (“MMI” in the plot) is rated helpful by 81% of survey participants. Overall, 66% of the users agree that
DIYA is useful. These results confirm the need and usefulness of DIYA, and suggest that the programming constructs employed by DIYA can be learned. Note that in these experiments, users are only exposed to simple tasks on a custom website, rather than real world scenarios, which explains the relatively low propensity to find DIYA useful.

**Privacy.** When automating a task that involves personal identifiable information, 83% of the users wanted a privacy-preserving system that ran locally. 66% of users wanted privacy protection even for tasks that did not involve personal identifiable information. As our system is able to run on the client as a Chrome extension and native application, we are able to offer users privacy.

### 7.3 Evaluating Implicit Variables

Instead of requiring users to define all their variables, DIYA introduces the implicit "this" variable that the users can define and use with select and paste actions. To evaluate this design decision, we conduct a user study with 14 users (7 men and 7 women, average age 25). The study was conducted over video conferencing. We ask the users to build an example skill using both the explicit and implicit naming methods. Overall, 88% subjectively preferred the implicit version, because it had fewer steps and was faster. We found users didn’t like talking to their computer as much.

### 7.4 Real Scenarios Evaluation

In our final experiment, we want to get users’ feedback of using the system in real life tasks drawn from our first user study, using websites they are familiar with: Walmart, a recipe website, a stock website, and a weather website. This test is more complicated as it uses a combination of constructs; it also illustrates the utility of the system because the tasks are more realistic. This end-to-end test also demonstrates that DIYA is a fully functional system on real websites.

Each task involves the user defining a skill and invoking it to see the result. We evaluated the following real-world scenarios, chosen based on the need-finding experiment:

1. **Calculate the average of temperatures.** The user creates a program that goes to weather.gov, enters their zip code, calculates the average high temperature for the week, and returns that value. This example exercises the multi-selection and aggregation function.

2. **Add items to an online shopping cart.** The user has a shopping list of items that they entered, and they need to add them all to a shopping cart on everlane.com. This scenario requires user input, copy and paste, and iteration.

3. **Notify when stock prices dip.** The user creates a skill on zacks.com to receive a notification when a stock quote goes under a fixed price. The skill is then triggered each day at a certain time. This tests the conditional and timer functions of the system.

4. **Add ingredients from a website to a shopping cart.** This task is similar to the task in Fig. 1. The user visits a cooking website, acouplecooks.com, to find the price of all the ingredients in a recipe on walmart.com. They need to define a price function, and apply it iteratively to ingredients in the recipe. This tests users’ understanding of calling functions in an iterative construct.

We conducted this study as an interactive user test (live over video-conferencing) using the same participants as the design decision study described above. Users first complete a warm-up task of recording a simple function to familiarize with DIYA. Then they are asked to complete each task manually and on DIYA following a predefined script. Whether each user completes the task manually or using DIYA first is randomized.

All users were able to install DIYA on their Chrome browser and complete the tasks successfully. We perform two surveys. First, to find out if defining a function in DIYA greatly adds to the cognitive load in comparison of just executing the task, we ask the users to complete the NASA-TLX survey [13] after each task. NASA-TLX is a standard set of metrics to assess the perceived workload of a task. The user is asked to rate their mental load, temporal stress (perceived time pressure), overall task performance (subjectively assessed by the study participant), required effort, and frustration. Lower scores are better in all categories of the survey, except for performance, where higher values are better. The graph shows the median of each categories as middle line in each box; the box shows the second and third quartiles, and the vertical lines show the range of the distribution, excluding outliers. The aggregated results of the survey shown in Fig. 7 indicate that there is no statistically significant difference across all five metrics between completing the tasks by hand and programming a skill using DIYA. We also asked the users to self-report the amount of time it took them to complete the task by hand, and to record the DIYA skill. We found no statistical difference, although we found some significant noise in the data due to self-reporting. Overall, both NASA-TLX and the timing comparison suggest that programming a skill is no harder than performing the task by hand. The key benefit is that the tasks can now run automatically in the future, which can result in a huge time savings, especially for trigger-based tasks like checking stock prices. This is a promising result that shows automation can be practical for end-users.

Second, users also complete a Likert-scale evaluation on the whole system. This survey shows how users would perceive its value with real-world scenarios. The results are shown in Fig. 6 as "Exp. B": 73% of the users find the system easy to learn; 46% find it easy to use, probably due to the complexity of the tested tasks. Nonetheless, 67% are satisfied with the system. On usefulness, 73% find the multimodal interface useful, and 80% of the users find DIYA useful. When
Figure 7. Evaluating the perceived workload of completing 4 real-world tasks by hand and with diya, using NASA-TLX scores. Lower scores are better in all categories, except for performance, where higher values are better.

compared with the results in Exp. A, since the tasks are more realistic but harder, the perceived usability and satisfaction are a little bit lower, but the perceived usefulness is higher.

7.4.1 Qualitative Feedback. During the user test, we also collected qualitative feedback from the participants. A user that was not able to program before said, “when you’re raised on sci-fi movies, the thought of a system that can learn what you need by voice is incredibly appealing.” A user saw the system as being very helpful in repeating common tasks to accomplish her job, “for me as a data person especially, during the COVID-19 crisis when local governments are behind on data standards, I’ve found the lack of such tool exhausting. The level of manual data entry required to achieve my basic analysis goals is often more than I can make time for, and one day that I fail to check is data that may be permanently lost. I love the idea of being able to program that cleanly, with my voice. I love that it can intelligently extract numbers from characters and perform basic operations, and run just by speaking.”

8 Discussion

This paper demonstrates that it is feasible for end users to automate webtasks using composable control constructs. Here we describe what our experimental results taught us about the tradeoffs we have made and make suggestions for future work.

8.1 Web Automation

Web automation is inherently fragile; it is not possible to automate all web tasks; automated routines break as web pages are updated. Nonetheless, web pages remain the only means of access to most of the services on the Internet. Where possible, web automation is useful.

Anti-Automation Measures. DIYA does not work on websites that actively block web automation. Websites such as Facebook or Google actively prevent bots from accessing their pages as an attempt to guard against fraudulent use. They can detect the use of automated browsing APIs, and can detect input that is driven by a program as opposed to a user with keyboard and mouse. Various techniques have been proposed to subvert these detection mechanisms [4] but as the subversion improves, so does the anti-fraud detection.

Elements Selection. In the diya implementation, web elements are selected using CSS selectors. We choose CSS selectors because they are an expressive, existing DSL for selecting elements based on semantic and structural information, and because they are a well-known standard already supported by browsers and web automation libraries. We also note that CSS selectors are already commonly used in hand-written web automation scripts.

CSS selectors are robust to changes in the content of the page, but not in changes in a website’s layout. They are also
incompatible with dynamic CSS modules, and automatically generated CSS classes adopted by certain popular CSS libraries like React. We detect some of those libraries and ignore those CSS classes, but this is necessarily incomplete.

Empirically, we observed that web pages with numerical information, such as weather or stocks, tend to have stable layouts and work well with CSS selectors. We also found that form fields are typically annotated with CSS IDs and classes, which is sufficient to identify them robustly. Conversely, we found that websites with a lot of free-form content, such as blogs, are challenging because similar pages can have vastly different hierarchies and low-level layouts. We also found that elements in the lists shown by search engines and shopping websites can be identified well, but sometimes advertisements change the layout of the page unexpectedly.

Our experience with CSS selectors suggest that a higher-level semantic representation for web elements could be beneficial. Our exploration shows that it is possible to identify a web element given its text label, color, size, and relative position to other objects on a page [33]. Adopting a similar representation may improve the robustness of DIYA.

Timing Sensitivity. If allowed to run at full speed, a web automation may fail because it can refer to web elements that are yet to be loaded. Thus, DIYA runs at a reduced speed to allow time for the page to react to the actions performed by the user, including any animation or external HTTP request that the page needs. We found a 100 millisecond slow-down for every Puppeteer API call to be generally sufficient to replay the scripts robustly. This can be sped up by automatically discovering the events in the page that signal the page is ready for the next action [3].

8.2 Voice Input

The current DIYA implementation uses the audio speech recognizer by Google Chrome, which we have found empirically to be quite brittle. We mitigated this limitation by showing the user the transcription generated by the API. If the transcription is incorrect, most likely we do not recognize any command and the user can issue the command again. DIYA uses a strict grammar-based NLU system, which has high precision (recognized commands are interpreted correctly) but low recall (not all commands are recognized). This can be made more robust by integrating with the Genie library for neural semantic parsing of ThingTalk commands [7].

8.3 Privacy

DIYA is designed to run locally on the user’s machine to protect the user’s privacy. We chose a fully local implementation because DIYA must have access to the full browser profile of the user, including their cookies, stored passwords, etc. This information is too sensitive to be uploaded to a third-party server. Privacy protection is confirmed by our user study to be an important feature to keep in the future.

8.4 Skill Management and Editability

This paper focuses on how end users can define new skills. DIYA needs to be extended in the future to help users maintain their skills by providing an interface to view and edit skills. The users may need to record additional traces to handle alternative conditional execution paths, which the system would merge. Skills may need to be updated when the web pages they operate on change. Iterative refinement will also be needed to create more complex skills. Since the skills are succinctly and formally represented in ThingTalk, designed to be translated from and into natural language, the interface can be provided at either the natural-language or ThingTalk level to cater to users of different levels of technical expertise.

9 Related Work

DIYA is the first system that enhances a voice assistant with a PBD system that supports function composition. It translates multi-modal specifications into a fully compositional programming language with functions, conditional, trigger-based, and iterative constructs.

9.1 Virtual Assistants

Commercial virtual assistants such as Alexa and Google Assistant let users perform actions on the web using a voice interface. These systems rely on existing APIs that third-party developers must integrate. Furthermore, commercial virtual assistants only support one action at a time. They support only limited custom skills in the form of “routines”, which are sequences of voice commands that can be defined once and invoked subsequently by voice or with a timer.

Almond [6, 7] was the first virtual assistant to provide limited end-user programmability. Almond supports commands with three-clauses: “when” a condition happens, “get” some data, and “do” some action. Natural language sentences are translated into ThingTalk 1.0 programs, which consist of a single statement that connects together up to two skills. ThingTalk 1.0 does not support complex task that need sequential execution or variables. It also does not support user-defined functions: skills are mapped to some existing APIs and must be created by programmers, limiting the scope considerably.

The Brassau assistant [11] automatically generates reusable graphical widgets for each natural language command. Brassau widgets are limited in functionality to one command at a time, and in appearance by the automatic generation. DIYA skills on the other hand operate on the existing graphical web page, which can be shown to the user alongside the pure voice interface, to provide rich feedback.

Hey Scout, a browser-based voice assistant [52], let users perform simple web browsing tasks via a voice interface. It led to Mozilla’s public release for Firefox of a similar system [5]. Unlike our system, however, neither of these assistants interact with the content on web pages.
9.2 Multi-Modal PBD

Previous work has introduced multi-modal interfaces to extend the expressive power of PBD systems [18, 19]. None of these works support building complex tasks compositionally. They do not support nested function calls and do not support iteration. SUGILITE [18] uses PBD to create new skills for voice assistant; it automatically recognizes parameters from the input sentence and matches them to the demonstration. DIYA instead lets the user specify parameters explicitly, which is more precise. In APPINITE [19], users describe in voice the reason they are selecting each element as they perform the selection. DIYA subsumes this work by making predicates a primitive usable across iterations and function invocations.

PLOW [1] and PUMICE [20, 21] use a multi-modal dialogue agent to learn new high-level concepts in a new natural-language command. The user demonstrates each concept on web page (PLOW) or mobile app (PUMICE). In contrast, DIYA lets users build primitives from the ground up, allowing them to be combined using voice.

9.3 PBD for Automation

In this section, we discuss mobile and web automation via PBD, without multi-modal interaction. Without multi-modal interaction, the user has no ability to specify control constructs during the demonstration. CoScripter [16, 24] uses PBD to generate straight-line programs as natural language traces. The users can later edit the traces to add parametrization. CoScripter also supports the creation of interactive scripts that pause and ask the user for the parameters (rather than providing the parameters upfront). The system lacks support for control constructs and function composition. ActionShot [17] suggests recording users’ web navigation passively and make it available as input to CoScripter.

Early works [10, 15, 26, 29, 30] support iteration by automatically discovering loops given a demonstration of one or a few iterations, using program synthesis. Program synthesis is less reliable than letting the user specify the iteration in voice. If the synthesizer makes a mistake, the user must provide more demonstrations to correct it, which can be frustrating. Synthesis has not been applied to nested loops, which is more challenging due to the larger search space. DIYA, on the other hand, supports arbitrary nesting of loops, using function composition.

Helena [8] proposes a DSL that supports iteration and conditional constructs for scraping web content. The user demonstrates a straight-line execution of how one data item is to be scraped. The system uses program synthesis to generate an iterative construct in the DSL. Later, the user can edit the script with a Scratch-like interface to add conditionals and to correct the program synthesis. Editing the script requires the user to understand the formal Helena language, whereas a user need not know ThingTalk to use DIYA. Helena was intended for and thus evaluated with computer scientists [9], whereas we target non-technical users.

Whereas DIYA is intended to help the end user automate their personal task, KITE [22] is designed to help developers create a dialogue agent from a mobile GUI interface. The user supplies multiple straight-line browsing traces, which are then analyzed to create intent and slots for the agents.

Whereas DIYA uses CSS selectors to identify the web elements, VASTA [31] uses computer vision to recognize interactive elements. Sikuli [34] uses screenshots to refer to the GUI elements for automation. The use of natural language to describe web elements has also been proposed; it can be mapped to graphical elements directly using a neural network [23, 28], or translated first into a semantic representation that refers to the text of the element, its graphical attributes, and its relative position to other elements on the page [33]. Techniques used to identify the elements on the page are complementary to DIYA.

Ringer [3] addresses the problem that PBD systems might perform the actions before the page is ready, for example before a button appears, or fail to identify a button because the layout changed. It uses several related PBD traces to infer when a page is ready for the next action, and it uses heuristic features to identify elements on the page.

10 Conclusion

Virtual assistants are changing the way we interact with computers. Along with this, we need to empower individuals to build programs for virtual assistants, leveraging the vast information on the web, instead of having to rely only on applications built by developers.

This paper proposes DIYA, a system that lets users automate their complex tasks on the web using a multi-modal program-by-demonstration paradigm. DIYA is the first PBD system that supports composing control constructs and functions in one skill. It does so by letting users use voice to define and call functions, and specify control constructs during a demonstration. The multi-modal user specification is translated into a program in ThingTalk 2.0, a programming language we designed for this purpose. We find DIYA to be expressive enough to implement 81% of user-proposed skills, many of which require composition of control constructs. The users in our study find DIYA easy to learn and useful.

In summary, DIYA is an easy-to-learn system that lets consumers create useful virtual assistant web-based skills that require the full generality of composable control constructs.

Acknowledgments

This work is supported in part by the National Science Foundation under Grant No. 1900638 and the Alfred P. Sloan Foundation under Grant No. G-2020-13938.
References


