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

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** → **Programming by example**; *Domain specific languages*; • **Human-centered computing** → *Natural language interfaces*; Personal digital assistants.

Keywords: end-user programming, programming by demonstration, web automation, voice user interfaces, virtual assistants

ACM Reference Format:
Michael H. Fischer, Giovanni Campagna, Euirim Choi, and Monica S. Lam. 2021. DIY Assistant: A Multi-Modal End-User Programmable Virtual Assistant. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual Event, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454046

*Equal contribution

PLDI '21, June 20–25, 2021, Virtual Event, Canada

ACM ISBN 978-1-4503-8391-2/21/06.
https://doi.org/10.1145/3453483.3454046

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. 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 using voice. Recent projects let users create new skills by program-by-demonstration on mobile apps, such as ordering coffee or finding the closest restaurant [18, 19, 22, 31] . Instead of defining primitive skills on single apps, IFTTT supports composition of APIs in "if-this-then-that" constructs using a graphical user interface [14]. The Almond virtual assistant generalizes the if-this-then-that construct to "when-get-do," and uses a semantic parser to translate natural language into a formal language called ThingTalk 1.0 [6]. 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 recurring 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 to bear all standard programming language concepts 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 web 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 for web-based tasks. A DIYA user can define new skills involving GUI interactions on the web, and they can invoke the skills by voice. Parameters are given verbally or by pointing to them with the mouse. DIYA is designed to be powerful, yet easy to use and learn.

Figure 1 shows how a user defines a skill to research the cost of a recipe using DIYA[1]. 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 often do not know how to use APIs. Consumers know their task as visiting certain pages, choosing from available options, entering words in the appropriate input boxes, 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 robust API-based implementations, which exist only for the most frequently used skills. Once we capture the intent of the end users, GUI operations can be substituted with API calls, if they are available, by professionals in the future.

[1]A video showing how DIYA is used on a similar example is available at https://oval.cs.stanford.edu/papers/diya-pldi21.mp4.

Multi-Modal Specification. To give users the 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 actions in the web browser, and then replays a straight-line sequence of steps. More advanced systems use program synthesis techniques to generalize the action sequences to other data elements in iterative execution. Having the system infer what the user wants is applicable only to simple tasks. On the linguistic front, neural semantic parsers have been applied to translate individual natural-language sentences into single-statement programs [7]. Not only is it hard for users to verbalize complex, multi-statement programs, it is also difficult to train a sufficiently-accurate semantic parser.

In our multimodal system, programming by demonstration lets users perform multiple steps naturally, working with specific and 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 scoping with conventional nested "begin" and "end" constructs.

Our insight in the design is to support composability just with function definitions. As end users are accustomed to invoking 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 are obvious. The user only works with concrete values; DIYA automatically substitutes them 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 with summary statistics, and submitting repeated forms over each element of a list such as sending emails or 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 input boxes. 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. DIYA translates the specification into a skill written in ThingTalk, which can then be invoked either by pure voice or by 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.

DIYA **Specification**		**ThingTalk Code**	
Construct:	"Start recording price"	**function** price(*param* : String) {	(1)
Web primitive:	Open walmart.com	@load(*url* = "https://walmart.com");	(2)
Web primitive:	Paste in search box	@set_input(*selector* = "input#search", *value* = *param*);	(3)
Web primitive:	Click Search button	@click(*selector* = "button[type=submit]");	(4)
Web primitive:	Select price of top result	**let** this = @query_selector(*selector* = ".result:nth-child(1) .price");	(5)
Construct:	"Return this value"	**return** this;	(6)
Construct:	"Stop recording"	}	(7)
Construct:	"Start recording recipe cost"	**function** recipe_cost(*p_recipe* : String) {	(8)
Web primitive:	Open allrecipes.com	@load(*url* = "https://allrecipes.com");	(9)
Web primitive:	Type in search box	@set_input(*selector* = "input#search", *value* = "grandma's chocolate cookies");	(10)
Construct:	"This is a recipe"	@set_input(*selector* = "input#search", *value* = *p_recipe*);	(11)
Web primitive:	Click Search button	@click(*selector* = "button[type=submit]");	(12)
Web primitive:	Click the first result	@click(*selector* = ".recipe:nth-child(1)");	(13)
Web primitive:	Select all ingredients	**let** this = @query_selector(*selector* = ".ingredient");	(14)
Construct:	"Run price with this"	**let** result = this ⇒ price(this.text);	(15)
Construct:	"Calculate the sum of the result"	**let** *sum* = **sum**(*number* **of** result);	(16)
Construct:	"Return the sum"	**return** *sum*;	(17)
Construct:	"Stop recording"	}	(18)

Table 1. Sum The Price of Recipe Ingredients. The user performs the actions in the left column, and the corresponding ThingTalk program in the right column is generated. CSS selectors are simplified in the example for illustration purposes.

Bob's first task is to build a virtual assistant skill 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 and 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 with white chocolate macadamic nut cookie," or select the name with his mouse and say "run recipe with 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 a program in ThingTalk is expected to be specified. Each construct in the language will be formally covered in Sections 3 and 4.

Multi-modal end-user-programming. In DIYA, the user specifies the task step-by-step, using the most appropriate modality for each instruction: web primitive operations are provided by demonstration, while control constructs and voice assistant skills are accessed as voice commands. Every primitive or construct is converted into code directly, requiring no sophisticated inference, generalization, or guesswork in the implementation. The user is seeing the results of each action, including function invocations while inside a function definition. The results are shown in a pop-up, so the users can continue the demonstration by reacting to the results.

Integration with virtual assistants. All the skills in the virtual assistant are available to the user. The user can invoke user-defined skills (e.g. "price"), built-in functions (e.g. summation), and standard virtual assistant skills (e.g. weather, search). Conversely, all skills that the user defines are available in the virtual assistant to be used later in a voice-only user interface. This allows the user to seamlessly combine web-based tasks with existing APIs.

Functions for composability and control flow. Unlike conventional programming languages, ThingTalk conditional and iterative constructs can only be applied to a single operation or function. The conventional "begin" and "end" or "{"

and "}" notations do not transfer well to voice input. Instead, iteration is implied when a function is applied to a set of data, as shown in the example. Conditional operation is executed if a variable satisfies a predicate. For example, "return this if it is greater than 98.6." To support arbitrary composition of constructs we rely on function encapsulation, with the added advantage that the user can now refer to the sequence of operations with a meaningful name.

Conditionals do not have an "else" clause. In PBD the user is operating with concrete values, so they 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 created without explicit declarations of 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, PBD 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 conventional 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 tracking multiple parameters and variables in the same function. This increases the expressiveness of the language beyond what is possible with pure PBD, which only carries state implicitly in the web page 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 reading back the program to the user and editing conversationally; 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.

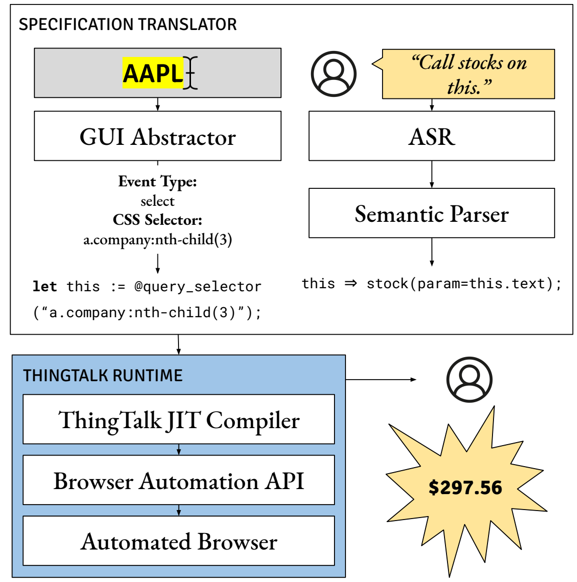

Figure 2. A high-level overview of the algorithm that DIYA uses to convert a multi-modal specification to ThingTalk.

The specification translator consists of two modules: a *GUI Abstractor* 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, 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 users. Drawing with the mouse, which involves a click and a drag, is not currently supported.

DIYA Web Primitives	ThingTalk Web Primitives	Description
Open page (*url*)	@load(*value*)	Navigate the browser to the given URL.
Click (*element*)	@click(*selector*)	Click on the element matching the CSS selector.
Cut/Copy (*element*)	**let** copy = @query_selector(*selector*)	Read the text in each element matching the CSS selector and bind it to variable "copy".
Select (*element*) ["this is a ⟨var-name⟩"]	**let** *var* = @query_selector(*selector*)	Read the text in each element matching the CSS selector and bind it to variable "this" and a local variable ⟨var-name⟩ if given.
"Start selection" [Select (*element*)]* "Stop selection" ["this is a ⟨var-name⟩"]	**let** *var* = @query_selector(*selector*)	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.
Paste (*element*)	@set_input(*selector*, *value*)	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.
Type (*element, value*) ["this is a ⟨var-name⟩"]	@set_input(*selector*, *value*)	Set the input elements matching the CSS selector to the given value or the parameter ⟨var-name⟩ if given.

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.

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 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. If the user applies a function to a list of values, the function is called with each element individually. 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 or typing in an input box, or selecting from an HTML drop-down box. For the first case, any time a paste operation refers to a "copy" variable assigned 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). DIYA 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 pages with complex layouts, in addition to the plain browser selection, DIYA 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, DIYA records which element the user is interacting with, and generates a CSS selector that identifies that element uniquely. When available, DIYA uses the 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 DIYA generates are robust to changes in the content of the page and small changes in layout. DIYA selectors work best with HTML pages where IDs and classes

DIYA Constructs	ThingTalk Constructs	Description
"Start recording ⟨func-name⟩"	**function** ⟨func-name⟩(){	Begin recording a new function, with the given name.
"Stop recording"	}	Complete the current function definition and save it for later invocation.
"Run ⟨func-name⟩ [with ⟨var-name⟩] [if ⟨cond⟩]"	[**let** result =] [⟨var-name⟩[, ⟨cond⟩] ⇒] ⟨func-name⟩([⟨var-name⟩.text])	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.
"Run ⟨func-name⟩ [with ⟨var-name⟩] at ⟨time⟩"	timer(⟨time⟩) ⇒ ⟨func-name⟩()	Execute the function every day at the given time. Optionally, the function is applied over each element of the named variable.
"Return ⟨var-name⟩ [if ⟨cond⟩]"	**return** ⟨var-name⟩	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.
"Calculate the ⟨agg-op⟩ of ⟨var-name⟩"	**let** ⟨agg-op⟩ = ⟨agg-op⟩(number **of** ⟨var-name⟩)	Compute the given aggregation operator based on the numeric values in the named variable, and save it as a new variable.

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.

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" operation is also mapped to "set_input," and refers to the literal unless parameterized by the user.

4 Control Constructs

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

Functions in DIYA 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 form filled 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 are 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 to 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 9 AM."

Function invocations, return statements, and iterative statements can be conditionally executed. The computation is performed on each element of the current selection or a 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

$$\text{this}, \mathit{number} > 98.6 \Rightarrow \mathit{alert}(\mathit{param} = \text{this.text});$$

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 a multimodal specification into a ThingTalk function, and how a ThingTalk function is 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 corresponding ThingTalk statements. The GUI abstractor uses a browser extension to inject JavaScript code that intercepts the actions on each page. The browser extension 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, and clipboard) from the browser on the entire 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 the ThingTalk compiler. 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 browser session 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 run-time 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 does the task they want recorded in the browser, during which the code is generated.

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 the Almond assistant [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 mitigate anti-spam measures.

7 Experimentation

To evaluate our system, we performed four experiments. (1) We conduct a need-finding survey to learn what kind of web tasks users would like to automate. (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.

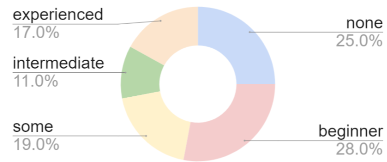

Figure 3. Programming experience of survey participants that proposed skills for DIYA.

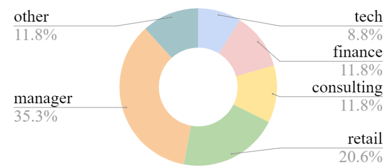

Figure 4. Occupation of survey participants that proposed skills.

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 Amazon Mechanical Turk (25 men and 12 women, average age = 34), each of whom was paid $12 for approximately 60 minutes of their time. Survey participants had a mix of programming experience (Fig. 3) and a variety of backgrounds (Fig. 4). In the survey, respondents were first shown the functionality of the system, then asked to describe 3 skills each that they would like to automate. We collected 71 valid skills.

The proposed skills span 30 different domains, with the most popular being food, stocks, local utilities, and bills (Fig. 5). Representative tasks are shown in Table 4. Of these 71 skills, we found 24% do not require any programming constructs, 28% need iteration, 24% need conditional statements, and 24% need a trigger (a timer plus a condition). In summary, 76% of the skills people want to automate require the control constructs we introduce to PBD.

99% of the skills are intended for the web and 1% are to be run on the local computer. 34% of skills are on websites that need authentication, showing that users are interested in skills that operate on their personal data. We found 81% of the web skills can be expressed using DIYA. For the remaining 19%, 11% require producing charts, and 8% require understanding videos and images. These functionalities are orthogonal to our system and can be added to the system as pre-defined skills. This shows that despite the simplicity of DIYA, it largely covers what people want to automate.

Privacy. When automating a task that involves personally 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 personally identifiable information. As our system is able to run on the

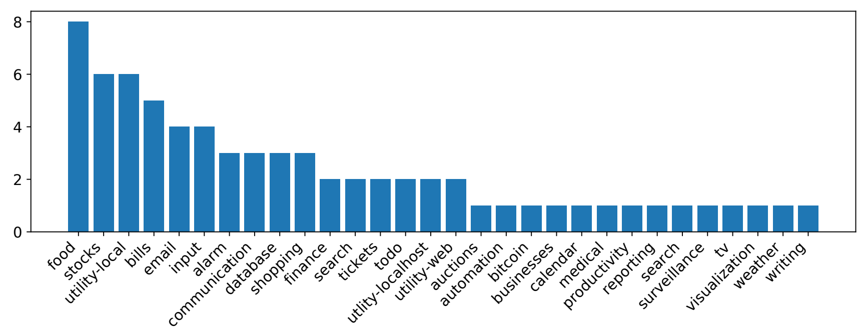

Figure 5. Number of skills organized by domains that users were interested in creating using DIYA.

Domain	Example Skill	Constructs
Communication	"Send a birthday text message to people automatically."	Iteration
Purchasing	"Make a reservation for the highest rated restaurants in my area."	Aggregation (max), Filtering
Purchasing	"Order a ticket online if it goes under a certain price."	Timer, Filtering
Purchasing	"Order ingredients online for a recipe I want to make, but only the ingredients I need."	Iteration, Filtering
Finance	"Check my investment accounts every morning and get a condensed report of which stocks went up and which went down."	Iteration, Filtering
Database	"Automate queries I do by hand every day for work for inventory levels and delivery times."	Iteration
Security	"Alert me when someone moves on the camera of my home security system."	Unsupported

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 the scope of this paper.

client as a Chrome extension and as a native application, we are able to offer users privacy.

7.2 Can Users Learn To Program In DIYA?

Our next study asks if users can effectively learn the programming constructs in DIYA. We conducted a remote user study with the same participants as the need-finding survey. Each participant was asked to perform five tasks, with each task designed to be a realistic example of each of the system's control constructs. 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

Construct	Task
Basic	Automate the clicking of a button.
Iteration	Send an email to a list of email addresses.
Conditional	Reserve a restaurant conditioned on rating.
Timer	Buy a stock at a certain time.
Filter	Show restaurants above a certain rating.

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

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 Fig. 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 multi-modal 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 in DIYA can be learned. Note that in these experiments, users only did simple tasks on a demo website, rather than real world scenarios, which explains the relatively low propensity to find DIYA useful.

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

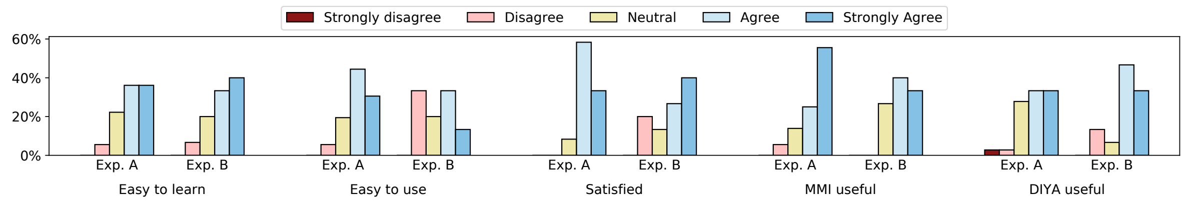

Figure 6. Results of our user studies. "Exp. A" refers to the construct learning study, while "Exp. B" refers to the real world evaluation study.

video conferencing. We ask the users to build an example skill using both the explicit and implicit naming methods. Overall, 88% preferred the implicit version because it had fewer steps and was faster. We found users did not like talking to their computer as much.

7.4 Real Scenarios Evaluation

In our final experiment, we want to get users' feedback on 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 temperature.* 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 scenario 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 enter, and they need to add them all to a shopping cart on everlane.com. This scenario requires user input, copy-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 every day at a certain time. This scenario 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 themselves with DIYA. They are then asked to complete each task manually and on DIYA following a predefined script. Whether each user completes the task first manually or using DIYA is randomized. All users were able to install DIYA on their Chrome browser and complete the tasks successfully.

To find how users would perceive DIYA's value in real-world scenarios, users complete a Likert-scale evaluation on the whole system. 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 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.

We also evaluate if it is harder to define a skill in DIYA in comparison to just executing the task once by hand. We ask the users to complete a 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 category as a 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.

As shown in Fig. 7, the aggregated results of the survey 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. Note that for tasks 2 and 4, which use iteration, users only performed a small number of iterations by hand. 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 run automatically in the future, which

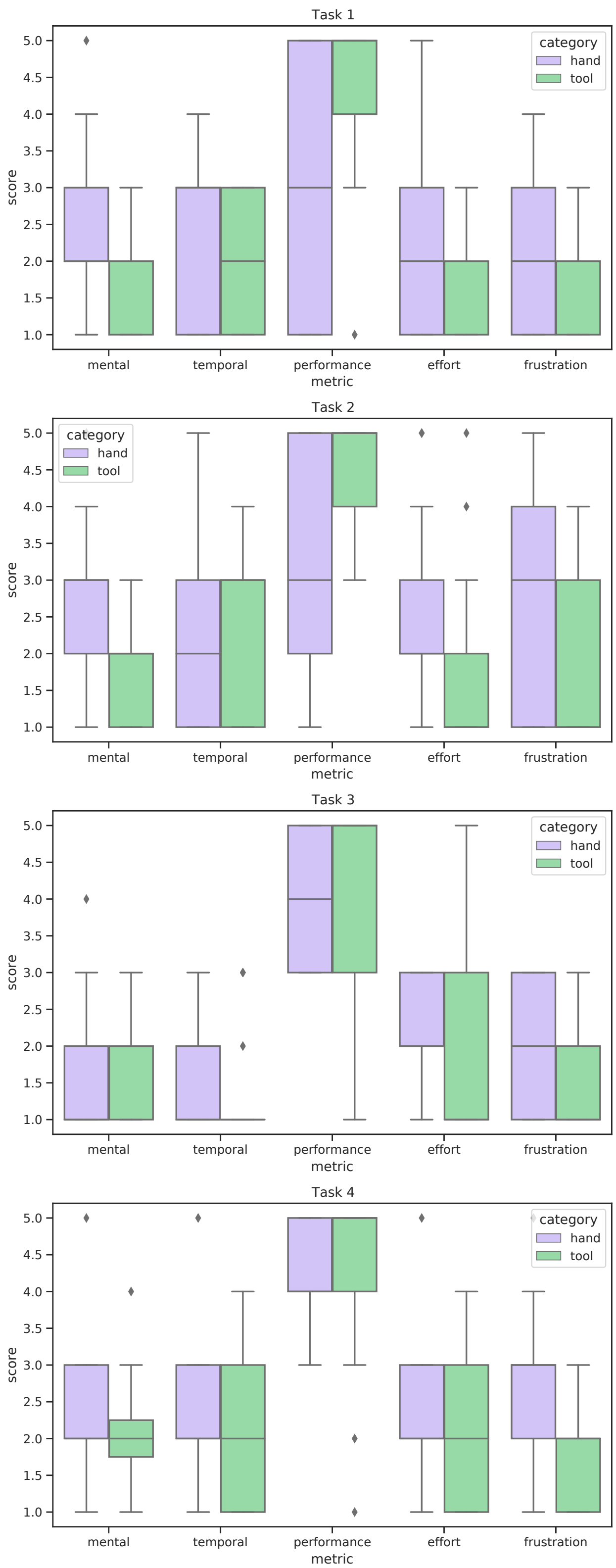

Figure 7. Evaluating the perceived workload of completing four real-world tasks by hand and with DIYA, using NASA-TLX scores. Lower scores are better in all categories, except for performance, where higher values are better.

can save a lot of time, especially for iterative or trigger-based tasks like checking stock prices. This is a promising result that shows automation can be practical for end users.

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." Another 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 a 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 web tasks using composable control constructs. Here we describe what our experimental results taught us about the tradeoffs we have made, and offer some 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 in 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 human using a keyboard and a mouse. Various techniques have been proposed to subvert these detection mechanisms [4] but as the subversion improves, so does the anti-fraud detection.

Element 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 are 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, web automation may fail because it can refer to web elements that have 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 automatic speech recognizer by Google Chrome, which we have found quite brittle empirically. 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. To help users maintain their skills, DIYA needs to be extended in the future 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 virtual 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 user-defined functions, multiple statements, or variables. All skills are implemented with APIs, limiting the scope considerably.

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

Hey Scout, a browser-based virtual assistant, let users perform simple web browsing tasks via a voice interface [32]. 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 of 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 uses PBD to create new skills for virtual assistants; it automatically recognizes parameters from the input sentence and matches them to the demonstration [18]. DIYA instead lets the user specify parameters explicitly, which is more precise. In APPINITE, users describe in voice the reason for selecting each element as they perform the selection [19]. 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 the new concepts on web pages in PLOW and on mobile apps in 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 cannot specify control constructs during the demonstration. CoScripter uses PBD to generate straight-line programs as natural-language traces [16, 24]. 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 making it available 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. 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 are more challenging due to the larger search space, whereas DIYA supports nesting of loops using function composition.

Helena proposes a DSL that supports iteration and conditional constructs for scraping web content [8]. 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 uses computer vision to recognize interactive elements [31]. Sikuli uses screenshots to refer to the GUI elements for automation [34]. Neural networks have been proposed to map high-level verbal descriptions of web elements (the text of the element, its graphical attributes, and its relative position to other elements on the page) to specific graphical elements [23, 28]. Recently, we show it is more accurate to use a neural network to first translate the natural-language description to a formal semantic representation, which is then used algorithmically to identify the element of interest in the target web page [33].

Ringer [3] addresses the problem that PBD systems might attempt actions before the page is ready, for example before a button appears, or fail to identify a button because the layout has 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 skills 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 to 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. The users in our study find DIYA easy to learn and useful.

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

Acknowledgments

We would like to thank the reviewer and the shepherd Dr. Ben Zorn for their feedback. 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

[1] James Allen, Nathanael Chambers, George Ferguson, Lucian Galescu, Hyuckchul Jung, Mary Swift, and William Taysom. 2007. Plow: A collaborative task learning agent. In *AAAI*, Vol. 7. 1514–1519.

[2] Tal Ater. 2019. annyang! Speech recognition for your site. https://github.com/TalAter/annyang.

[3] Shaon Barman, Sarah Chasins, Rastislav Bodik, and Sumit Gulwani. 2016. Ringer: Web Automation by Demonstration. *SIGPLAN Not.* 51, 10 (Oct. 2016), 748–764. https://doi.org/10.1145/3022671.2984020

[4] berstend. 2020. puppeteer-extra-plugin-stealth. https://github.com/berstend/puppeteer-extra/tree/master/packages/puppeteer-extra-plugin-stealth.

[5] Julia Cambre, Alex C Williams, Afsaneh Razi, Ian Bicking, Abraham Wallin, Janice Tsai, Chinmay Kulkarni, and Jofish Kaye. 2021. Firefox Voice: An Open and Extensible Voice Assistant Built Upon the Web.

[6] Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. 2017. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International Conference on World Wide Web - WWW '17*. ACM Press, New York, New York, USA, 341–350. https://doi.org/10.1145/3038912.3052562

[7] Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S. Lam. 2019. Genie: A Generator of Natural Language Semantic Parsers for Virtual Assistant Commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. ACM, New York, NY, USA, 394–410. https://doi.org/10.1145/3314221.3314594

[8] Sarah Chasins and Rastislav Bodik. 2017. Skip Blocks: Reusing Execution History to Accelerate Web Scripts. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 51 (Oct. 2017), 28 pages. https://doi.org/10.1145/3133875

[9] Sarah E. Chasins, Maria Mueller, and Rastislav Bodik. 2018. Rousillon: Scraping Distributed Hierarchical Web Data. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology* (Berlin, Germany) *(UIST '18)*. Association for Computing Machinery, New York, NY, USA, 963–975. https://doi.org/10.1145/3242587.3242661

[10] Allen Cypher. 1995. EAGER: PROGRAMMING REPETITIVE TASKS BY EXAMPLE. In *Readings in Human–Computer Interaction*, RONALD M. BAECKER, JONATHAN GRUDIN, WILLIAM A.S. BUXTON, and SAUL GREENBERG (Eds.). Morgan Kaufmann, 804–810. https://doi.org/10.1016/B978-0-08-051574-8.50083-2

[11] Michael Fischer, Giovanni Campagna, Silei Xu, and Monica S. Lam. 2018. Brassau: Automatic Generation of Graphical User Interfaces for Virtual Assistants. In *Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services* (Barcelona, Spain) *(MobileHCI '18)*. Association for Computing Machinery, New York, NY, USA, Article 33, 12 pages. https://doi.org/10.1145/3229434.3229481

[12] Jack Franklin et al. 2020. Puppeteer Headless Chrome Node.js API. https://github.com/puppeteer/puppeteer.

[13] Sandra G Hart. 2006. NASA-task load index (NASA-TLX); 20 years later. In *Proceedings of the human factors and ergonomics society annual meeting*, Vol. 50. Sage Publications Sage CA: Los Angeles, CA, 904–908.

[14] If This Then That 2011. If This Then That. http://ifttt.com.

[15] Tessa Lau, Steven A. Wolfman, Pedro Domingos, and Daniel S. Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Mach. Learn.* 53, 1–2 (Oct. 2003), 111–156. https://doi.org/10.1023/A:1025671410623

[16] Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. 2008. CoScripter: Automating & Sharing How-to Knowledge in the Enterprise. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Florence, Italy) *(CHI '08)*. Association for Computing Machinery, New York, NY, USA, 1719–1728. https://doi.org/10.1145/1357054.1357323

[17] Ian Li, Jeffrey Nichols, Tessa Lau, Clemens Drews, and Allen Cypher. 2010. Here's What i Did: Sharing and Reusing Web Activity with ActionShot. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Atlanta, Georgia, USA) *(CHI '10)*. Association for Computing Machinery, New York, NY, USA, 723–732. https://doi.org/10.1145/1753326.1753432

[18] Toby Jia-Jun Li, Amos Azaria, and Brad A. Myers. 2017. SUGILITE: Creating Multimodal Smartphone Automation by Demonstration. In *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems* (Denver, Colorado, USA) *(CHI '17)*. Association for Computing Machinery, New York, NY, USA, 6038–6049. https://doi.org/10.1145/3025453.3025483

[19] Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M Mitchell, and Brad A Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 105–114.

[20] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M. Mitchell, and Brad A. Myers. 2019. PUMICE: A Multi-Modal Agent That Learns Concepts and Conditionals from Natural Language and Demonstrations. In *Proceedings of the 32nd Annual ACM Symposium on User Interface Software and Technology* (New Orleans, LA, USA) *(UIST '19)*. Association for Computing Machinery, New York, NY, USA, 577–589. https://doi.org/10.1145/3332165.3347899

[21] Toby Jia-Jun Li, Marissa Radensky, Justin Jia, Kirielle Singarajah, Tom M Mitchell, and Brad A Myers. 2020. Interactive Task and Concept Learning from Natural Language Instructions and GUI Demonstrations. In *The AAAI-20 Workshop on Intelligent Process Automation (IPA-20)*.

[22] Toby Jia-Jun Li and Oriana Riva. 2018. Kite: Building Conversational Bots from Mobile Apps. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services* (Munich, Germany) *(MobiSys '18)*. Association for Computing Machinery, New York, NY, USA, 96–109. https://doi.org/10.1145/3210240.3210339

[23] Yang Li, Jiacong He, Xin Zhou, Yuan Zhang, and Jason Baldridge. 2020. Mapping Natural Language Instructions to Mobile UI Action Sequences. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. Association for Computational Linguistics, Online, 8198–8210. https://doi.org/10.18653/v1/2020.acl-main.729

[24] Greg Little, Tessa A. Lau, Allen Cypher, James Lin, Eben M. Haber, and Eser Kandogan. 2007. Koala: Capture, Share, Automate, Personalize Business Processes on the Web. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (San Jose, California, USA) *(CHI '07)*. Association for Computing Machinery, New York, NY, USA, 943–946. https://doi.org/10.1145/1240624.1240767

[25] Anton Medvedev. 2020. finder: CSS Selector Generator. https://github.com/antonmedv/finder.

[26] Brad A. Myers, Richard G. McDaniel, and David S. Kosbie. 1993. Marquise: Creating Complete User Interfaces by Demonstration. In *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems* (Amsterdam, The Netherlands) *(CHI '93)*. Association for Computing Machinery, New York, NY, USA, 293–300. https://doi.org/10.1145/169059.169225

[27] Tim Nolet. 2020. Puppeteer Recorder. https://github.com/checkly/puppeteer-recorder.

[28] Panupong Pasupat, Tian-Shun Jiang, Evan Liu, Kelvin Guu, and Percy Liang. 2018. Mapping natural language commands to web elements. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, Brussels, Belgium, 4970–4976. https://doi.org/10.18653/v1/D18-1540

[29] Gordon W Paynter. 1999. Familiar: Automating Repetition in Common Applications.. In *New Zealand Computer Science Research Students' Conference*. Citeseer, 62–69.

[30] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann. 2017. Learning Syntactic Program

Transformations from Examples. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 404–415. https://doi.org/10.1109/ICSE.2017.44

[31] Alborz Rezazadeh Sereshkeh, Gary Leung, Krish Perumal, Caleb Phillips, Minfan Zhang, Afsaneh Fazly, and Iqbal Mohomed. 2020. VASTA: A Vision and Language-Assisted Smartphone Task Automation System. In *Proceedings of the 25th International Conference on Intelligent User Interfaces* (Cagliari, Italy) *(IUI '20)*. Association for Computing Machinery, New York, NY, USA, 22–32. https://doi.org/10.1145/3377325.3377515

[32] Janice Tsai and Jofish Kaye. 2018. Hey Scout: Designing a Browser-Based Voice Assistant. (2018). https://aaai.org/ocs/index.php/SSS/SSS18/paper/view/17543

[33] Nancy Xu, Sam Masling, Michael Du, Giovanni Campagna, Larry Heck, James Landay, and Monica S. Lam. 2021. Grounding Open-Domain Instructions to Automate Web Support Tasks. In *Proceedings of the 2021 Annual Conference of the North American Chapter of the Association for Computational Linguistics (NAACL-HLT 2021) (To Appear)*. https://arxiv.org/abs/2103.16057

[34] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology* (Victoria, BC, Canada) *(UIST '09)*. Association for Computing Machinery, New York, NY, USA, 183–192. https://doi.org/10.1145/1622176.1622213

[35] Tantek Çelik, Elika J. Etemad, Daniel Glazman, Ian Hickson, Peter Linss, and John Williams. 2018. Selectors Level 3 (W3C Recommendation). https://www.w3.org/TR/selectors-3/.