Brassau: Automatically Generating Graphical User Interfaces for Virtual Assistants

Michael Fischer  Giovanni Campagna  Silei Xu  Monica S. Lam
Computer Science Department
Stanford University
{mfischer, gcampagn, silei, lam}@stanford.edu

ABSTRACT
Today’s virtual assistants have a text or speech-based conversational interface. While flexible, conversational interfaces have limitations: it is difficult to monitor simultaneous queries, common commands are difficult to recall, outputs are not interactive, and incomplete commands have to be handled with a series of clarifying questions. In this paper, we present Brassau, a virtual assistant that automatically converts natural language commands into a personalized dashboard of interactive widgets.

Brassau introduces a novel template-based method that leverages the large corpus of existing images and apps to make the widgets visually diverse and interesting. Users can turn an image into a template in a few minutes, and Brassau matches the virtual assistant command to the best template and stylizes it to create the GUI. This approach enables Brassau to handle the growth of virtual assistant commands by simply adding more templates. Most of the users in our evaluation find Brassau widgets match the intent of the commands and are preferred over plain widgets.

ACM Classification Keywords
H.5.m. Information Interfaces and Presentation (e.g. HCI): Miscellaneous

Author Keywords
Graphical user interfaces; Aesthetics; Visual preferences; Colorfulness; Complexity.

INTRODUCTION
Conversational virtual assistants allow users to accomplish 15,000 pre-defined [16] tasks, using natural language. Furthermore, virtual assistants, such as IFTTT [14] and Almond [5], let users express compound tasks that connect multiple services. For example, copy pictures from Facebook to Dropbox, or translate stories on Washington Post into Chinese. Virtual assistants have the potential to be highly personalized for different IoTs, web accounts, preferences, and habits. Natural language lets users directly specify the tasks they want.

The conversational interface of a virtual assistant, however, is not without its weaknesses. A voice-only virtual assistant cannot display graphical data such as images, videos or maps. It provides information serially, which is problematic for long lists of results. Tasks that require interactivity, such as adjusting the volume of music to the right level, are slow to do with voice. It is difficult to correct speech recognition errors. In contrast, a graphical interface can show the list of results, let a user click to see more information, and display settings which the user can adjust.

This paper proposes a graphical virtual assistant called Brassau¹ that automatically generates interactive, personalized interfaces from users’ commands so they can see and interact with the results more quickly. Because these interfaces are typically highly specialized and simple, we refer to them as widgets. These widgets have all the frequently used parameters filled in and provide a convenient way for users to modify the input and issue similar commands.

Brassau uses the open-source Almond virtual assistant framework [5] to translate the commands into programs in a formal language called ThingTalk. ThingTalk is a powerful language that lets users connect primitives from the open Thingpedia interface repository in a single command. Moreover, users can add predicates to qualify and filter out the results. Brassau also uses the Almond runtime system as its execution engine.

Brassau uses the semantic information in the natural language sentence and the type information of the input and output parameters in Thingpedia to generate the interactive widget automatically. The widgets must not only be functional, but also easily identifiable and aesthetically appealing. Our approach is to collect a database of GUI templates, find the closest template to a task and stylize it to create the GUI. Since a good interface is known to be subjective [27], this technique also makes it easy to substitute and experiment with different templates. In fact, we can generate multiple GUIs and give the end users a choice.

¹Brassau is a chimpanzee whose paintings were mistaken as painted by a human artist.
Templates provide a graphically interesting background and a well-designed layout of input and output parameters. We created a template markup language, called BANANA, to annotate the templates. Templates can be made without writing a line of code. Our matching algorithm uses a heuristic function to find the closest template based on the semantics and the parameter types. Our stylizing algorithm adjusts colors and fonts to create a consistent design.

**Contributions**
- The Brassau Graphical Virtual Assistant is a novel virtual assistant that supports glanceable notifications, repeating and reusing tasks, and controlling background tasks with the help of a graphical interface.
- Brassau’s Algorithm generates interactive widgets automatically from a virtual assistant command, written in natural language, and translated by Almond to ThingTalk. The algorithm generates diverse, interesting, and semantically relevant GUIs by matching the commands to a template in the database and filling the template. Thus, the algorithm can handle the growth of virtual assistant commands by simply growing the template database.
- A fully functional Brassau has been implemented, and a user study of Brassau shows that the widgets are visually appealing and appropriate to the natural language command that generates them; furthermore, the templates are easy to make.

**RELATED WORK**

**Visual Design and User Experience**
Prior work has addressed the importance of visually appealing design to enhance user experience. Lindgaard et al. [19] show that the visual appeal dominates the first-impression judgments of perceived usability and trustworthiness. The study by Bateman et al. [3] compares the interpretability and recall of plain and visually embellished charts. They find that visual embellished charts are recalled notably more without impacts to interpretability. Although good design has significant impact, it is time consuming to build. Many attempts have been made on the programmatic generation of appealing design for typesetting [17], tag clouds [13, 2], data visualization [29], and natural language queries [33, 11, 32].

**Automatic Generation of GUI**
Research systems have been developed to automatically or semi-automatically lay out user interfaces. In most of these works [34, 37, 6, 30, 8, 26], optimization-based techniques are used to arrange the graphical components such as buttons and menus using different metric, given the specification of the program. Some other works [36, 20] focus on selecting the most appropriate graphical components based on predefined rules or runtime data. SUPPLE [9, 10] takes a step further, by not only generating the layout but also selecting the appropriate interactors and dividing the interface into navigational components. SUPPLE is also able to generate user interfaces for dynamic compositions of devices. Similarly, the universal controller projects [21, 22, 23] by Nichols et al. provides automatically generated UIs for dynamically assembled collections of audio-visual appliances.

All these works focus on optimizing the choice and layout of components, but do not consider the visual design and embellishment of the interface. Webzeitgeist [18] introduces the concept of design mining and provides a tool which help designers parametrically search for visual design examples. ICrafter [25] presents a template-based interactive system which can be dynamically matched to different services. However, creating a template requires coding in a traditional programming language, which not only makes it harder than our system, but can also causes security issues since it runs arbitrary code. Additionally, ICrafter templates only apply to a specific program signature, unlike Brassau’s templates which can be reused for many similar but different programs.

**Multimodal Virtual Assistant**
The majority of virtual assistants nowadays appears to the user as a conversational agent with a chat-based interface. Research has been done on multimodal interfaces to enable a richer and more natural user experience [28, 15, 31]. Although these works give virtual assistants a multimodal interface, the graphical part is still hard coded, which will not scale with the rapidly increasing ability of virtual assistants.

Amazon’s Alexa [1] has added a graphical user interface by introducing the Echo Show this year. Echo Show uses a built-in touch display to show information and do light interactions. Only six fixed plain templates are provided for developers to choose from and developers have to manually specify what content to show and where the content should be placed in the templates. Existing skills cannot be supported by the Echo Show without manual developer work.

**USER EXPERIENCE**
In a text-based virtual assistant, such as Google Assistant [12], the user initiates the conversation by giving the virtual assistant a command. The virtual assistant then responds immediately with a result or a series of interrogative questions to gather needed parameters. If the user wants subsequently to execute the same command, they have to repeat the whole process, or scroll back to find the same command.

With Brassau, as above, the interaction begins with the user typing in their command at the top of the screen. The commands are turned into interactive widgets and added to the users’ dashboard (Figure 2a). Previously used commands preserve their last result, and can be reissued.

The dashboard consists of a set of icons, one for each of the widgets. Users can manage their widgets by reordering them with a press and hold interaction on the icons, or deleting them by tapping the ‘x’ on the upper right corner of the icon. Tapping on the widget expands it, allowing the user to interact with the content. When the widget is expanded, a textual description is shown on the top.

**A User Scenario**
Ann does not like getting notifications as each email message arrives. However, she wants to get any news from her dad about her mom, who is sick. She also wants to be responsive to her boss.
She tells Brassau, “monitor mails from my dad” and “monitor mails from my boss”. Brassau automatically creates a widget for each event. Even though both events monitor GMail, Brassau creates distinctive icons containing her dad’s and boss’s profile pictures, as shown in Figure 2a. With a glance, Ann sees messages from both her dad and boss while she is in a meeting. She quickly swipes through the messages from her dad, shown in Figure 2b, and defers responding to her boss’s messages until after the meeting, shown in Figure 2c. Personalized notifications are an important virtual assistant use case, and widgets let users prioritize them. This example also shows how Brassau lets users scan through long messages quickly.

Ann’s mom is from China, and prefers to read in Chinese, but Ann finds writing in Chinese difficult. Ann tells Brassau “Translate ‘I’ll be home for Christmas’ to Chinese”. Brassau creates a widget, shown in Figure 2d, in which Ann can vary the input easily to get another translation. Here Brassau turns a task into a reusable widget for other inputs to help users re-issue similar commands.

Ann has a baby girl at home; she sets up a webcam in the room so she can see her while at work. She tells Brassau “show my camera in the nursery”. Ann would leave the widget maximized on her dashboard so she can see her baby throughout the day, shown in Figure 2a. This shows how Brassau keeps people up-to-date with glanceable commands, so they do not have to ask their virtual assistant constantly for updates.

Whenever Ann does not have an early meeting, she asks Brassau to order a medium latte to deliver to her house. Brassau automatically creates a widget so Ann only needs to tap a button. This shows how Brassau helps users perform repeated tasks more quickly. As shown in Figure 2e, Ann can also change the kind of coffee, its size, or the delivery location, if she so pleases.

Ann used to forget to turn off the lights before leaving home. Now, Brassau can do it automatically after she asks: “turn off the lights every day at 9 am”. She can also disable the rule easily if she happens to be working from home, and re-enable it afterwards (Figure 2f). This shows how Brassau supports background tasks.

When she gets home from work, she likes to turn on music and plays with her new multi-color light-bulb. Instead of saying “Brassau, turn down the volume” over and over again, she uses the Brassau widget to change the slider until the sound is just right (Figure 2g). Instead of limiting herself to the choice of colors that she can name, she uses the Brassau widget to pick the hue from a color picker (Figure 2h). This shows how Brassau helps users with interactive controls.

Ann likes to watch a video before she goes to bed. She knows that Brassau can connect YouTube to her smart TV, so she asks Brassau to “search ‘recipes with the Swedish chef’ on YouTube then play it on my TV”. Brassau produces a widget that lets Ann substitute in a new title on subsequent nights quickly (Figure 2i). This shows how Brassau can produce widgets for compound commands.

The goal of Brassau is to let anybody put together a unique dashboard of widgets tailored to their interests and habits easily, in natural language. In summary, Brassau can help with personalized notifications, navigation of long results, reusing commands, glanceable commands, repeated tasks, background tasks, interactive controls, and compound commands.

**ALGORITHM DESIGN PROCESS**

We used an iterative design process to come up with the solution proposed in this paper. We began with need finding by looking at what people wanted their virtual assistant to do. We compiled a list of the most popular commands for the Almond virtual assistant, and an author of the paper sketched out what a GUI would look like if each was laid out by a designer. The results, shown in Figure 3, highlighted the need of a good layout for elements corresponding to the inputs and outputs and a semantically relevant and interesting background to make the interface visually attractive.

Our first attempt to create a GUI automatically was to assemble it from the bottom up, starting with the program specification.
We now give an overview of how Brassau converts a command into an executable program in ThingTalk. From that, Brassau extracts the list of inputs and outputs to show in the widget. It finds the best matching template from the database. It fills the template, and stylizes it to maintain visual consistency. The interactive elements are hooked up to the ThingTalk runtime in Almond to execute the widget.

**USER-INTERFACE TEMPLATES**

Brassau’s Algorithm lays out controls and display elements in the widget using the concept of templates, which are semantically tagged images with replaceable areas. Here we present the BANANA language and insights on what images make useful templates.

**The BANANA Template Markup Language**

To create a visually interesting and semantically relevant GUI, the BANANA markup language captures the semantics of the image, the layout for the inputs and outputs, and the color information. The formal definition of BANANA is provided in Figure 5. A BANANA template consists of an image, a list of tags, a list of boxes for placing inputs and outputs, the corner colors of the image, and the color palette of the image. Each tag is a word, and can be required or optional. Required tags have a positive weight when present in the program and a negative weight otherwise. They let us include very specific images, which work great if they match and are disconcerting otherwise. For example, Slack colors should only be used for Slack widgets, and not Google. Optional tags have a positive weight when present, and zero otherwise.

The template’s color palette is used to choose foreground colors, such that elements can be made to match the color scheme of the image. Finally, the corner colors are used to colorize the buttons placed near the corners.

Each box is represented as a pair of points in the coordinate space of the image. Associated with each box is its dominant color; this is used to choose a contrasting color for elements overlaid on the box. In addition, the average color along each side is included. Our matching algorithm may choose a template with more boxes than needed, and the image might have content inside the box. We would like to be able to cover unused boxes, so we introduce the “cover” attribute; its presence indicates that the box must be covered up. The side colors are used to construct a gradient that covers the box and blends with the rest of the image.

Each box has a type attribute, such as "image", "text", or "button", which determines how it is supposed to be used. It also includes a rank ordering. Intuitively, larger and more prominent boxes have a lower order, because they should be used first. A box can be also annotated with text alignment, font size and font family, from a set of 5 predefined families. These annotations exist to give a template designer more control over the output of Brassau.

Brassau provides a graphical interface so users can tag their images with individual boxes and labels. All the color information about the image and the boxes is automatically generated. The dominant color in each box is the most frequent color in the box, and the color palette of the background is constructed

---

Brassau’s Adaptive Non-specific Annotations for Novel Assistants

We analyzed the program and mapped the input and output parameters to graphical elements in a repository, such as icons, buttons, colors, and backgrounds. For interactive controls, we relied on the standard browser controls, combined with the Bootstrap library of components [24]. We set the background color to a gradient of colors extracted from the logo. When put together, the elements did not match with each other and looked visually inconsistent, since they had different styles and a varied level of quality. Overall, the designs looked too formulaic and boring, nothing like the sketches made by hand. This experience taught us that more entropy is needed in the design process to create a visually interesting set of widgets.

In response, we investigated using a top-down, knowledge-intensive approach. Can we leverage the large corpus of existing images and apps to create diverse, well-designed, and semantically relevant GUIs? Using a test set of 91 different virtual assistant commands, we experimented with different images to understand what makes a good GUI template and what information is needed to match templates to commands automatically. From the learning, we created a template markup language called BANANA, and an interactive tool so users can provide the necessary information quickly. We incrementally built up a small collection of 65 templates, refined BANANA, improved Thingpedia descriptors to provide needed information, and derived heuristics to find the best matching template and fill the template.

Extensibility is particularly important as the variety in virtual assistant commands will grow with more devices, services, and more sophisticated compound constructs. Our approach allows Brassau to scale up by simply increasing the template database. Our BANANA interactive tool is easy enough for non-developers to create templates. Furthermore, this approach is amenable to machine learning.

**OVERVIEW**

We now give an overview of how Brassau converts a command in natural language into an interactive widget. We shall use the command “search tweets about "pierre brassau"”, shown in Figure 4, as a running example throughout the rest of the paper. Brassau first uses Almond to translate the command

---

Figure 3: Sketches used to lead the design of the templates.
by the modified mean quantization algorithm [4], which clusters colors according to their distance in the RGB space.

Template t
: \text{img tag} : \text{box} \text{ cc} \text{ pal}

Image img
: \text{<pictures>}

Tag tag
: \text{[required]} \text{optional} \text{<word>}

Box box
: \text{Pop-left} \text{ Photon-right} \text{ Cdon} \text{ Ctop Cleft Cbottom Cright}

type h align? fsfz? fffam? cover?

Point p
: \text{<x> <y>}

Color c
: \text{<r> <g> <b>}

Type type
: \text{input} \text{ text} \text{ image} \text{ title} \text{ map}

| slider | button | switch | logo |

Hierarchy h
: \text{<number>}

Alignment align
: \text{left} \text{ right} \text{ center} \text{ justify}

Font Size fsz
: \text{<number>}

Font Family fffam
: serif \text{ sans} \text{ monospace} \text{ handwritten} \text{ display}

Corner Colors cc
: \text{top-left} \text{ top-right} \text{ bottom-left} \text{ bottom-right}

Palette pal
: \text{c}

Figure 5: The formal definition of BANANA.

Template Collection
We collected 65 templates, as shown in Figure 6. The process of coming up with templates was driven by identifying a functionally diverse set of devices within Thingpedia. We identified four categories of templates: screenshot-based, skeuomorphic, semantic, and neutral.

Skeuomorphic Designs
As widgets are specialized versions of existing apps, it is natural and appealing for them to adopt the style of the original, professionally designed apps. An example of such a design is the email widget in Figure 2c. A template in this category can be obtained as a screenshot of the official app or website. Thingpedia contributors may wish to supply templates for their apps in the future. Annotating these templates is easy because the images have clearly delimited interactive areas, and could be done automatically. Not all APIs have GUIs; for example, the Philips Hue light-bulb allows a “disco lights” mode (which continuously changes the color), but the Hue app does not include this feature. For these types of apps we need other user interfaces as we will describe in the following sections.

Almond
Figure 4: An example of generating a widget with Brassau.

Semantic Designs
Semantic designs contain graphic elements that share semantics with the programs, such as specific images or brand decorations. Examples are shown in Figures 2e and 2f. Semantic designs have the advantage that they can be used for many different programs.

Neutral Designs
The algorithm might not be able to find a suitable template for a program in all cases. Thus, we devised a set of neutral templates for the most common program signatures. They are intended to act as a “catch all” for APIs that do not have a more semantic display in the database. These templates have a flat color background, manually placed controls, and no decorations. The set of neutral designs will likely remain small in number.

A FUNCTIONAL BRASSAU
Brassau is created by substituting the text-based interface of Almond with automatically generated GUIs. We now describe how we create a functional Brassau, and subsequent sections will describe how we make Brassau visually attractive.

Overview of Almond
Let us first describe Almond, the framework on which Brassau is built. Almond uses Thingpedia, a crowdsourced and open-source repository that stores the necessary information on IoT devices and web services to support natural language translation as well as the execution of the commands. Thingpedia has three kinds of primitive commands: WHEN functions determine when the program should run; GET functions retrieve data; and DO functions perform an action or display data to a user. Some examples of each are shown in Figure 7.

Each function entry in Thingpedia has a definition of its API and implementation. The API declaration includes a list of input parameters, which may be optional, and a list of output parameters. Parameters have a name and a type; values are
Almond translates natural language into ThingTalk. Almond uses a formal language called ThingTalk to let users program. The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user to resolve any ambiguity or defects in parsing. The Twitter entry, used in the example in Figure 4, is shown in Figure 8.

Almond uses a formal language called ThingTalk to let users connect these primitives. Each ThingTalk program is a single `when-get-do` statement, which combines up to three primitive calls: `when` and `get` functions return data which can be passed to the `do` function. Data returned can also be filtered through predicates such as equality or string containment.

The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user for credentials of the device or account if they have not been entered already, executes the program and passes the results to the chat interface to present to the user.

### Brassau Widgets

Brassau uses ThingTalk to interface with the Almond framework. Almond translates natural language into ThingTalk, and passes it to Brassau. Brassau then reads Thingpedia entries to interpret the input and output parameters, and creates a GUI so users can add or modify input parameters. The updated ThingTalk program is passed to the Almond runtime; results passed in as keyword-value pairs. While output parameters have a scalar type, it is assumed that each call may return a set of such results in a list. Names and types are significant as they are used by the semantic parser. A Thingpedia entry also contains a long description of the function, which can be used to reconstruct a canonical natural language representation of each program. This description can be shown to the user to resolve any ambiguity or defects in parsing. The Twitter entry, used in the example in Figure 4, is shown in Figure 8.

Almond uses a formal language called ThingTalk to let users connect these primitives. Each ThingTalk program is a single `when-get-do` statement, which combines up to three primitive calls: `when` and `get` functions return data which can be passed to the `do` function. Data returned can also be filtered through predicates such as equality or string containment.

The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user for credentials of the device or account if they have not been entered already, executes the program and passes the results to the chat interface to present to the user.

### Brassau Widgets

Brassau uses ThingTalk to interface with the Almond framework. Almond translates natural language into ThingTalk, and passes it to Brassau. Brassau then reads Thingpedia entries to interpret the input and output parameters, and creates a GUI so users can add or modify input parameters. The updated ThingTalk program is passed to the Almond runtime; results passed in as keyword-value pairs. While output parameters have a scalar type, it is assumed that each call may return a set of such results in a list. Names and types are significant as they are used by the semantic parser. A Thingpedia entry also contains a long description of the function, which can be used to reconstruct a canonical natural language representation of each program. This description can be shown to the user to resolve any ambiguity or defects in parsing. The Twitter entry, used in the example in Figure 4, is shown in Figure 8.

Almond uses a formal language called ThingTalk to let users connect these primitives. Each ThingTalk program is a single `when-get-do` statement, which combines up to three primitive calls: `when` and `get` functions return data which can be passed to the `do` function. Data returned can also be filtered through predicates such as equality or string containment.

The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user for credentials of the device or account if they have not been entered already, executes the program and passes the results to the chat interface to present to the user.

### Brassau Widgets

Brassau usesThingTalk to interface with the Almond framework. Almond translates natural language into ThingTalk, and passes it to Brassau. Brassau then reads Thingpedia entries to interpret the input and output parameters, and creates a GUI so users can add or modify input parameters. The updated ThingTalk program is passed to the Almond runtime; results passed in as keyword-value pairs. While output parameters have a scalar type, it is assumed that each call may return a set of such results in a list. Names and types are significant as they are used by the semantic parser. A Thingpedia entry also contains a long description of the function, which can be used to reconstruct a canonical natural language representation of each program. This description can be shown to the user to resolve any ambiguity or defects in parsing. The Twitter entry, used in the example in Figure 4, is shown in Figure 8.

Almond uses a formal language called ThingTalk to let users connect these primitives. Each ThingTalk program is a single `when-get-do` statement, which combines up to three primitive calls: `when` and `get` functions return data which can be passed to the `do` function. Data returned can also be filtered through predicates such as equality or string containment.

The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user for credentials of the device or account if they have not been entered already, executes the program and passes the results to the chat interface to present to the user.

### Brassau Widgets

Brassau uses ThingTalk to interface with the Almond framework. Almond translates natural language into ThingTalk, and passes it to Brassau. Brassau then reads Thingpedia entries to interpret the input and output parameters, and creates a GUI so users can add or modify input parameters. The updated ThingTalk program is passed to the Almond runtime; results passed in as keyword-value pairs. While output parameters have a scalar type, it is assumed that each call may return a set of such results in a list. Names and types are significant as they are used by the semantic parser. A Thingpedia entry also contains a long description of the function, which can be used to reconstruct a canonical natural language representation of each program. This description can be shown to the user to resolve any ambiguity or defects in parsing. The Twitter entry, used in the example in Figure 4, is shown in Figure 8.

Almond uses a formal language called ThingTalk to let users connect these primitives. Each ThingTalk program is a single `when-get-do` statement, which combines up to three primitive calls: `when` and `get` functions return data which can be passed to the `do` function. Data returned can also be filtered through predicates such as equality or string containment.

The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user for credentials of the device or account if they have not been entered already, executes the program and passes the results to the chat interface to present to the user.

### Brassau Widgets

Brassau uses ThingTalk to interface with the Almond framework. Almond translates natural language into ThingTalk, and passes it to Brassau. Brassau then reads Thingpedia entries to interpret the input and output parameters, and creates a GUI so users can add or modify input parameters. The updated ThingTalk program is passed to the Almond runtime; results passed in as keyword-value pairs. While output parameters have a scalar type, it is assumed that each call may return a set of such results in a list. Names and types are significant as they are used by the semantic parser. A Thingpedia entry also contains a long description of the function, which can be used to reconstruct a canonical natural language representation of each program. This description can be shown to the user to resolve any ambiguity or defects in parsing. The Twitter entry, used in the example in Figure 4, is shown in Figure 8.

Almond uses a formal language called ThingTalk to let users connect these primitives. Each ThingTalk program is a single `when-get-do` statement, which combines up to three primitive calls: `when` and `get` functions return data which can be passed to the `do` function. Data returned can also be filtered through predicates such as equality or string containment.

The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user for credentials of the device or account if they have not been entered already, executes the program and passes the results to the chat interface to present to the user.

### Brassau Widgets

Brassau uses ThingTalk to interface with the Almond framework. Almond translates natural language into ThingTalk, and passes it to Brassau. Brassau then reads Thingpedia entries to interpret the input and output parameters, and creates a GUI so users can add or modify input parameters. The updated ThingTalk program is passed to the Almond runtime; results passed in as keyword-value pairs. While output parameters have a scalar type, it is assumed that each call may return a set of such results in a list. Names and types are significant as they are used by the semantic parser. A Thingpedia entry also contains a long description of the function, which can be used to reconstruct a canonical natural language representation of each program. This description can be shown to the user to resolve any ambiguity or defects in parsing. The Twitter entry, used in the example in Figure 4, is shown in Figure 8.

Almond uses a formal language called ThingTalk to let users connect these primitives. Each ThingTalk program is a single `when-get-do` statement, which combines up to three primitive calls: `when` and `get` functions return data which can be passed to the `do` function. Data returned can also be filtered through predicates such as equality or string containment.

The Almond virtual assistant accepts a natural language command and translates it into ThingTalk using a machine-learning algorithm. The ThingTalk runtime system prompts the user for credentials of the device or account if they have not been entered already, executes the program and passes the results to the chat interface to present to the user.
program. The title of a command is formed by combining the short descriptions of all primitives in the program. If the title is identical to the action button, it is not shown. We describe how inputs and outputs are handled below.

### Input Parameters
A ThingTalk program, translated from natural language, is often underspecified due to missing input parameters. Therefore, the user interface needs to accommodate extra input parameters needed to run the program. Brassau maps the input parameter types in Thingpedia into a control element, according to the mapping shown in Figure 9. For numeric types, a slider is used if the range is known or can be inferred by the unit of measure. For enumerated types, radio buttons are used if there are three options or fewer, otherwise we show a drop down menu to preserve space in the widget. As a special case, if the enumerated choices are “on” and “off”, a switch is shown instead. Entity types, such as company names and languages, are shown with an icon and a name from Thingpedia when specified. See for example the language entity “Chinese” in Figure 2d. If the parameters are not specified, the user is shown a drop down menu with all the options. For compound commands, the icon of the primitive associated with the parameter is shown, as in Figure 2i. Similarly, for hashtags and usernames, a # or @ symbol is shown before the input.

Whenever the values of the input parameters are changed via the GUI, the program is correspondingly modified, and a new canonical sentence is shown on the top box (Figure 2a).

### Output Parameters
Brassau maps output parameter types in Thingpedia into display elements, as shown in Figure 10. For example, parameters of type URL are turned into clickable links, and parameters of type Measure have their unit shown. Arrays are hidden because their contents are redundant with other parameters. Figure 4 shows how parameters in the Twitter example are mapped to control and display elements. Information in the array types is already present in the “text” output.

For ThingTalk functions that return multiple results, each result is mapped to a different page of the widget. The user can switch pages with left and right arrows, and the badge at the top indicates the number of unread results.

### TEMPLATE MATCHER
To match a program with the best template in the collection, we calculate a heuristically defined cost function for each template and choose the one with the minimum cost. Let $C(t, p)$ be the cost function for a program $p$ using template $t$ in collection $T$, and $\hat{t}$ be the best template for $p$.

$$C(t, p) = \alpha S(t, p) + \beta H(t, p) + \gamma F(t, p) \quad (1)$$

$$\hat{t}(p) = \arg\min_{t \in T} C(t, p) \quad (2)$$

where $\alpha$, $\beta$ and $\gamma$ are heuristically assigned coefficients. $S$ is a semantic similarity measure of the template to the program, computed using the template tags and the words in the program description. $H$ is the hue difference of the background and the program logo. $F$ is computed by the algorithm described below to measure the fit between parameters and boxes.

We first observe that only unspecified and required input parameters and one output parameter need to be displayed in the GUI. Including other input parameteres increases the widget’s generality but is not necessary. After all, a user can always change the top input box above the widget directly, in text.

Our algorithm orders the inputs and outputs based on their priority, described below. In order of priority, it assigns each element to a box of matching type, using the table shown in Figure 11, where possible. If there are unassigned elements, it
tries to assign them to boxes of a different type, again in priority order. A cost is assessed for every mismatched assignment, and for every unassigned element and unassigned box.

As discussed above, unspecified, required input parameters have the highest priority. In fact, an infinite cost is assessed because the program will not run without them. Output parameters are next in priority, after all that is the purpose of widget. Priority is given to picture and video output types since Brassau is a graphical virtual assistant. Interactive elements, such as switches and slides, have higher priority than input texts as they often serve the primary function of the widgets. Lastly, hard-to-specify inputs have higher priority since the user is more likely to vary free-form text than short entities. Figure 4 shows an example assignment of elements to boxes in a template; note that “text 3” happens to be empty in this tweet.

**STYLIZING THE APPLICATION**

The final step of creating the widget is to style it. To color the input and output elements, we seek colors that are legible and look good within the context of the overall design. We determine this programmatically by calculating the contrast ratio between each of the palette colors and the color of the box the element will be filling. To calculate the contrast, we compute the ratio between the luminance of the two colors.

The palette color with the highest contrast is chosen to be the foreground color for the box. This makes the text match the other decorative elements in the template, as in Figure 2e. If none of the colors have a contrast ratio more than 2.5, a generally accepted usability guideline to ensure readability, we choose the color using a split-complementary color scheme. This scheme is known to have enough contrast.

To display as much of the text as is reasonable, we adjust the font size dynamically based on the content, with minimum and maximum sizes, to best fit within the box. This displays shorter information, such as stock price or steps, more prominently.

**EVALUATION**

We have created two full working prototypes of Brassau, a web version and an Android version. We used the LabelImg tool [35] to annotate the templates, and the Color Thief library [7] to extract the color palette. Brassau is open-source.

We evaluate if Brassau’s Algorithm, coupled with an extensible database of templates, can be used to create functional and visually appealing widgets for an open-world of virtual assistant commands. In the following section, we first present the baseline program and template sets, and describe five experiments to evaluate the correctness of Brassau’s Algorithm, ease of template creation, visual appeal, match with the program intent, and end-to-end experience.

**Baseline Programs**

As discussed previously, we experimented with a Baseline Programs set of 91 virtual assistant commands in the development of our algorithm. The Baseline Programs represent a large portion of what Almond can do without replicating commands that are overly similar. A few Thingpedia primitives are dropped from our study either because they are similar to commands in the set, or we do not have access to the device to test them. Brassau produces working widgets for all these 91 test cases.

There is a great variety in this Baseline: 54 use private data and 37 use public resources. 23 commands are WHEN, 34 are GET and 34 are DO operations. 41 of the generated widgets are interactive, showing one or more input parameters that can be changed, and 7 of which include either a switch or a slider. Brassau hides one or more inputs for 15 of the widgets. 16 widgets show a list for results, and the text content in 14 widgets is too long and only a snippet is shown.

We have created a Starter Template Set of 65 templates. The wide variety displayed supports the need for a knowledge-based approach Figure 6. There are 20 screenshots, 11 skeuomorphic, 25 semantic, and 9 neutral designs. 27 templates have a strong brand association (such as the use of brand colors or additional logos) and 29 have semantic decorations. Additionally, 5 UI screenshots (among which, SMS and weather) have no branding differentiation, making them reusable for other purposes.

Of the 24 semantic displays, 3 include a real picture of the device they represent, and the rest use a drawing or flat colors. Of the other 32 non-neutral templates, 3 are monochrome or line-art, 20 use flat colors or simple gradients, and 9 use realistic colors. 25 of the templates include at least one input box, 34 at least one text box, and 14 at least one image box. 6 templates are annotated with an explicit position to place the logo, which otherwise would be placed at the upper left position. 34 templates have a box for the action button to execute the program.

**Correctness of Brassau’s Algorithm**

Can Brassau’s Algorithm handle new virtual assistant commands correctly with a small template set? While we fully expect that many templates are necessary to create good-looking widgets for new commands, we want to know if the algorithm can create functionally acceptable widgets without more templates. We created two new program sets, as follows.

The Partial Programs set contains 83 programs derived from the base command set by dropping one or more of the parameters. Commands with more than 2 parameters generate several alternatives, whereas commands with just one meaningful input are skipped. This set of commands forces Brassau to work harder because there are more input elements to display.

The Compound Programs set contains 12 compound programs, designed to evaluate if Brassau can handle programs more complex than the primitives it was tuned with.

**Neutral Templates**

Our first test uses just the 9 neutral templates, which are used as a catch-all solution when no better templates match. We applied the neutral templates to each of the Baseline, Partial and Compound Programs set. Brassau built functional apps for all the 186 programs, except for 4 in the Baseline and 18 in the Partial, because it cannot find a template that matches the signatures of the commands. We need more neutral templates.
that carry more outputs, and also outputs combined with the input type of a map or an image. This suggests that we may wish to synthesize a neutral template from required inputs and outputs, if one does not exist, to ensure completeness.

Visually, ignoring the lack of color but focusing on the layouts, 45% of the programs (38 Base, 39 Partial, 6 Compound) are as good as the full template set. 20% (28 Base, 9 Partial) are reasonably good, and the rest are decent but not great.

**Starter Template Set**

We apply the full Starter Template Set on the Partial Programs. For 66 programs in the Partial Set, the widget looks and behaves identically to the fully specified version, as Brassau had already created input elements for those inputs when given the original programs. Brassau generated 17 widgets with different templates from the original one. While Brassau produces a working widget for all the programs, unlike in the case of neutral templates, the semantic match for 9 of the problematic programs was poor, due to a lack of choice.

Brassau creates functional widgets for all the Compound Programs. In 9 out of the 12 cases, Brassau creates a high quality widget, choosing the best template and making the best use of it. Brassau uses a neutral template for 1 because that is the only one matching the signature. And for the remaining 2 cases, Brassau produces the template for the DO primitive, with no WHEN parameters, for a lack of choice of better templates.

The results show that the ranking and matching heuristics in Brassau’s Algorithm can handle a wide range of virtual assistant commands. The current weakness in the system lies in the small size of the template database, which can be increased easily, as discussed next.

**Ease of Template Creation**

To test how easy it is to add templates to the system, we recruited 5 students, 2 of which had experience in design and 3 which didn’t, as users to add a template of their choice to the database. Users chose their least favorite template from the Baseline Programs, and after reading a page of instructions they redesigned it. They looked for an appropriate image from the web, then tagged it. Each user completed the task in less than ten minutes. One user commented, “All I did was make three boxes and rank the order and it was done. It knows what is the most important and does it.” All were satisfied with the final result, and said that they would use the tool to build interfaces for personal use.

**Visual Appeal of Templates**

To evaluate if the use of templates improves the visual appeal of widgets, we devise the following experiments. We vary the graphical presentation of the widgets as follows:

- **Plain.** The input and output boxes selected to be presented by Brassau are laid out, centered, sequentially from top to bottom, and with no background.
- **Layout.** The inputs and output are placed into the boxes of the best-fit template. However, the elements are not styled and the background is not shown.
- **Brassau.** The full algorithm is used, which includes the background and the custom layout.

We chose 39 of the commands from the original test programs, and created 6 sets of 39 widgets each. For each set, the 39 commands are randomly partitioned equally among the widget generation methods, and the resulting widgets are randomly shuffled. We then present each set of widgets to 40 Amazon Mechanical Turk workers. Users are then asked to vote for the top 3 widgets that they would like to use. After filtering for invalid responses, we collect data from 227 users.

Overall, Brassau’s widgets receive 62% of the votes for top 3, the layout method receive 16%, and the plain widgets receive 22%. We analyze the votes, and plot for each presentation method, the percentage of people who voted for it 3, 2, 1, and 0 times in Figure 12a. Less than 1% chose all their favorite widgets from the Layout method, and 1.5% chose only Plain widgets. In contrast, 29% chose all their favorite widgets from Brassau’s widgets, with 93% of the people picking at least once from Brassau. This suggests that Brassau’s Algorithm creates more attractive widgets.

**Suitability of the GUI Interface**

In this test we wish to determine if Brassau’s generated GUI captures the semantics of the command. A natural language command is shown to the users together with the GUI it generates. They are then asked to rate how well the GUI matches the intent of the command using a 5-point Likert scale, from strongly disagree (1) to strongly agree (5).

Because some of the services in the 91 commands were not available online at the time of the experiment, the test is done on 76 of the commands; 100 Mechanical Turk workers are each shown 40 GUI command pairs. From the 400 ratings received, 11% are 1 or 2, 13% are 3, and 76% are 4 or 5. The average over all the ratings is 4.0.

Commands that show a semantic link between the command and the template have the least variance in the ratings. For widgets that show a picture of a word in the command, such as a dog, thermostat, or tv, the standard deviation is between 0.6 and 0.7. The five templates with the highest standard deviation (1.2-1.4) are in the bottom 10 of overall matches, suggesting that there is more differences in opinions at the low end. Two have skeuomorphic templates, two are semantic with “flat” designs, and one is a neutral template.

Widgets with less colors tend to have lower ratings. Four of the five lowest pairs are black and white or neutral, the fifth has a
solid yellow background. Commands that contains uncommon words, such as “xkcd” and “meme”, tend to have lower ratings, perhaps because workers are unfamiliar with the command or how it should look like. The 5 highest ranked pairs have more variety in their colors; they are either screenshots, or use a semantic template that contains an image of the object. None of the top 10 commands use a neutral template. As summarized in the histogram in Figure 12b, the users finds the majority of the widgets acceptable.

End-To-End Experience
Brassau is built on top of Almond, an experimental virtual assistant. As such, its limited functionality and natural language understanding do not allow us to evaluate Brassau in practice. We conducted a small user study where five students tried our system end-to-end, starting with natural language inputs. We presented users with a cheat sheet showing 350 \texttt{WHEN}, \texttt{GET}, and \texttt{DO} example commands and asked them to try entering commands into the system. We then showed them the 91 widgets from the base command set, and asked for feedback. Evaluations lasted 30 minutes.

The users gave some concrete suggestions on the visual presentation; P3 wanted the news widget to contain an image as well as text and P2 wanted it to use the real New York Times font. They observed that the most visually appealing widgets had strong decoration and short pieces of content. They specifically liked the Uber widget, which shows a car drawing stylized with the Uber colors. All users commented that they wanted the minimized widgets to be smaller, so that the main widget would have more room to display information. P3, P4, and P5 commented that the list display would be better if they could swipe instead of clicking left and right, and found that the unread indicator was not well positioned. We will incorporate these inputs into the system in the future.

When shown the list of 91 widgets, different users found different apps useful, depending on what devices they owned. Examples of widgets they liked include the music player, translation, memes, and the light bulb. While one liked the Uber widget, another did not because they would rather use the original app. One liked the slider to set a volume, one did not. The difference in opinion of what is good demonstrated the importance of users being able to personalize their dashboard.

Users were satisfied and felt that the commands they entered were well represented by the GUIs. P1, P3, and P5 commented that they liked the simplicity of the GUIs; “it’s clean, easy to use, only a few buttons, and I like the boxes”. Brassau did not work in some cases due to errors in Almond’s parsing.

Two of the users commented on the overall usefulness of a dashboard of customized widgets. P1 liked the system and described Brassau as a “phantasmagoria of functions.” P2 commented that the system was like Facebook because they could do many tasks within the same app. They liked to see the news and chat with friends in the same app. They would like to piece together a custom app for all their favorite functions.

Four of the users, P1, P3, P4, and P5 wanted to do more with the content once it was displayed. P1 tried a compound command that would upload newspaper articles to Facebook, and was confused that they did not have the opportunity to change what would be posted. They wanted additional actions such as being able to share the images and seeing other \textit{related content}. Confirmation, sharing, and other general functionalities can be baked into the Brassau framework in the future and made available to all the widgets.

LIMITATIONS AND FUTURE WORK
Our evaluation of Brassau’s limitations identified ways in which the Thingpedia type system can be improved. Currently Thingpedia has a very simple type system. We recommend adding the concept of range and resolution to numerical inputs, so we can choose whether to use a slider or a text box. We also recommend adding a concept of records to Thingpedia, to make relationships of data explicit. For example, sport scores are represented by a pair of teams and a pair of numbers; records provide a way to associate the team with its score.

Brassau sometimes does not utilize the space well because it does not know the size of the returned value, which can change dynamically. For example, an image returned from the Cat API could be either horizontal or vertical, and a generic square box for the image may waste space. We plan to explore adapting the layout after receiving the return value in the future.

Brassau’s Algorithm currently can only represent one ThingTalk program in one widget. Our end-to-end user test suggests that people prefer the ability to perform additional interactive actions. Future work will consider showing contextual actions in a widget, and how users can combine multiple ThingTalk programs to make the best use of interactivity.

CONCLUSION
This paper introduces Brassau, a graphical virtual assistant that automatically generates interactive widgets from natural language commands, translated by Almond to ThingTalk. Brassau allows users to assemble together a dashboard of personalized widgets, making it easier to repeat commands and see the results.

Brassau introduces a novel template-based method to leverage the large corpus of existing images and apps to make the widgets visually diverse and interesting. Users can turn an image into a template easily by marking up the bounding boxes for control and display elements, ranking them, and adding semantic tags to the image. The template also contains color information automatically extracted from the image. Brassau matches the ThingTalk program to a template and stylizes it to create the GUI. Brassau presents the first template system which matches its program dynamically based on types and semantics. Brassau’s Algorithm can improve as more templates are added, with no developer effort.

Our experiments show that Brassau’s Algorithm is general and supports the diverse space of virtual assistant commands, even with a limited template set. User tests show that in 62% of the cases, users prefer the visually rich interface created by Brassau, and in most cases the user interface produced by Brassau is appropriate to the command. Preliminary end-to-end user tests suggest that the system is useful, easy to understand and easy to extend.
REFERENCES


