The Design and Implementation of Stateful ConvoBlocks

Mingyan Claire Tian

Submitted in Partial Fulfillment
of the
Prerequisite for Honors
in Computer Science
under the advisement of Professor Franklyn Turbak

May 2023

©2023 Mingyan Claire Tian

The Design and Implementation of Stateful ConvoBlocks

by

Mingyan Claire Tian

Submitted to the Computer Science
on April 14, 2023, in partial fulfillment of the
requirements for the degree of
Bachelor of Arts

Abstract

Conversational AI is becoming ubiquitous in our world today. With the increasing complexity of voice-first technology, there is a pressing need for pedagogical tools to develop conversational AI applications. ConvoBlocks (aka the MIT App Inventor Conversational Agent Platform) is a web-based block-programming interface that enables low-code creation of conversational agents (CAs) that runs on real smart-home devices. However, the interface only supports universal accessibility for intents and entities and carries no contextual information, which can be restraining and cumbersome when specifying multi-branch conversation. Other commercial CA programming interfaces like Google Dialogflow CX use a state-machine agent design that promotes naturalness of conversations.

Recognizing the advantages of state-intent associations in CA development, I introduce Stateful ConvoBlocks, an extended model of ConvoBlocks that supports a one-to-many state-intent relationship as well as context-based intent screening. With a learner-centered design mindset, I implemented a state-intent mapping in the back-end, a tabbed view of component states in the front end, as well as the *when-intent-spoken* block and a *set-current-state* block that handles the logic of state-switching in a conversation flow. To investigate the usability and learnability of Stateful ConvoBlocks, I also designed a user study testing whether the new interface design improves the expressiveness of App Inventor in specifying complex conversations along with the partner model between learners and the new interface. Results from the study will inform both design and pedagogy guidelines for the Stateful ConvoBlocks interface.

Acknowledgments

I'd like to express my deepest gratitude and appreciation to all the people who have supported and encouraged me directly and indirectly throughout the thesis process. First and foremost, I would like to thank my advisors: Lyn Turbak, Hal Abelson and Evan Patton.

Lyn, thank you for introducing me to the world of research and supporting me through the journey. I still remember meeting you for the first time in my Freshman year, and being amazed by your whole-hearted passion for programming languages and excitement for teaching. Thank you for helping me discovered my academic passion, for your expertise, your feedback, and your late-evening meetings to accommodate my schedule. I would not have made it to this point if not for you.

Hal, thank you for showing me your love for empowering kids in computation, for guiding me through all of my academic explorations, and for encouraging me to reach out and communicate with researchers across different teams. Your undying curiosity and drive for learning will always be an inspiration to me.

Evan, thank you for your incredible support during the implementation process for this thesis. Thank you for helping me plan out the implementation timetable given the time constraints and for always being patient when debugging. I've learned so much from your coding style and your thinking process.

I am also grateful for my committee: Catherine Delcourt and Carolyn Anderson. Thank you for taking interest in this project and for spending time to provide insightful ideas and helpful feedback.

Thanks to the staff and student researchers of the App Inventor team. Thank you for sharing your knowledge with me and for supporting me through the process of building and testing the new project.

Finally my thank goes to my family and friends for their unwavering support, love, and encouragement. Their patience, understanding, and encouragement have been a constant source of motivation and inspiration.

Contents

1 Introduction **15**

1.1 MIT App Inventor and the K-12 Conversational Agent Framework 16

1.2 Challenges Facing ConvoBlocks Programmers 17

1.3 Existing Mechanisms to Solve These Challenges 17

1.4 Introducing States 18

1.5 Contributions 18

2 Background and Related Work **21**

2.1 Conversational AI 21

2.2 Educational Initiatives for Conversational AI 23

2.3 Programming Conversational AI Platforms 24

2.3.1 Current ConvoBlocks Interface 24

2.3.2 Alexa Developer Console 26

2.3.3 Google Dialogflow CX 27

2.3.4 Voiceflow 29

3 Design of State-based Intents **31**

3.1 Design Motivation 31

3.1.1 Dress-ordering Service 31

3.1.2 Little Red Riding Hood 33

3.2 Design Round 1 34

3.2.1 Separating Slot-filling Dialogues 34

3.2.2 State Machine Representation of Conversation 36

3.3 Design Round 2 37

3.3.1 Representing States by Different Blocks Editor Windows 37

3.4 Final Design . . . 38
3.4.1 State/intent Association . . . 38
3.4.2 Designer View: Tabs . . . 39
3.4.3 Blocks Editor: Contexualized Block Name . . . 42

4 Implementation of Final Design **47**
4.1 State Intent Association with Tabs . . . 48
4.2 Blocks Interface . . . 50

5 User Study Design **55**
5.1 Research Questions . . . 55
5.2 Participants . . . 56
5.3 Workshop Design Outline . . . 56
5.3.1 Day 1 . . . 57
5.3.2 Day 2 . . . 58
5.3.3 Day 3 . . . 60
5.3.4 Data Collection . . . 60
5.3.5 Data Analysis . . . 61

6 Summary of Contributions and Future Work **63**
6.1 Summary . . . 63
6.1.1 Current Version . . . 63
6.1.2 Known Limitations . . . 64
6.2 Future Work . . . 65
6.2.1 Immediate Future . . . 65
6.2.2 Near Future . . . 66
6.2.3 Far Future . . . 67

A Implementations for State-based ConvoBlocks **69**
A.1 Tabs . . . 69
A.1.1 DesignerEditor.java . . . 69
A.1.2 AlexaDesignEditor.java . . . 69
A.1.3 AlexaNonVisibleComponentsPanel.java . . . 76
A.1.4 SimpleVisibleComponentsPanel.java . . . 77

A.1.5 SimpleNonVisibleComponentsPanel.java 77
A.1.6 MockIntent.java 77
A.1.7 MockAlexa.java 78
A.2 Blocks 78
A.2.1 voice.js 78
A.2.2 alexa.js 79
A.2.3 components.js 80

List of Figures

2-1 Conversational AI architecture [12]. 22

2-2 The current ConvoBlocks interface. (a) The Designer view. (b) The Blocks editor. 25

2-3 The ConvoBlocks interface highlighted according to design goals [30]. The blocks here creates a storybook agent. 26

2-4 (a) An example representation of four flows, the "Default Start Flow", a "Customer Information" flow, a "Food Order" flow, and a "Confirmation" flow. (b) Within the "Food Order" flow, a user might develop pages (e.g., "Edit Order" page, "Add Pizza" page, "Add Drink" page, etc.). [26]. . . . 29

3-1 An example conversation in which an end user tests the clothing store agent that developers create in the Dialogflow tutorial [26]. In the first turn, the agent tries to collect information about the size of a shirt, but the end user invokes the greeting intent. The agent then repeats its greeting and question about shirt-size, but the end user responds with the color of the shirt. The agent repeats its question again, and the end user provides the size. Finally, the agent asks about the color of the shirt and the end user responds with a color, and the agent confirms the end user's order. When the end user initially responds with a color to the agent's question about size, the agent does not recognize the color since the color intent was out of scope at that point in the conversation. This is due to flow and page modularization. 33

3-2 Mockup for the designer interface in initial design. The Echo Dot image and non-visible components panel is replaced by a state-machine graph to illustrate the conversational flow. 36

3-3 Current display of the top half of the ConvoBlocks interface. *Purple:* The purple box highlights a panel called "design tool bar". *Yellow:* The yellow box highlights the dropdown list containing all screens and Alexa skills the developer created for this project. 38

3-4 The association between states and intents in Lorina's *Dress-ordering Service* agent. 39

3-5 Example Designer page of Stateful ConvoBlocks interface's final design, highlighted according to design decisions. *Yellow:* The yellow box illustrate the tabs panel of the interface. It contains the set of "states" the developer defined for the conversation. *Pink:* The pink box highlights the "add state" button for the interface. *Purple:* The purple box highlights the Non-visible Components Panel for the selected tab. When a new component is dropped onto the selected tab screen, it will show up in this panel. *Blue:* The blue box highlights the Properties panel for the selected component. Intent utterances and slot types are defined here. 40

3-6 (a) 'Ordering' tab in the Viewer panel on the Designer page. (b) 'Shipping' tab in the Viewer panel on the Designer page. 41

3-7 (a) Viewer panel in the Designer page before intents are added for the 'Ordering' state. (b) Viewer panel after intents are added for the 'Ordering' state. 41

3-8 Pop-up window that prompts the developer for a new state name when the '+' button is clicked. 42

3-9 The blocks drawer for intent *StoreHours*. 43

3-10 (a) *when-intent-spoken* block for *StoreHours* intent, which is of state "global". (b) *when-intent-spoken* block for *AskColor* intent, which is of state "Ordering". (c) *when-intent-spoken* block for *Shipping* intent, which is of state "Shipping". 43

3-11 The "Voice" blocks drawer for all Alexa-related blocks. The purple box highlights the *set-current-state* block. 44

3-12 The endpoint event for the *NewOrder* intent. These blocks cause Alexa to respond with the sentences attached to the Ask block and change the current state of the conversation from 'global' to 'Ordering'. 44

4-1 User workflow to create a conversational AI agent. The user first implements the Voice User Interface (VUI) and endpoint function using a block-based interface. The blocks are converted to JSON and JavaScript, which define the agent's functionality on Alexa devices [30]. 47
4-2 Blocks editor that includes example code for Lorina's *Dress-ordering Service* agent, developed using this Stateful ConvoBlocks interface implementation. 52
4-3 (a) Alice's conversation with the *Dress-ordering Service* agent, developed in this Stateful ConvoBlocks implementation. The conversation is very similar to the one in Section 3.1. (b) Alice trying to trigger an intent in the "Shipping" state while the conversation is still in "Ordering" state. . . 53

6-1 *when-intent-spoken* block for "StoreHours" intent (a) before switching to a different project and back (b) after switching to a different project and back. 65
6-2 Mockup for the improved *Components* panel (left) and *Properties* panel (right) in the Designer editor. 66
6-3 Mockup for the improved blocks editor interface. 67

Chapter 1

Introduction

Conversational Artificial Intelligence (AI) is a rapidly growing field that has the potential to revolutionize the way we interact with machines. Conversational AI systems are designed to simulate human conversation and enable users to interact with machines using natural language. With advances in technology, algorithms, and sheer compute power, it is now practical to utilize AI techniques in everyday applications in the domains of transportation, healthcare, gaming, productivity, and media [10].

In recent years, conversational AI has gained significant attention in the field of education. Researchers and educators worldwide are adopting this new approach to teaching and learning that leverages conversational AI technology to enhance the education experience [28, 16]. Conversational AI Education systems can provide students with personalized, interactive, and engaging learning experiences [27, 31]. ConvoBlocks is one such conversational AI development platform dedicate to children's education. It builds on regular App Inventor and simplifies the programming process for building conversational agents that run on Amazon Alexa-enabled devices [31].

The objective of this thesis is to extend the original ConvoBlocks interface with state-machine based functionalities and investigate their effectiveness in improving the education experience. The thesis will examine the current state of conversational AI technology and its application in K-12 education. It will also discuss the design and implementation of the extended Stateful ConvoBlocks. Finally, it will present the design of a user study to investigate the benefits and challenges of using Stateful ConvoBlocks in education and evaluate its impact on student learning outcomes.

1.1 MIT App Inventor and the K-12 Conversational Agent Framework

MIT App Inventor is a cloud-based platform that allows people to create mobile applications without needing extensive programming experience. It provides a visual, drag-and-drop interface that allows users to create apps by simply selecting and arranging various puzzle-piece-like blocks.

As one of the most extensively used platforms in CS education, MIT App Inventor is designed to be accessible to users of all skill levels, from beginners with no programming experience to more advanced users looking to create complex apps [35].

The MIT App Inventor Conversational Agent Platform, better known as ConvoBlocks, builds on regular App Inventor and simplifies the programming process for conversational agents. More specifically, it lower the barrier of entry to AI programming. Conversational AI programming usually involves complex technical terminology like intents, slots, long short-term memory (LSTM) networks, etc. ConvoBlocks enables students to learn such complex concepts through an intuitive blocks-based development environment, easing their need to handle the nuances of text-based programming languages. This allows people with little or no previous programming experience to start building workable agents quickly, which enhances their self identity as a programmer and encourages an engaged learning environment [17].

MIT App Inventor and ConvoBlocks have empowered thousands to create software and conversational agents with real-world usefulness. Students have created agents to teach people sign language, improve mental health, and diagnose illnesses [32]. These platforms promote computational thinking skills and help learners see themselves as creators rather than only consumers in technology and AI. Educationally, it encourages learners to develop knowledge through creating and help broaden and diversify participation in the field of artificial intelligence.

To facilitate learning, in the workshop design of this thesis (see Chapter 5), I embedded computational action activities. These pedagogy interventions are designed to boost students' confidence in terms of feeling like programmers and being able to make their own agents that have a positive impact in their communities.

1.2 Challenges Facing ConvoBlocks Programmers

Conversation is one of the most primary and intuitive methods people use to communicate with each other. Human conversation requires the ability to understand the meaning of spoken language, relate that meaning to the context of the conversation, create a shared understanding and world view between the parties, model discourse and plan conversational moves, maintain semantic and logical coherence across turns, and generate natural speech [10].

While the ConvoBlocks interface provides unique education advantages in democratizing AI, it relies on Amazon's Alexa service to enable its users to create custom conversational agents. As a result, it shares most limitations and challenges of the Alexa platform. Despite how conversational AI is becoming ubiquitous in everyday applications, current agents, like Google Home, Apple's Siri and Amazon Alexa, are focused on short, task-oriented interactions [15]. Agents like Alexa may excel at simple interactions like playing music or answering short questions, but they often have a hard time producing long, free-form conversations that occur naturally in social interaction [4].

Aside from its difficulty in handling complex conversations, agents like Alexa also have problems with mis-recognized speech. Alexa sometimes mistake porches for Porsches and Pampers for cancer [22, 29], and researchers have found that industrial speech recognition systems by companies like Amazon, Google, etc. did substantially worse when the natural language input comes from a black speaker rather than a white speaker [11]. These flaws in Alexa's speech recognition system then presents additional challenges for ConvoBlocks programmers, as the agents they develop share the same natural language understanding system as Alexa.

1.3 Existing Mechanisms to Solve These Challenges

In reality, compared to human interactions, conversational AI interfaces can never be as natural, because their designs are inherently more constrained [6]. For any interaction to become natural, humans must first understand "cultural standards" of that interaction interface [20]. Since conversational interfaces are often not developed according to a pre-defined set of standards, end users often have trouble formulating the right mental model

of how their conversational AI works [6], and this challenge is pretty much universal to all current voice interfaces, including ConvoBlocks, Alexa, Google Dialogflow and Voiceflow.

Both Google Dialogflow and Voiceflow combat this challenge of naturalness with the use of a dynamic state-machine visual structure. As discussed in Section 2.3, this visual structure helps users build a mental image of the conversation structure, which improves the feeling of naturalness. Dialogflow also bounds its intents to separate pages of the conversation, which helps reduce the challenge of misinterpreting user input.

In ConvoBlocks, we can also address these challenges by mimicking the state-based aspects of these other platforms. Programmers can create a list of states with code blocks and manually filter intents through a series of conditionals. We can also build and connect a mobile app to our skill and generate the visual representation of conversation flow on the app. There are a few problem with this approach. For one, it is very time consuming and error-prone for beginner-level programmers, who represent the main audience for the ConvoBlocks interface. For another, it distracts programmers from their purpose of using the platform — to learn about building conversational agents without having to be proficient with coding language details.

1.4 Introducing States

In this thesis, I present a better solution to address the aforementioned challenges by introducing states to ConvoBlocks. My design and implementation of Stateful ConvoBlocks allows programmers on the platform to specify complicated multi-turn conversations and includes visual feedback that helps programmers form a mental image of their conversation design. State/intent associations, state handlers and state-based intent screening all happen in the back end. Programmers can build their agent with simple, intuitive interactions with Designer view components and Blocks Editor blocks. All design decisions of the Stateful ConvoBlocks interface are made with a learner-centered mindset. For my design iterations and final design product, see Chapter 3.

1.5 Contributions

The main contributions of this thesis fall under the following categories.

1. I compared four existing conversational AI programming interfaces in terms of their design and learnability aspects:
    - ConvoBlocks
    - Alexa Developer Console
    - Google Dialogflow CX
    - Voiceflow
2. I introduced state-based features into ConvoBlocks that increase its expressiveness in specifying conversations:
    - I extended ConvoBlocks with the notion of "state" that specifies a context in which ConvoBlocks intents are active.
    - I modified the ConvoBlocks design editor to allow creating states and associating intents and entities with a state.
    - I added new blocks to the ConvoBlocks blocks editor that express the dynamic control of state changes in dialogues.
3. I designed a user study to investigate the impact of state-based intents:
    - Answer questions about usability of the new design
    - Investigate whether the introduction of states help programmers specify conversational context
    - Investigate how the introduction of states influence programmers' perception of conversational agent flexibility and trust, especially for students with varying levels of experience interacting with conversational agents.

Chapter 2

Background and Related Work

This chapter describes related work, including a general overview of the state-of-the-art for conversational AI, K-12 AI education initiatives, platforms dedicated to developing conversational agents, and understanding conversational design.

2.1 Conversational AI

Conversational AI is a sub-domain of Artificial Intelligence that deals with speech-based or text-based AI agents that have the capability to simulate and automate verbal interactions. Advancements in methods (like machine learning and deep learning) required to develop highly accurate AI models combined with the rise in practical implementation and demand of these agents in healthcare, customer care, e-commerce and education has made Conversational AI Agents like chatbots and voice assistants ubiquitous in everyday applications [12].

Most voice-based conversational agents follow a similar architecture [1]. First, the agent comprehends speech signals and converts them to text by a process called automatic speech recognition (ASR). After obtaining the text, the agent tries to understand the meaning and intention of the user using existing knowledge by a process called natural language understanding (NLU). Once the concepts and intents are identified, the agent starts the process of response generation (RG), which involves identifying relevant responses based on context, knowledge, personalization, and some form of planning. Planning may involve optimizing for some reward such as sentiment, serving the goal in a goal-directed dialogue, or increasing user engagement. This process can be managed

by a dialogue manager (DM), which acts as an engine for maintaining the state and flow of the conversation. Finally, once the output response is produced in textual form, the agent converts it to speech by a process called text to speech (TTS) [10]. These NLP processes flow into a constant feedback loop with machine learning processes to continuously improve the AI algorithms, as seen in Figure 2-1. These principle components allow it to process, understand, and generate response in a natural way.

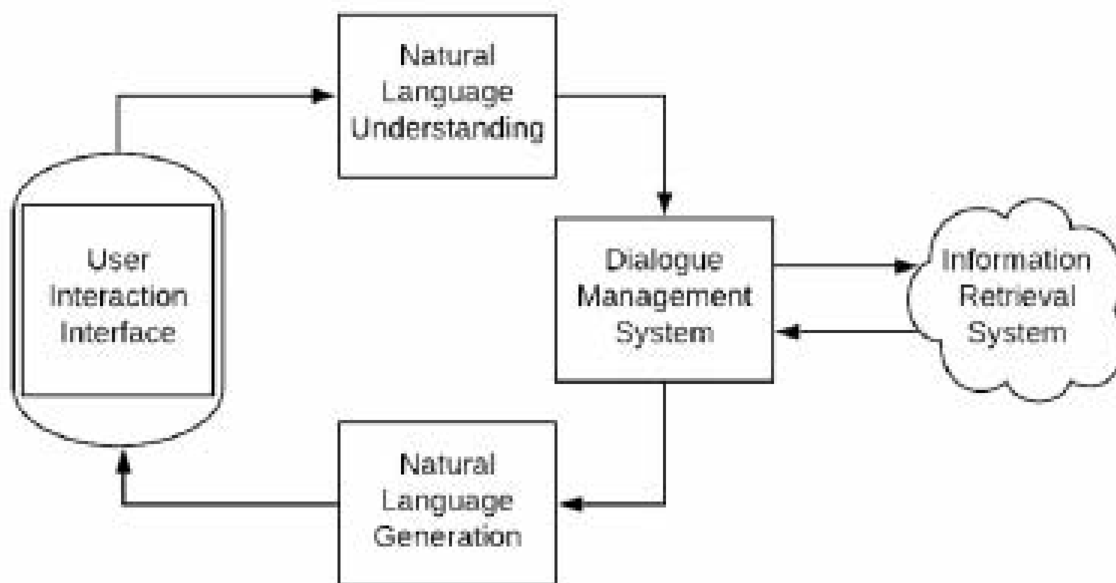

Figure 2-1: Conversational AI architecture [12].

Conversational agents are usually separated into two paradigms: goal (or task)-oriented and open-domain. Task-oriented conversational AI is designed to complete specific tasks or help users achieve a particular goal. These tasks could include booking a hotel room, scheduling an appointment, or ordering food. Task-oriented conversational AI uses natural language processing (NLP) to understand the user's intent and context, then follows a set of pre-defined rules or workflows to complete the task. These chatbots are usually designed to be highly focused, with a limited range of conversation topics and responses. They are ideal for use cases where the user has a clear goal in mind and wants to complete it quickly and efficiently.

On the other hand, open-domain conversational AI is designed to mimic human conversation and engage in more free-form discussions [2]. These chatbots are not focused

on completing specific tasks or achieving particular goals but are designed to engage in more open-ended conversations. Open-domain chatbots can generate responses based on a wide range of inputs, including user queries, current events, and trending topics. They use more advanced NLP techniques, such as deep learning algorithms, to generate more human-like responses.

In terms of applications, task-oriented conversational AI is commonly used in industries such as customer service, hospitality, and retail, where there are clear workflows and processes to follow. Open-domain conversational AI, on the other hand, is more suited to applications such as entertainment, education, and personal assistance, where there is less structure and users may have a wider range of conversation topics.

Both task-oriented and open-domain conversational AI have their own strengths and weaknesses. Task-oriented chatbots are generally more efficient and focused, while open-domain chatbots are more engaging and flexible.

2.2 Educational Initiatives for Conversational AI

The exponential rise in the use of conversational AI [24] has led researchers to investigate societal implications of agents' ubiquity. As much as people can create conversational agents to make positive change in their communities, unfortunately, agents can also be used to spread misinformation and invade privacy. Since conversational agents can be personified [23] and people can build relationships with them, just like with other humans [23, 25], it is important for people to be able to calibrate their levels of trust towards conversational agents (CAs), according to their actual level of trustworthiness. In the past, our team has investigated the link between a conversational AI educational intervention and participants' perceptions and levels of trust of CAs, finding that participants most often mentioned learning something about CAs as reasons for changes in their trust [7].

Along similar lines, researchers have noted the importance of teaching a breadth of types of AI, a need for CA pedagogical artifacts, and the unique societal questions and challenges CAs present due to their relational nature [14, 19, 23]. In order to ensure a more informed populace that understands the technologies they interact with every day, as well as to inspire the next generation of AI researchers and software developers, many researchers are developing education initiatives to empower more people, especially

those with little or no prior programming background, to understand AI and its societal implications [28]. For instance, the AI4K12 initiative is developing tools and curriculum based on five core “Big AI Ideas”, and MIT’s Responsible AI for Social Empowerment and Education (RAISE) initiative is developing vocational-technical and K-12 tools for AI education [28, 16].

Over the past few years this community of researchers, educators and entrepreneurs is rapidly expanding [32]. They develop tools and curriculum worldwide, often emphasizing approaches that encourage students to take “computational action” [21]. Researchers claim that empowering students to develop meaningful projects that can help others in their community encourages diversity and equity, as well as class engagement [27, 31, 28]. These pedagogy interventions are shown to increase students’ self-efficacy and identity as programmers. Thus, in Chapter 5, I embedded computational action activities in my curriculum design in this thesis.

2.3 Programming Conversational AI Platforms

In this section, I analyze a number of conversational AI development platforms, with respect to the range of programming abilities they address, their method of structuring the design of conversations, and the conversational agent concepts they can teach. These platforms, from least required experience to most, are ConvoBlocks [30, 32, 33], Voiceflow [34], the Alexa Developer Console [4] and Google Dialogflow CX [9].

2.3.1 Current ConvoBlocks Interface

There are few dedicated tools to teach people conversational AI, and even fewer to teach young people [8]. One such tool is ConvoBlocks (a.k.a. the MIT App Inventor Conversational Agent Platform). It is a web-based block-programming interface that enables learners to develop agents that can converse with the user and respond to utterances using natural language, share data with mobile phone applications developed in MIT App Inventor, and generate unique responses using machine learning [30]. For example, a learner could create an app that allows users to ask Alexa to tell them a story, give them cooking directions, calculate their annual carbon footprint etc.

The ConvoBlocks interface in App Inventor is easy to use and does not require ex-

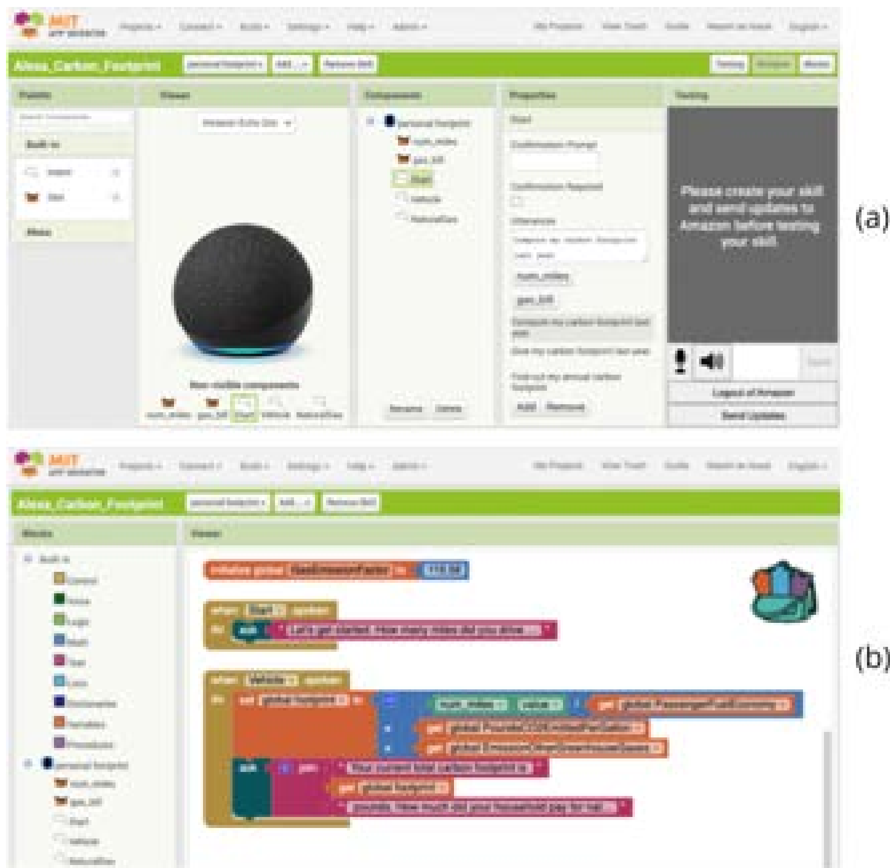

Figure 2-2: The current ConvoBlocks interface. (a) The Designer view. (b) The Blocks editor.

tensive knowledge of programming. Developers can simply drag and drop intent and slot components onto their project's designer interface, define their utterances and customize the responses using puzzle-piece-like blocks.

The main goals of this interface are to empower students with little or no programming experience to **(1) learn computational thinking skills**, **(2) learn conversational AI concepts**, and **(3) develop conversational AI applications** [30]. Figure 2-3 illustrates examples of how the interface implements these three goals, including how different code blocks can teach conversational AI concepts.

The pink boxes highlight the interface's attention to computational thinking skills, including events (*when*), conditionals (*if*), and data (*get slot*). The blue boxes highlight conversational AI concepts, including *invocation name*, *intent* and *slot* (i.e., entity), as well as a `generate text` block containing long short-term memory (LSTM) neural networks. The yellow boxes illustrate the learnability of the interface. The leftmost highlight shows the "drawers" where blocks can be easily dragged-and-dropped into the interface, and the smaller yellow box shows how block-based coding can prevent syntax errors, as

Figure 2-3: The ConvoBlocks interface highlighted according to design goals [30]. The blocks here creates a storybook agent.

only certain blocks (e.g., `say` and `text` blocks) can be connected to each other [30].

Unlike the rest of the platforms discussed later in this section, the ConvoBlocks interface is specifically developed to teach conversational AI concepts and empower young students to develop customized, workable conversational agents with middle and high school students as its target audience. It provides students with a less-abstracted, yet still accessible, method to program agents. Using block-based coding structure, ConvoBlocks enables low-code skill-building and teaches students the low-level programming concepts of functions, conditionals, and events, while still abstracting away syntax errors through puzzle-block-like code [31].

2.3.2 Alexa Developer Console

The Alexa Developer Console allows developers to create agents very similar to those developed with ConvoBlocks. They are event-driven, can be trained to recognize intents and entities, and run on Alexa-enabled devices [5]. It contains a GUI for training intent- and entity-recognition, an IDE etc. Its functionality is similar to that of ConvoBlocks, yet its tutorials are much more advanced.

Skills are like apps for Alexa. They let end-users use their voices to perform everyday tasks like checking the news, listening to music, playing a game, and more. Both organizations and individuals can publish skills in the Alexa Skills Store to reach customers on

any Alexa-enabled device [3].

In the Alexa Developer Console (and any platform taht interfaces with Alexa, like ConvoBlocks), programmers build their own Alexa skills with intents and slots. An intent represents a specific action or request that a user wants to perform when interacting with an Alexa skill. It defines the user's intention or the purpose behind the spoken command. Slots, on the other hand, are placeholders or variables within an intent that capture specific pieces of information from a user's spoken command. Slots allow the agent to gather and extract relevant data from the end-user's input, such as names, dates, numbers, or any other specific information required to fulfill the end-user's request. When an end-user interacts with a skill and provides input that matches an intent, Alexa's natural language understanding system processes the user's speech and extracts the slot values. These slot values are then passed to the skill's back end along with the intent. In the back end code, the programmer can access the slot values and use them to customize the behavior and response of the skill [4].

To create a custom skill in the Alexa Develoepr Console, a programmer must first design the custom voice interaction model of the skill through JSON. The interaction model defines the spoken part of a skill, including the statements, phrases, and questions spoken to and by the skill. Afterwards, the programmer can build the "brain" of the skill — the endpoint function. This is a service that can receive JSON requests and return JSON responses. Generally, in the context of Alexa, this service is an Amazon Lambda Function to be hosted on Amazon Web Services, but it can also be a custom HTTP web service. This function contains event handlers that decides what kind of JSON response to send back to Alexa when a particular intent is triggered. The programmer can write it in JavaScript, Node.js, and Python. Finally, the programmer can test and run the new skill using a number of pre-built or custom Alexa simulator or Alexa-enabled device [3].

2.3.3 Google Dialogflow CX

Google Dialogflow CX enables developers to create agents for Google Assistant devices [9]. Dialogflow CX provides a unique method of agent development using state machines, and allows developers to access advanced features such as "flows" and "pages" [9].

The Dialogflow CX platform allows developers to develop agents similar to those using ConvoBlocks and the Alexa Developer Console. Developers can create agents that classify

custom intents, extract entities and generate responses, just like they could in the other platforms. Dialogflow CX also allows developers to integrate their virtual assistant with other services, such as databases, APIs, and webhooks, to provide more sophisticated functionality. This can include integrating with third-party services like Google Maps or Stripe for payment processing.

One of the key advantages of Dialogflow CX is its ability to handle complex conversation flows and multi-step processes by describing and visualizing these conversations as state machines. The platform allows developers to design conversation flows that can branch off into different paths based on the user's input, allowing for more natural and flexible interactions. Complex dialogues often involve multiple conversation topics, where each topic requires multiple conversational turns for an agent to acquire the relevant information from the end-user. Dialogflow CX builds this state-based structure is using components called "flows" and "pages". Flows are used to define these conversation topics and the associated conversational paths end users might take, and pages define specific states in the conversation [9].

An important difference between agents developed using ConvoBlocks or the Alexa Developer Console and agents developed through Dialogflow CX is their component scope. In Alexa (and similarly ConvoBlocks), all intents and entities are top-level. At any point in the conversation, all intents are active and listening. That means end users can invoke any intent or access any entity at any time. With Dialogflow CX agents, however, intents and entities are associated with specific pages and flows. End users can only access the components when the relevant flow and page is active. Figure 3-1 shows an example state-machine conversation construction using flows and pages.

In terms of pedagogy, Dialogflow CX has setup instructions and tutorials available, but they are intended for more experienced programmers. As a commercial conversational agent-development platform, Dialogflow CX's cloud-based nature requires developers to be familiar with the notion of cloud computing beforehand. Its linked tutorials also begins with a list of definitions including flows, pages, state handlers, entities, and intents etc. To be able to follow these tutorials, developers also have to be familiar with concepts like training and deployment, as well as have sufficient background in graph theory, to understand properties of state-machines and directed graphs.

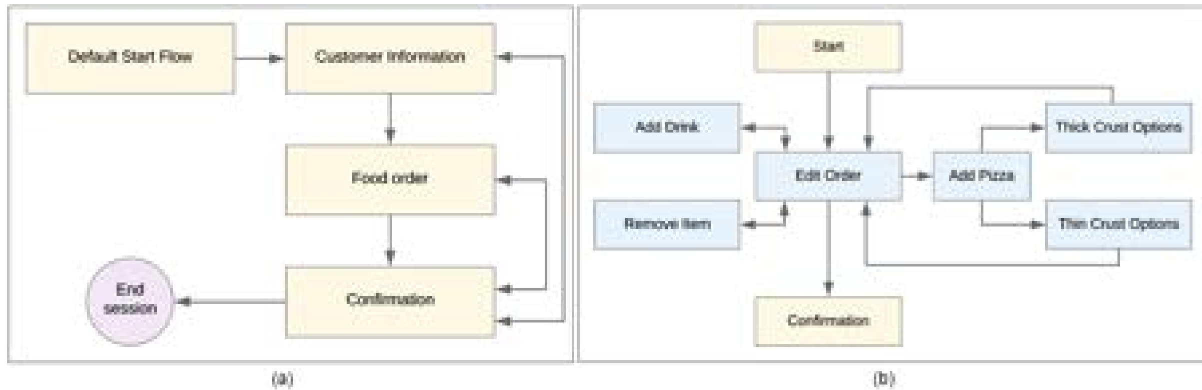

Figure 2-4: (a) An example representation of four flows, the "Default Start Flow", a "Customer Information" flow, a "Food Order" flow, and a "Confirmation" flow. (b) Within the "Food Order" flow, a user might develop pages (e.g., "Edit Order" page, "Add Pizza" page, "Add Drink" page, etc.). [26].

2.3.4 Voiceflow

Voiceflow is a no-code conversational AI platform that enables users to design and build chatbots without any programming knowledge. What's unique about the Voiceflow interface is its design process. It starts by designing the converstaion flow, and then proceeds to add conversational elements like intent and prompts.

When programmers starts a new project in Voiceflow, they always start by adding a "Start" block, which represents the beginning of the conversation. Then, they may add other blocks to define the different steps or interactions in the conversation, which includes prompts, user responses, conditionals, API calls, etc. When the design of the flows is completed, they can then customize the settings of each block to define its behavior. For example, they may add a "Prompt" block to ask the end-user a question, and then add a "Choice" block with conditions to handle different user responses based on their selection. Testing and debugging is handled on the same screen.

Another unique point is that Voiceflow enables users to integrate with third-party services such as APIs, databases, and other services to add functionality to the conversational experience. This allows users to perform actions such as booking a flight or making a purchase within the conversation. When designing their own agents, Voiceflow also allow programmers to select different Natural Language Understanding (NLU) platforms. The resulting agent can then leverage the capabilities of the selected platform, with a range of options including Voiceflow itself, Dialogflow, IBM Watson, Microsoft LUIS, Rasa, etc.[34]

Chapter 3

Design of State-based Intents

This chapter describes the design process of my Stateful ConvoBlocks interface. To illustrate the system, I present two motivational conversation scenarios that drove my design: *Dress-ordering Service* and *Little Red Ridding Hood*. A version of both agents are implemented using the Stateful ConvoBlocks interface and will be provided to students in the user study. I then discuss the design decisions I went through and present my final design, using the *Dress-ordering Service* agent to demonstrate the workflow.

3.1 Design Motivation

In this section, I provide two example conversations that motivates my design of the Stateful ConvoBlocks interface, and describes how each example is expressed in other conversational AI programming platforms.

3.1.1 Dress-ordering Service

This *Dress-ordering Service* agent represents a category of conversation patterns called Open Request. This type of conversation handles sets of related requests, or topics, that require entity extraction and agent elicitation (a.k.a. "slot filling"). Multiple entities are captured and preserved as context across the set of unique requests (i.e., intents). It is compatible with Inquiry patterns and user- or agent-initiated Repair patterns [18].

To illustrate the final agent developers create in the Stateful ConvoBlocks, I provide an example conversation below, in which a fictional end user—Alice from *Alice in Wonderland*—orders a dress.

```
[After arriving at the March Hare's house, Alice needs to order a blue dress for her
    tea party with Mad Hatter. Her old dress is torn when she followed the White
    Rabbit down the rabbit hole, and she couldn't attend a crazy tea party without a
    trusty blue dress!]
Alice: Hi there
Agent: Hi! I'm an agent that orders dresses. How can I help?
Alice: Where is your store located?
Agent: You can find us at 123 Duchess Street, Wonderland.
[Alice realizes this is quite a trek, and at 8 PM, she's not sure she'd make it in
    time for closing.]
Alice: What are your hours?
Agent: We're open until 9 PM on weekdays.
Alice: Can I order a dress online?
Agent: Ok, let's start a new order. I'd like to collect a bit more information about
    the order. What size would you like?
Alice: Small
Agent: I'd like to collect a bit more information about the order. What color would
    you like?
Alice: Blue
Agent: Ok, so you'd like a small blue dress. What's your shipping address?
Alice: March Hare's House, Wonderland.
Agent: Ok, your order for a small blue dress will arrive in two days. Thanks for your
    order!
[Alice sighs with relief. The tea party is three days away. She would make it!]
```

This conversation consists of three paths: store location, store hours, and making an order. When Alice go down the path of making a new order, she steps through the 'ordering' state and 'shipping' state. In Dialogflow CX, this conversation can be built as shown in Figure 3-1.

However, since the original Convoblocks interface is Alexa-enabled, the behavior of 'shipping' state following after 'ordering' state is more difficult to realize. As discussed in Section 2.3, all intents and slots are top-level. So intents in 'shipping' state (like asking for shipping address) are also accessible even before Alice created her order of a small blue dress.

To simulate such behavior, developers in ConvoBlocks could create a custom states dictionary in App Inventor, mentally assign intents to each state and add condition statements in every Intent event handler. However, such a solution requires the developer

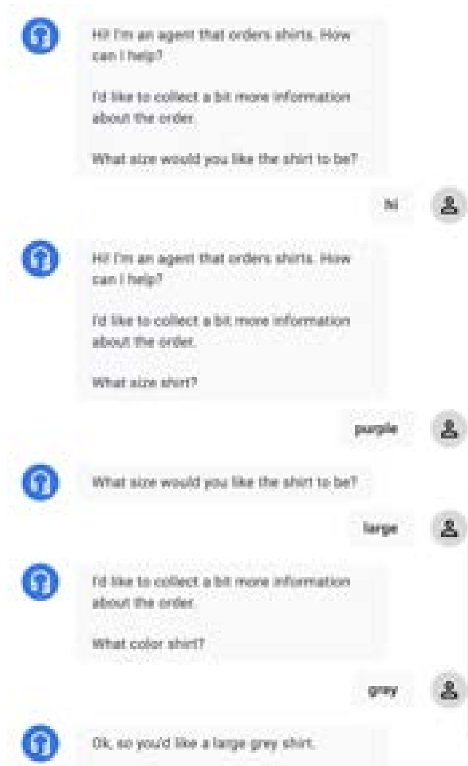

Figure 3-1: An example conversation in which an end user tests the clothing store agent that developers create in the Dialogflow tutorial [26]. In the first turn, the agent tries to collect information about the size of a shirt, but the end user invokes the greeting intent. The agent then repeats its greeting and question about shirt-size, but the end user responds with the color of the shirt. The agent repeats its question again, and the end user provides the size. Finally, the agent asks about the color of the shirt and the end user responds with a color, and the agent confirms the end user's order. When the end user initially responds with a color to the agent's question about size, the agent does not recognize the color since the color intent was out of scope at that point in the conversation. This is due to flow and page modularization.

to memorize the structure of the conversation and is both error-prone and inefficient in teamwork cases. It also takes away from the main pedagogical purpose of having learners build these complex conversations — to practice constructing multi-turn conversations.

3.1.2 Little Red Riding Hood

This *Little Red Riding Hood* agent represents a category of conversation patterns called Extended Telling. Two common cases of this type of conversation are storytelling and instruction giving, both typically require more than one turn to complete [18]. The teller gives the first part of the story and then waits for the recipient's approval to continue.

To give a brief illustration of on example extended telling conversation, I provide an example where Alice asks the agent to tell her a story. The agent tells her the story about

little red riding hood.

```
Alice: Can you tell me a story?
Agent: Once upon a time there was a dear little girl called Little Red Riding Hood.
    She lives at home with her mom.
Alice: What's next?
Agent: One day her mother told her to bring some wine and cake to her grandma, who
    lived out in the woods.
Alice: Okay?
Agent: In the woods Little Red Riding Hood met the Big Bad Wolf, who then lied his way
     into grandma's house and devoured Grandma. When Little Red Riding Hood arrived at
     the house, she did not realize it was Big Bad Wolf pretending to be grandma, so
    she was devoured too.
Alice: Then what happened?
Agent: A huntsman passed by and saved Little Red Riding Hood and her grandma.
Alice: Wait what happened?
Agent: The huntsman passing by heard loud snoring, and found Big Bad Wolf. He cut the
    wolf's stomache and got Little Red Riding Hood and her grandma out.
```

Implementation of this conversation also requires the agent to be able to carry some contextual information across user-agent interactions. For instance, one implementation would involve setting conversational states and moving the current state as the dialogue progresses.

3.2 Design Round 1

This section describes my initial design for Stateful ConvoBlocks. It involves design decisions regarding how to handle different types of conversations as well as how to display conversations to the developer.

3.2.1 Separating Slot-filling Dialogues

In this design, I separated conversations with agents into two categories: slot-filling dialogues and extended telling dialogues.

Slot filling dialogues represent pieces of conversations that uses a series of questions and answers to gather information from end users in order to complete a specific task or process. The term "slot" refers to a specific piece of information that needs to be

collected, such as a preferred color, address, name, or date of birth. In a slot filling dialog, the system prompts the user for information by asking a series of questions. The user responds to each question with the requested information. The system then uses this information to fill in the appropriate slots and move on to the next question. In real life, these dialogues are commonly seen in business-related applications, such as customer service, appointment booking, and reservation systems. In Stateful ConvoBlocks, these conversations can also be used to create chatbots that help end users complete tasks more efficiently and effectively.

Aside from slot-filling dialogues, there are also other kinds of multi-turn conversations, like extended telling dialogues. These conversations allow contextual information to be carried within conversation flow. Examples include story-telling and giving instructions. Though these conversation does not require the agent to extract information from the end user, the agent still needs to remember which part in the conversation flow the agent-user pair is currently at, and when they can move on to the next part.

In the initial design, I used Alexa Dialog Interface API to implement the slot-filling dialogue types. The Alexa Dialog Interface API contains set of tools and resources that allows developers to build natural, conversational interactions between users and the agent. The API provides a framework for building multi-turn dialogues with slot elicitation, slot confirmation, intent confirmation and slot validation, allowing developers to create complex, branching conversations that respond dynamically to user input. It also supports the use of context and session attributes, which allow developers to store and retrieve information across multiple turns of the dialog. I also created custom state-switching blocks in the Blocks editor to enable other kinds of topic switching. However, my implementation for this design did not go through. After communicating with Amazon's Alexa development team, I found out that there are internal issues with this Dialog Interface API and it is not likely to be fixed in the next few month. Additionally, having learners separate their agents' conversational design into slot-filling dialogues and extended telling dialogues and handle each in a different manner adds unnecessary complexity to their development process. So I decided to combine the two types of conversations and handle all turn-taking conversations in the same way.

3.2.2 State Machine Representation of Conversation

Another design decision in the initial design is to replace the Echo Dot image in the Viewer panel of the Designer page in the current ConvoBlocks interface with a generated state machine illustration of the agent's conversation structure, as seen in Figure 3-2. Like the Dialogflow CX interface, this state-machine illustration allows learners to have a clear visual representation of the conversation, which helps them design the structure of their agent. However, unlike the Dialogflow CX interface, the state machine is not dynamically created by the user, but generated from the intents developers drag in and their associated paths and states developers define in the component Properties panel. To preserve the design of MIT App Inventor, developers code the logic for each intent in the Blocks editor.

Figure 3-2: Mockup for the designer interface in initial design. The Echo Dot image and non-visible components panel is replaced by a state-machine graph to illustrate the conversational flow.

Despite the advantages of a visual representation of the conversation's state machine, this design had a few important flaws. This design requires learners to have some background in graph theory to understand the reference to a state machine. Given that the main target audience of the interface is middle or high school students, they are unlikely to have much background in directed graphs or state machines. They also need to understand flow/page modularization, where pages (each rectangular box) represent specific situations in a conversation, and flows are chunks of pathways in the conversa-

tion that connect these situations. In past workshops, we have found that the concept of modularization has been cited among the most difficult concepts for participants, and in conversational AI curricula it is usually not introduced until the second or third tutorial [7]. This graphic state machine design requires learners to be familiar with agent modularization from the start, which is not the best for the pedagogical purpose of the interface. Therefore, after some experimentation, I decided to remodel the Designer View, as discussed in Section 3.4.2.

3.3 Design Round 2

This section describes the second design for Stateful ConvoBlocks. Specifically, it involves design decisions regarding the organization of the blocks editor.

3.3.1 Representing States by Different Blocks Editor Windows

This design categorizes intent components by their *state* property and displays them on separate pages. It includes adding a new `Add State` button in the design tool bar next to the existing `Remove Skill` button, a new drop-down list of all states next to the `Add State` button. When the developer selects a specific state from the drop-down list, it gives the developer the Design Editor and Blocks Editor corresponding to the selected state.

This design is inspired by the `Screens` feature for mobile app development in MIT App Inventor. The `Screens` feature also contains a `Add...` button and a dropdown list that shows all the screens the developer have created in their app. The developer can click on a screen name in the list to switch to that screen. When they switch screens, the Designer view and Blocks editor will update to show the components and blocks for that screen. The developer can also switch screens using the "Open another screen" block in their blocks code.

Since most ConvoBlocks learners are already familiar with the regular MIT App Inventor interface [32], in order to strengthen the tie between the ConvoBlocks interface and App Inventor, the `States` feature in this design is intended to do something similar, and allow developers to group intents and slot components into different states in a conversation just like they could group buttons, canvases and other features into different

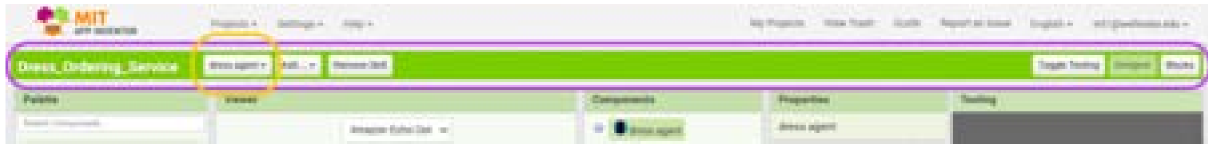

Figure 3-3: Current display of the top half of the ConvoBlocks interface. *Purple:* The purple box highlights a panel called “design tool bar”. *Yellow:* The yellow box highlights the dropdown list containing all screens and Alexa skills the developer created for this project.

screen in a mobile app. However, after experimenting with this design, I found that this approach also has a few problems. The design tool bar is shared across the whole project, including any Alexa skill or mobile app the developer added to the project. Since the notion of states is limited to each Alexa skill specifically, it does not make sense to allow developers to access Alexa skill states when they are developing mobile apps. Hence, I decided not to change the design tool bar and put state-related functionalities in the viewer panel of Alexa skills instead. Design details are discussed in Section 3.4.2.

3.4 Final Design

This section describes the final design of the Stateful ConvoBlocks interface. In this section, I present a fictional character named Alice to illustrate the interface’s capabilities. Alice is interacting with a conversational agent developed by her sister, Lorina.

3.4.1 State/intent Association

As discussed in Section 3.1, having intents associated with states can help developers specify complex multi-turn conversations and organize their flow. Stateful ConvoBlocks supports a one-to-many relationship between states and intent components. Use Lorina’s *Dress-ordering Service* agent as an example. As shown in Figure 3-4, each state can have any number of intents, but each intent can only correspond to one state.

When Lorina first created the project, there is only one unique default state, *global*. She then add two new states, *Ordering* and *Shipping*. She put four intents in the *global* state, two in the *Ordering* state and one in *Shipping* state, and define when the current state of the conversation changes in the Blocks editor. In Stateful ConvoBlocks, for each intent, if it corresponds to the *global* state, then it is active all the time, meaning the

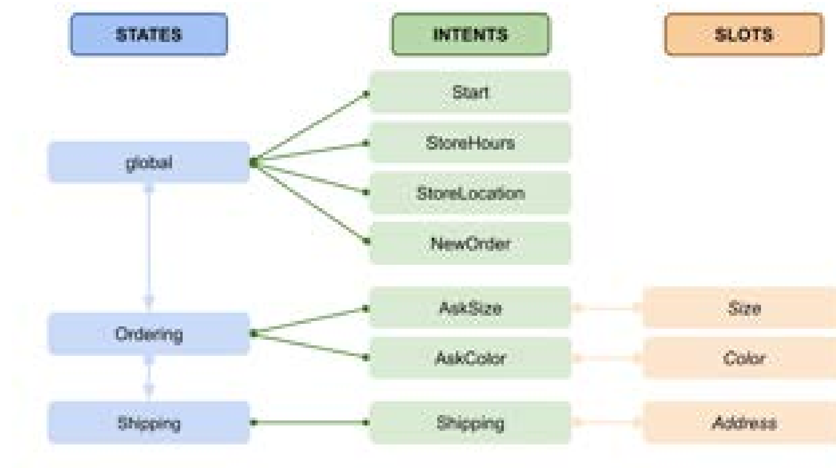

Figure 3-4: The association between states and intents in Lorina's *Dress-ordering Service* agent.

end user can invoke it at any point in the conversation. If an intent corresponds to a custom made state, say *Shipping*, then that intent is only active when the conversation's progress is at state *Shipping*. For example, when Alice is using the *Dress-ordering Service* agent, she can ask for the store location at any time, even when she is in the middle of making an order, because the `StoreLocation` intent is of *global* state. However, when she started making a new order and before she gave both dress color and size to the agent, she cannot invoke the `Shipping` intent. At that point she is still in the *Ordering* stage of the conversation. Intents she can invoke include `Start`, `StoreHours`, `StoreLocation`, `New Order`, `AskColor` and `AskSize.` If she tries to trigger the `Shipping` intent, the agent will give her the response `this intent cannot be invoked in the current state.` To be able to invoke the `Shipping` intent, she needs to finish providing all required details of her order so the conversation can move on to the *Shipping* stage.

Adding this state/intent association can help reduce misclassification of intents, since there are fewer intents for the natural language processing module to choose from at any given time. It also helps learners in their development process, as it provides them with additional tools to manage their components and scope their conversation. Implementation details for this state/intent association can be found in Chapter 4.

3.4.2 Designer View: Tabs

To illustrate this state/intent association for developers on the interface, I redesigned the Designer editor in ConvoBlocks. I introduced the use of tabs in the Viewer panel to

represent the states. Figure 3-5 showcases the Designer view of Lorina's *Dress-ordering Service* agent implemented with my final design.

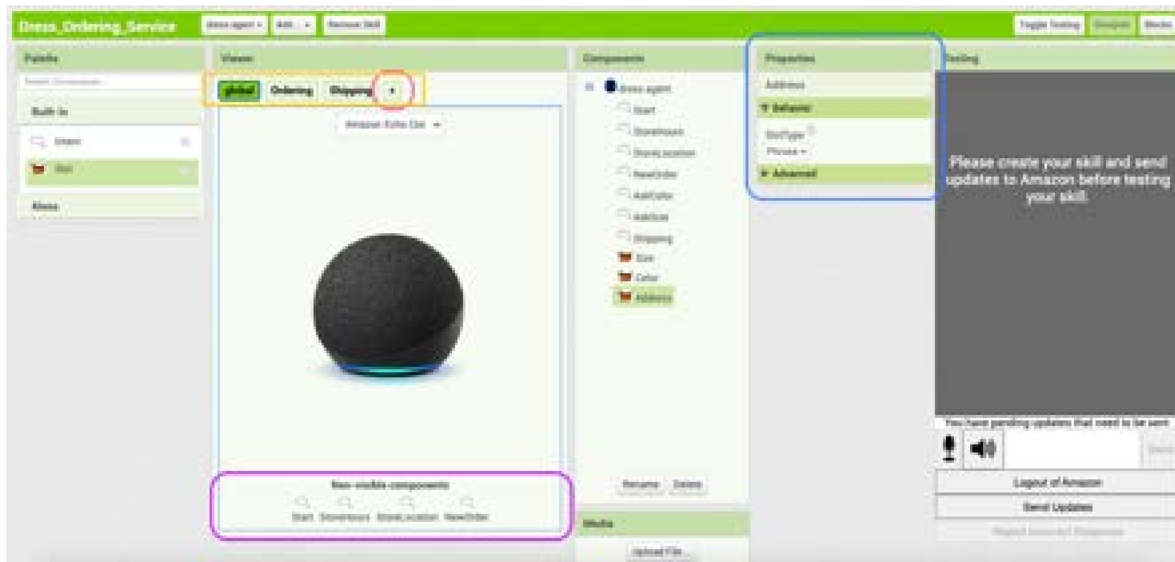

Figure 3-5: Example Designer page of Stateful ConvoBlocks interface's final design, highlighted according to design decisions. *Yellow:* The yellow box illustrate the tabs panel of the interface. It contains the set of "states" the developer defined for the conversation. *Pink:* The pink box highlights the "add state" button for the interface. *Purple:* The purple box highlights the Non-visible Components Panel for the selected tab. When a new component is dropped onto the selected tab screen, it will show up in this panel. *Blue:* The blue box highlights the Properties panel for the selected component. Intent utterances and slot types are defined here.

In the tab panel (highlighted in yellow), each tab represents a state. For each state, its intent and slot components show up on the tab corresponding to that state under "Non-visible Components" (highlighted in purple). In Lorina's *Dress-ordering Service* agent, there are seven intents and three slots in total. As shown in Figure 3-6 (a), when Lorina selects the `Ordering` tab, the tab widget turns green and shows the intents of state *Ordering*, including `AskColor`, `AskSize` as well as the slots these intents use. When Lorina switch to the `Shipping` tab, she can instead see the intents of state *Shipping*, which only includes `Shipping` (plus the slot this intent uses).

To add a new state to the skill, developers can use the + button at the end of the tab panel. Figure 3-7 illustrates part of Lorina's implementation process. When Lorina first created the project with an empty skill, the only tab in the Viewer panel is the *global* tab. She can drag and drop intents onto the *global* tab. `StoreHours` and `StoreLocations` are both generic information-retrieval inquiries, and `NewOrder` intent is responsible for tell the agent to begin (or start over) creating a new blank order. Hence, when Alice is

Figure 3-6: (a) 'Ordering' tab in the Viewer panel on the Designer page. (b) 'Shipping' tab in the Viewer panel on the Designer page.

talking to Lorina's agent, all these intents should be able to be invoked at any point, so Lorina put these intents under state *global*.

Figure 3-7: (a) Viewer panel in the Designer page before intents are added for the 'Ordering' state. (b) Viewer panel after intents are added for the 'Ordering' state.

If Lorina just wanted to create a simple Alexa skill, she could just use the *global* state without adding any new state or tabs. Since intents in the *global* state can be invoked at any time, just like intents in the original ConvoBlocks interface, beginners

who are yet to learn about multi-turn conversation can still program using the interface without having to deal with state-related functionalities. Therefore, even though my design increases the complexity of designing conversational agents, as developers must consider how to best scope the conversation, beginner-level learners can still choose to simplify their development process.

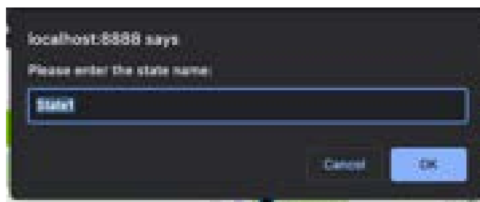

Figure 3-8: Pop-up window that prompts the developer for a new state name when the '+' button is clicked.

As seen in Figure 3-7, Lorina can add a new state to the conversation structure by clicking the `+` button highlighted in purple. This brings a pop-up window that prompts her for the name of the new state (see Figure 3-8). The default name for adding the ith state is `State\textit{i}`. Lorina can name the new state `Ordering`. When she clicks `OK`, an *Ordering* state is created and a corresponding tab is added to the tab panel and automatically selected (Figure 3-7 (b)). In the new tab, the Non-visible Components panel is now blank. To add new intents to the *Ordering* state, Lorina can simply drag and drop an intent component from the Built-in Components Palette and drop it anywhere on the *Ordering* tab. The new Intent component will automatically be assigned state *Ordering* and appear in the Non-visible Components panel under that tab.

3.4.3 Blocks Editor: Contexualized Block Name

To illustrate the state/intent association for developers on the Blocks editor page, I also re-designed the `when-intent-spoken` blocks. This can be found in the blocks drawer for each intent component (see Figure 3-9).

The `when-intent-spoken` block defines what occurs when a specific intent is spoken to Alexa. It can contain a set of voice, control, logic, or math etc. blocks that eventually define how the agent should process the end-user's input and respond when the intent is triggered. In my final design, I appended a `state` property to the block. For instance, in Lorina's *Dress-ordering Service* agent, the endpoint block for intent `StoreHours` now displays the state property of the intent by saying "when *StoreHours*.spoken in state

Figure 3-9: The blocks drawer for intent *StoreHours*.

Figure 3-10: (a) *when-intent-spoken* block for *StoreHours* intent, which is of state "global". (b) *when-intent-spoken* block for *AskColor* intent, which is of state "Ordering". (c) *when-intent-spoken* block for *Shipping* intent, which is of state "Shipping".

global". Figure 3-10 shows how different intent endpoint blocks show corresponding state properties for each intent. This way, the interface still provides the developer with sufficient state-related information in the Blocks editor and gives them a clear idea of the state/intent association while allowing the developer to put all coding blocks in the same page.

In my final design for Stateful ConvoBlocks, developers can program state-switching

logic in the Blocks editor. Figure 3-11 shows the new `set-current-state` block in the *Voice* drawer, which takes in a state name string as input and changes the current state of the conversation to the input state.

Figure 3-11: The "Voice" blocks drawer for all Alexa-related blocks. The purple box highlights the *set-current-state* block.

Figure 3-12: The endpoint event for the *NewOrder* intent. These blocks cause Alexa to respond with the sentences attached to the Ask block and change the current state of the conversation from 'global' to 'Ordering'.

Figure 3-12 illustrates the use of this new block in Lorina's *Dress-ordering Service* agent's code. These blocks define how the agent responds when the `NewOrder` intent is triggered — it starts a new blank order and sets the current state to *Ordering* state behind the scene. For instance, here's a piece of Alice's conversation with the agent Lorina developed:

```
...[current state: 'global']
Alice: Can I order a dress online? [trigger NewOrder intent]
Agent: Ok, let's start a new order. I'd like to collect a bit more information about
    the order. What size would you like? [give specified response]
...[current state: 'Ordering']
```

This dialogue snippet started when in the default *global* state. Since the `NewOrder` intent has state property *global*, it can be successfully triggered by Alice's input. The agent responds with the response defined in the `Ask` block, and the current state of the conversation is set to *Ordering*. This means that Alice can now begin giving the details of her order by invoking the `AskColor` and `AskSize` intents. For implementation details of these features, see Chapter 4.

Chapter 4

Implementation of Final Design

As described in Section 2.3.1, the main goals of the conversational AI interface's design are to empower students with little or no programming experience to learn about computing and AI while developing agents that can converse, share data with mobile phone apps, and generate responses using machine learning. A diagram showing the user workflow of the original ConvoBlocks interface is shown in Figure 4-1.

Figure 4-1: User workflow to create a conversational AI agent. The user first implements the Voice User Interface (VUI) and endpoint function using a block-based interface. The blocks are converted to JSON and JavaScript, which define the agent's functionality on Alexa devices [30].

To enable the design decisions I made in Section 3.4, the following capabilities were implemented in Stateful ConvoBlocks:

- the ability to specify a name for a conversation state, which enables the agent to scope its conversations,
- the ability to specify a state property for each intent, which enables the agent to store and retrieve the state/intent relationship for the skill,
- the ability to display state-based grouping of intents, which allows the interface to

communicate a clear information hierarchy to the programmer,

- and the ability to manage the conversation flow between an end user and the agent based on the end user's input and the agent's responses. This enables the developer to program a set of conditions in the endpoint function that determine when the agent transitions to another state. This transition between states may involve the agent-user pair
  - moving forward with the conversation,
  - back up to the last stage of the conversation,
  - jump to another topic in the conversation.

4.1 State Intent Association with Tabs

In this section, I describe the implementation involving state/intent association in the Designer view.

To create the tabs functionality in the web view that display states, I overloaded the constructor of the *DesignerEditor* class with additional parameters, `useTabs` (boolean) and `convoStates` (ArrayList of state names).

```
public DesignerEditor(ProjectEditor projectEditor, S sourceNode,
                      V componentDatabase, W visibleComponentsPanel,
                      boolean useTabs, ArrayList<String> convoStates) {
    ...
    if (useTabs) {
    //create a tab for each state in the convoStates list
    //create a set of visible + nonvisible components in each tab
    //add the functionality of selecting the '+' button
    ...
    }
    ...
}
```

DesignerEditor is the ancestor of all design in App Inventor. The *AlexaDesignEditor* class inherits from *DesignerEditor*, calling the overloaded constructor with `useTabs` set to true and `convoStates` containing the default `global` state when the project is first loading (before it reads and loads the content of any JSON file that stores the data of the developing skill). In this constructor, I added a section of code that only executes when

`useTabs` is true, creating a set of *AlexaVisibleComponentsPanel* (the Echo dot image) and *AlexaNonVisibleComponentsPanel* (purple box in Figure 3-5).

In the current case, `useTabs` is only set to true in the Alexa designer. In the designer for regular App Inventor for mobile app development, `useTabs` is set to false, so the state-related feature is limited to App Inventor's Alexa interface only. In the future, a similar tab panel feature may be added to the regular App Inventor interface too. Therefore, I put the tab-related functionality in the *DesignerEditor* class so future developers of App Inventor only needs to toggle the `useTabs` flag to extend the feature to other parts of App Inventor.

The actual state/intent association is stored in the *skillname.alexa*. By adding a new `State` property to the *MockIntent* class, I add a new JSON entry with key `State` in each Intent component JSON object. Here is the JSON `AskColor\verb` intent object. Notice that it contains an entry `State` with valuee `Ordering`|. This represents that intent `AskColor` is of state `Ordering`

```
1    {...,{"$Name":"AskColor", "$Type":"Intent", "$Version":"1", "State":"Ordering", "
        Utterances":"I want a {Color} dress, give me color {Color}, 0", "Uuid":"
        1206787999"},...}
```

Then when a project is reloaded, it reads `state` fields from *skillname.alexa* and re-populates the convoStates list and recreates a tab and its contents to deliver the visual representation of tabs, intents and states in the front end (that we can see in Figure 3-5).

When Lorina is developing the *Dress-ordering Service* agent, she creates new states by clicking the `+` tab in the tab panel and entering a new state name, say `Ordering`, in the prompt window. This creates a new tab with title `Ordering` as well as a new set of Alexa Visible and Non-visible components panel. When she drops a new intent, say `AskColor`, somewhere on the `Ordering` tab, the new intent is created and appears on the Non-visible Components panel, and `AskColor` intent automatically gets *State* property of `Ordering` (the state corresponding to the currently selected tab). This is enabled by the *AlexaNonVisibleComponentsPanel* class I created, which extends the *SimpleNonVisible-ComponentsPanel* class, overriding its *addComponent()* function, which changes the *State* property of the newly added Intent component. I also override the *onPropertyChange()* listener so that when an intent state property is changed, the intent component is moved to the correct tab panel.

4.2 Blocks Interface

Aside from the Designer view, in order to communicate the state/intent association to developers like Lorina in the Blocks editors as well, I implemented the new design (see Section 3.4.3) of the `when-intent-spoken` block.

This `when-intent-spoken` block is an event handler. It defines what occurs when a specific intent is spoken to Alexa. This block can contain regular App Inventor blocks (e.g., an if-statement block) as well as other endpoint blocks, such as the `say` block. The inner blocks that this block contains define the JavaScript for a skill's endpoint. The drop-down menu contains a list of intents defined by VUI components in the Designer view.

The display of all the `when-intent-spoken` blocks are defined in *components.js*. By adding the following code snippet to *domToMutation*, it obtains the `State` property value for the selected intent instance in the block's drop-down menu and describes it before the `do...` plug-in.

```
...
if (this.typeName == 'Intent' && !this.isGeneric) {
    var state = top.BlocklyPanel_getComponentInstancePropertyValue(this.instanceName,
        "State");
    this.appendDummyInput('STATEINFO').appendField('in state ').appendField(state);
}
...
}
```

For effective intent-screening based on states, I implemented a `current state` field in *alexa.js*, which generates the endpoint Lambda function (as discussed in Section 2.3.2) for Alexa. To implement a new block designed to change the current state of the conversation, I defined the visual display of the new block in *voice.js*. The block takes in a string input and changes the `current state\verb` to the value of the string.

```
Blockly.Blocks.voice_state_set = {
    category: 'Voice',
    helpUrl: Blockly.Msg.LANG_VOICE_SAY_HELPURL,
    init: function () {
        this.setColour(Blockly.VOICE_LAMBDA_CATEGORY_HUE);
```

```
        this.appendValueInput("STATE")
            .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("text", Blockly.
                Blocks.Utilities.INPUT))
            .appendField("set current state to")
            .setAlign(Blockly.ALIGN_RIGHT);
        this.setPreviousStatement(true, null);
        this.setNextStatement(true, null);
        this.setTooltip(function () {
            return Blockly.Msg.LANG_VOICE_SAY_TOOLTIP;
        });
    },
    typeblock: []
};
```

To implement the logic of this new `set-current-state` block, I added the following code snippet in *alexa.js* under `case 'voice' of \textit{AI.Blockly.Alexa.createLambdaChildren` current-state' block to the `current state` field in the generated Lambda function. Note that this implementation requires users to use state names that do not contain any apostrophes, otherwise the program may end up generating invalid strings.

```
case 'voice_state_set':
    var state_input = currBlock.getInputTargetBlock("STATE");
    if (state_input == null) {
        break;
    }
    func += "current_state = '" + state_input.getFieldValue("TEXT") + "';\n";
    break;
```

The intent pre-screening based on states is implemented in *alexa.js*. In the `canHandle()` function of *when-intent-spoken* blocks in *alexa.js*, I added conditions that check for the state property of the intent and compare it to the current state. If the state property of the intent is not 'global' and does not match the current state of the conversation, a `wrong-state-flag` is raised and a *WrongStateHandler* that I added handles it by telling Alexa to generate the response *'This intent cannot be invoked in the current state.'* Otherwise, the endpoint function proceeds to execute the behaviors Lorina, the programmer on Stateful ConvoBlocks, defined in the *when-intent-spoken* block for this intent. Console logs enable easy debugging in the AWS console for these logic.

With this implementation of the Stateful ConvoBlocks interface and its improved expressiveness, Lorina can build a dress-ordering agent for her sister, Alice, that supports the dialogue features discussed in Section 3.1. Figure 4-2 shows the block code that creates such an agent and Figure 4-3 shows example conversations Alice can have with Lorina's agent. The full implementation of the Stateful ConvoBlocks interface can be found in Appendix A.

Figure 4-2: Blocks editor that includes example code for Lorina's *Dress-ordering Service* agent, developed using this Stateful ConvoBlocks interface implementation.

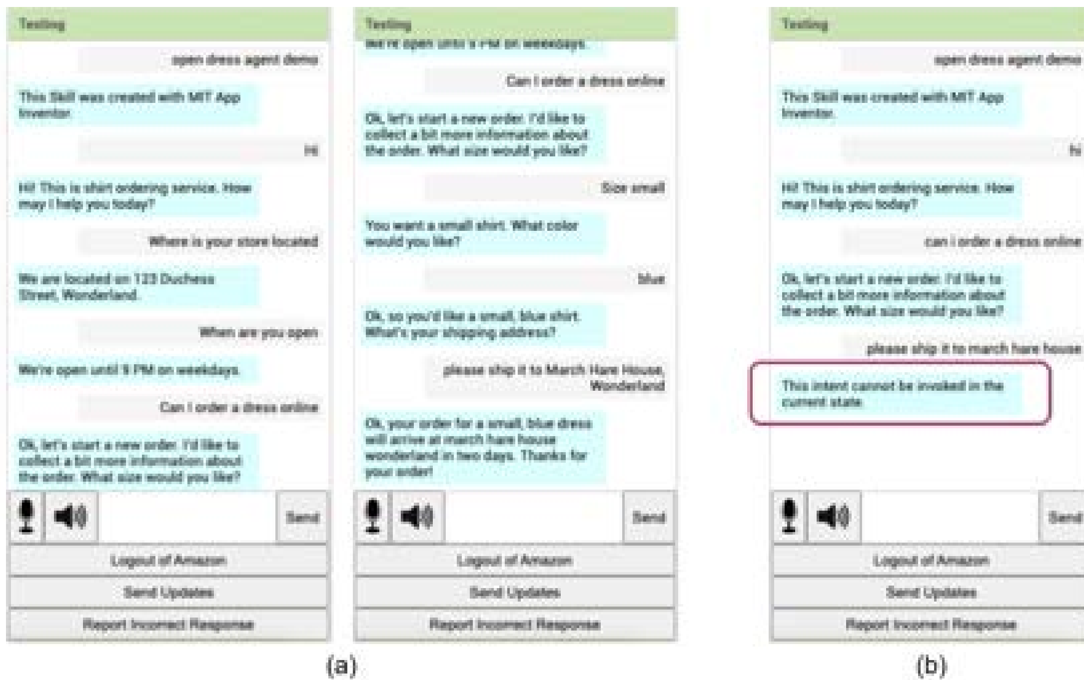

Figure 4-3: (a) Alice's conversation with the *Dress-ordering Service* agent, developed in this Stateful ConvoBlocks implementation. The conversation is very similar to the one in Section 3.1. (b) Alice trying to trigger an intent in the "Shipping" state while the conversation is still in "Ordering" state.

Chapter 5

User Study Design

As discussed in Chapter 1, analysing people's perceptions of conversational agents is important in for effective CA development and better curriculum design. This is especially true for *conversational* technology since and people naturally build relationships with it, just like with other humans [23]. This chapter describes the user study design for my Stateful ConvoBlocks interface to investigate the relationship between learners and CAs created with Stateful ConvoBlocks.

5.1 Research Questions

The main purpose of this study is to investigate the usability and learnability of Stateful ConvoBlocks, and whether the new changes improves the expressiveness of App Inventor in specifying multi-turn conversations. With the study, I aim to answer the following research questions:

RQ5.1.1: How will the new features of Stateful ConvoBlocks affect the usability of the interface?

- Specifically, are the state-related functionalities intuitive to novice programmers?

RQ5.1.2: How will the new features of Stateful ConvoBlocks affect the learnability of the interface?

- Specifically, does the state-related functionalities help participants understand conversational AI concepts?
- How does the state-related functionalities affect participants' self-identity as programmers?

- Is there a difference across age groups (i.e. parents and their children)?

RQ5.1.3: Does the new features of Stateful ConvoBlocks help learners specify complex conversations?

- Specifically, does the state-related functionalities help programmers build a mental image of their conversational agent's design when they program the agent?
- Does this affect their perceived naturalness of the agents they build?

5.2 Participants

To recruit participants, I will send out interest forms to K-12 education email lists. The interest form will as potential participants to indicate their availability over two sets of three-day periods in June.

To be a participant, the only requirements are to have an internet-connected computer and the ability to test Android apps and Alexa-based agents, which can be done with any Alexa-enabled device, including an Echo Dot, the Alexa App, or any Alexa simulator on a computer. There is a built-in Alexa simulator on the Stateful ConvoBlocks interface, which will be available to all participants. The goal is to obtain valid responses from 15 to 20 pairs of students and their parents, who will then be separated into two groups of roughly the same size based on the availability they indicated on their interest form.

Overall, the workshops are designed for students aged 11-18 and their parents to learn about conversational agents, how to program them, and what it means to have them in the world. Among the participants, one group (the "Stateful Group") will be introduced to the new Stateful ConvoBlocks interface, while the other (the "Original Group") will use the original ConvoBlocks interface. The same set of Conversational AI concepts will be taught to both groups.

5.3 Workshop Design Outline

In the workshops, students and their parents will complete programming curriculum and societal impact activities over two days. Each day involves around 5 hours of Zoom interaction.

5.3.1 Day 1

Original Group

At the start of the first day, we will provide participants in the Original group with a pre-survey (as described in Section 5.3.4). Next, we will lead participants through a tutorial, introducing them to the original ConvoBlocks interface and key conversational AI vocabulary, like "intents", "training" and "testing". The agent developed in this tutorial could respond with basic information about a clothing store when asked. Throughout the day, students could either follow along with the group tutorials or go ahead and complete extra tutorials.

Next, in the second programming session, participants will learn to build another agent. This agent demonstrates the fundamental conversational AI concept of entity extraction using slots, enabling the agent to generate a customer order based on the clothing size participants give during the conversation.

In the third programming tutorial, we will lead participants to build a fully-fledged clothing store agent. This agent has basic capabilities to answer simple inquiries about certain facts about the store, as well as more advanced capabilities to make a new online order, confirm shipping address, and going back to edit the order. Using the original ConvoBlocks interface, this agent will contain seven intents and two slots. Participants will then be encouraged to think about and experiment with ways that an end-user may interact with the new agent.

Finally, towards the end of the programming activities on the first day, participants will engage in an ideation session in which they will envision the future of conversational agents, including answering prompts about what their ideal agent might look like, sound like, say and do, and compare that ideal agent with their experience creating agents with the original ConvoBlocks.

Stateful Group

Similar to the Original group, the Stateful group will be provided with the same pre-survey at the start of the first day. Their first and second programming session is also going to be identical to that of the Original group. Those two tutorials will teach them to build a simple inquiry conversation with an agent, which will not involve state-related

functionalities in any case.

In the third programming session, participants in the Stateful group will be introduced to states. We will first engage them in a group discussion that brainstorms what a fully-fledged clothing store agent should be able to do. Participants will draw up a picture a shared virtual whiteboard of the conversational stages that the clothing store agent may involve, such as answering simple facts about the store, making new orders online, confirming shipping address, editing existing orders, etc. They will also draw out the logic of how the conversational path should turn between each stage of the conversation. After the discussion, we will lead participants to build said clothing store agent using Stateful ConvoBlocks. We then encouraged students to try and experiment with different ways to scope the agent's conversational flow and test how that would change the agent's response to a particular utterance at a particular stage of the conversation.

Towards the end of the first day, participants in the Stateful group will also engage in an ideation session about the future of conversational agents, and compare their image of the ideal agent with their experience using Stateful ConvoBlocks.

5.3.2 Day 2

Original Group

On the second day, the original group will engage in two more programming sessions. Using the original ConvoBlocks interface, we will first walk participants through building a storybook agent that tells the story of *Little Red Ridding Hood*. The story will be broken down into six segments and participants will learn to give end-users control over how the story goes using conditionals. In the process they will learn about programming concepts like conditional statements and lists.

During the second programming session, participants in the original group will be provided with a PDF tutorial on creating a cookbook agent using the original ConvoBlocks interface that guides the end-user through steps of making a banana bread. Participants are encouraged to build the agent on their own using the PDF tutorial, or they can go on and build any other agent that involves more than two intents. We will be taking questions from participants and helping anyone in need.

The second day will conclude with participants filling out a mid-survey, where they

will revisit their choices and responses in the pre-survey, assessing whether their opinions have changed. They will also be asked about their learning experience with the original ConvoBlocks interface, particularly the level of control they felt over the agents they created. Finally, they will be asked to give their own definition of conversational concepts taught in the first day, such as skills, intents, slots, utterances, and reflect on their connections.

Stateful Group

On the second day, the Stateful group will also engage in two more programming sessions. Using the Stateful ConvoBlocks interface, they will first be walked through building a storybook agent that tells the story of *Little Red Ridding Hood*. Similar to the tutorial given to the Original group, the story will be broken down into six segments. However, the tutorial given to the Stateful group will use states instead of conditionals and lists to control the story flow. In the process they will gain more insight into what conversational states can be used for.

During the second programming session, participants in the Stateful group will be provided with a PDF tutorial on creating a cookbook agent using the Stateful ConvoBlocks interface that guides the end-user through steps of making a banana bread. Like the Original gropu, participants in the Stateful group are also free to either build the cookbook agent or an agent of their own choice. This session is intended to allow participants to practice building multi-turn conversations with states. We will be taking questions from participants and helping anyone in need.

Like those in the Original group, participants in the Stateful group will also end their second day by filling out a mid-survey. In addition to revisiting their answers in the pre-survey, participants in the Stateful group will also reflect on their experience learning about states, particularly the level of control they felt over their agents, concepts they found challenging, etc. Finally, they will be asked to give their own definition of conversational AI concepts taught on day one, including skills, states, intents, slots, and their relation.

5.3.3 Day 3

For both groups, the third day will involve three rounds of societal impact activities. First, participants will be organized into teams of two student-parent pairs. In the first round, we will give a presentation about the computational action process. Afterwards, teams will discuss the presentation and how it relates to conversational agents and human identities.

In the second round, we will give a presentation discussing the UN Sustainable Development Goals, in order to give participants a jumping off place for finding issues that affect people in their community. Teams will then reconvene to discuss their ideas for the three most important environmental and technological changes people could make to foster sustainability in their community. Students and parents will share the knowledge they gained from their experience with conversational AI and discussed how AI can help with the issues presented. Finally, each team will present their ideas and/or prototypes to the group. Throughout the activities, participants can develop conversational agents which could either help address the challenges presented, or help them present their ideas. This can be built upon the agent they developed during Day 2. In the end, students will complete a final post-survey, reflecting on how their opinions about themselves, conversational agents, and their partner models may have changed.

5.3.4 Data Collection

Data will be collected throughout the three-day workshops, which will involve three different surveys: a pre-survey, mid-survey and post-survey. These surveys will collect information about participants' demographics, their partner models with the agent, as well as their self-efficacy and identities as learners and programmers. The surveys will included multiple-choice, 5-point Likert scale and long-answer questions that allow participants to expand on their Likert scale answers. Participants will be told not to search online or discuss answers with others while completing the surveys.

Aside from the surveys, data will also be collected during ideation sessions and during the final presentation. Participants will be asked to record their discussion on a shared whiteboard during the ideation sessions, and the whiteborads are then collected along with final presentations and skills teams created. Code that participants used to create

the agents for their final presentation will also be collected after the workshop.

5.3.5 Data Analysis

Both quantitative and qualitative analysis will be performed on the data collected from the workshops.

For quantitative analysis, we will perform statistical tests corresponding to the distributions of the data, both across time ranges for the same sample of participants (e.g. compare parents' data between pre- and mid-survey) and across independent samples of participants with the same timestamp (e.g. compare parents' data with children's data in mid-survey). This is attempting to address RQ5.1.2 and RQ5.1.3. Final agents students developed will also be analyzed for its complexity and flexibility by studying the number of blocks they used, types of blocks they used, number of states they created and the balance of the conversational flow they constructed [36].

To investigate the effect of the new features of Stateful ConvoBlocks and identify common characteristics participants desired in conversational agents, qualitative data from the workshop (such as long–answer responses, white board discussions and final presentation) will be analyzed through thematic analysis and organized by participant age group, gender, and prior experience with AI and App Inventor. Results from these qualitative data will help inform RQ5.1.1 and inform future design and pedagogy guidelines for the Stateful ConvoBlocks interface.

Chapter 6

Summary of Contributions and Future Work

6.1 Summary

For this thesis, I developed a State-based version of the ConvoBlocks interface and designed associated user study. I also conducted an overview of conversational AI, and analyzed four conversational agent platforms. Through this research, I increased the expressiveness and reliability of the ConvoBlocks interface in specifying complex multi-turn conversations by designing and implementing major state-related capabilities. My user study is designed to answer three research questions to evaluate the usability and learnability of the Stateful ConvoBlocks interface as well as investigate the how agent flexibility and curriculum design influence learners' trust, partner model, self-efficacy and identity as programmers. In the following sections, I describe my contributions as well as potential areas for future work.

6.1.1 Current Version

In its current version, Stateful ConvoBlocks supports a simple one-to-many association between states and intents. For an individual intent, it can either be accessible in all states ('global'), or it can be accessible only in one state (programmer-defined state).

To achieve a learner(programmer)-centered design, the final interface provides dynamic learner visual feedback on this state/intent association in the Designer editor Viewer panel (see Figure 3-6, and in the *when-intent-spoken* blocks in the Blocks editor

(see Figure 3-10). In this design, most of the state-related functionalities is non-end-user facing, meaning that end users (like Alice, in our example) don't receive direct information about the state flow in the conversation. As shown in Figure 4-3, end users are informed that they are not in the right stage in the conversation when they try to invoke an intent outside of the current scope of the conversation. These intent screenings are designed to take place in the back end to keep end users' interaction with the agent as natural as possible.

In the current design of the user study, I present a three-day curriculum plan along with surveys and hackathon-like activities to teach core conversational AI concepts and computational action. Currently, the main theme for the social impact activities is centered around sustainability in the community. The study is designed to experiment with the impact of the current version of Stateful ConvoBlocks on learners of different age groups, and provide design feedback to improve the interface as well as pedagogy recommendations for the curriculum.

6.1.2 Known Limitations

In the current Stateful ConvoBlocks, each intent can only be associated with one state. However, in natural language dialogues, especially complicated branching conversations, it might make sense for a request-response pair (i.e. an intent under ConvoBlocks) to be accessible across several, but not all, topics. That kind of conversation requires a many-to-many association between states and intents, which is currently not supported in the interface.

Furthermore, even though the interface is designed with a learner-centered principle in mind, it does not provide any visual feedback to the learner about states in the *Components* panel or *Properties* panel in the Designer editor, or in the text-enabled *Testing* panel. The display of the "state" property on when-intent-spoken block sometimes goes blank when the programmer switches between projects (see Figure 6-1).

Finally, during my time working on this project, the world of generative AI exploded and changed the way people view Conversational AI in general. There is both a great deal of excitement and a lot of anxiety around where Conversational AI is headed and how to best integrate this technology. Voiceflow introduced their GPT-powered AI Builder, allowing programmers to experiment with large language models, try prompt chaining,

Figure 6-1: *when-intent-spoken* block for "StoreHours" intent (a) before switching to a different project and back (b) after switching to a different project and back.

generate an instant assistant, give the agent memory, etc. [13] Amazon also just announced their plan of integrating generative AI into Alexa in the near future. These changes has significantly expanded the capabilities of conversational AI, enabling more natural, personalized, and engaging interactions between humans and machines.

6.2 Future Work

6.2.1 Immediate Future

This section discusses work that I plan to do in the short term future (i.e., about a month). A few App Inventor team staff has tested the current Stateful ConvoBlocks platfor. Based on their feedback on issues they've uncovered, I plan to enhance the implementation with the following changes:

- Fix the "state" property value display issue with *when-intent-spoken* blocks when the programmer switch between project editors (as discusses in Section 6.1.2).
- Add a learner(programmer)-feedback feature in the Properties panel of each intent component to emphasize the state-intent association for programmers. This can be done by overloading the *addProperty()* function in the *MockComponent* class. An illustration of what this design might look like is highlighted in *purple* in Figure 6-2.
- Add a learner(programmer)-feedback feature in the Components panel in the Designer view. An illustration of what this design might look like is highlighted in *yellow* in Figure 6-2. Each state is a mock component wrapped around its corresponding intents and slot components. To implement this, I plan to reference the structure of "VerticalArrangement" layout component for inspiration. Since the same code that generates this Components panel also generates the list of agent-related blocks drawers in the lower left of the Blocks editor, this implementation

will also enhance programmer-feedback feature in the Blocks editor page.

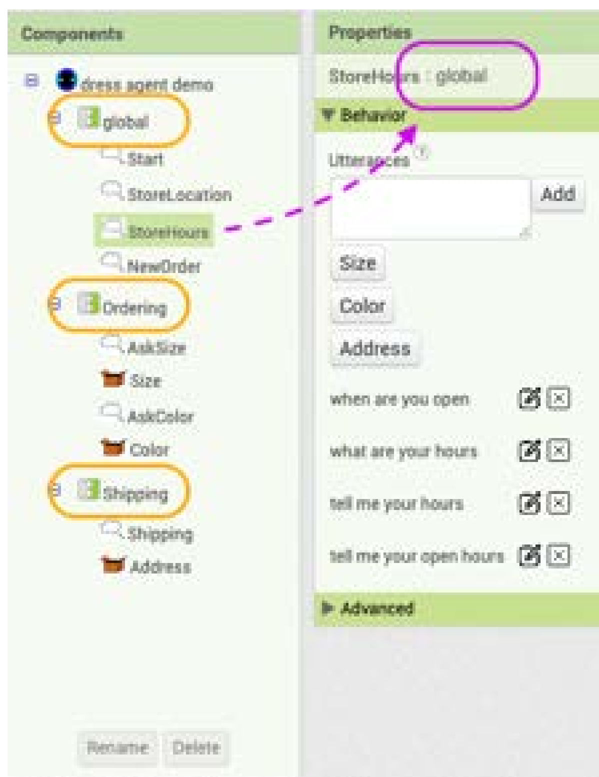

Figure 6-2: Mockup for the improved *Components* panel (left) and *Properties* panel (right) in the Designer editor.

6.2.2 Near Future

This section discusses work that can be done within the next few months. I plan to execute these plans with the MIT App Inventor team.

In terms of design and technical implementation, I plan to further enhance interface-programmer feedback about state/intent association with the following:

- In the "Advanced" drawer in the Properties panel for the intents, implement additional feature that allows programmers to associate an intent with multiple states. This addresses the first issue mentioned in Section 6.1.2.
- Add learner(programmer)/end user-feedback feature in the Testing panel on App Inventor. Currently, App Inventor displays conversations with agents in a chat-window format. Hence I plan to implement this feature by adding timestamp-like log messages that record the changing process of the current state, which should help illustrate the conversation flow.

- Improve displaying mechanism of the Blocks editor. As shown in Figure 6-3, I plan to add the tab panel feature that I implemented in the Designer view in the Blocks editor as well. Each tab would only contain blocks relevant to that state, so this design should help programmers organize their workspace when developing complex agents.

Figure 6-3: Mockup for the improved blocks editor interface.

In terms of the user study, we need to observe how learners actually interact with the Stateful ConvoBlocks interface. I plan to send out recruitment email in May and conduct two sets of three-day workshops in June. Qualitative and quantitative analysis will then be performed on the collected data, as described in Chapter 5. After analyzing the results, we will reflect back on the research questions detailed in Chapter 5. Agent design recommendations and pedagogy recommendations will then be formulated and delivered.

6.2.3 Far Future

This section describes interesting long-term projects that are related to this research.

- Learners' trust and partner model with more complicated and flexible conversational agents, and how these results might help improve the Stateful ConvoBlocks system so that it helps programmers build healthy relationships with the agent and understand their actual level of trustworthiness.
- Development of a well-rounded curriculum based on the Stateful ConvoBlocks in-

terface. Given the new feature of this interface, it allows people with little or no programming background to build much more complex chatbots. To better facilitate learners' self-efficacy and identity as programmers, it is worthwhile to develop a new pedagogy framework that will take people from beginners to building industrial-level agents.

- Extend state-based functionalities to regular App Inventor. As discussed in Chapter 4, the implementation of Stateful ConvoBlocks specifically leaves room for future developers to extend state-based workflow organization strategies to regular App Inventor for mobile app development.

The Stateful ConvoBlocks builds upon the interaction framework of Amazon Alexa. By introducing the concept of conversation states, the platform enables learners with little or no prior experience with AI technology to build complex multi-turn conversational agents, extracting away the syntax of text-based languages. Work described in this thesis gives a promising start that helps foster AI literacy for nearly anyone, and educators, researchers and agent developers can better prepare K-12 students for an agent-filled world.

Appendix A

Implementations for State-based ConvoBlocks

This appendix contains java and javascript code I implemented for the State-based ConvoBlocks interface.

A.1 Tabs

The following code implements the Tabs feature in the Viewer panel of Designer editor.

A.1.1 DesignerEditor.java

Add fields to DesignerEditor class.

```
protected ArrayList<String> convoStates;
protected HashMap<String, SimpleNonVisibleComponentsPanel<?>> statesPanelMap;
protected TabPanel tabPanel;
protected List<DropTarget> allOtherVisNonVisComponentsPanels;//all visible and non-
    visible component panels other than the default set
```

A.1.2 AlexaDesignEditor.java

Constructor of AlexaDesignEditor class is changed to:

```
public AlexaDesignEditor(ProjectEditor projectEditor, AlexaDesignNode sourceNode) {
   super(projectEditor, sourceNode, AlexaDeviceDatabase.getInstance(sourceNode.
       getProjectId()),
```

```java
    new AlexaVisibleComponentsPanel(new AlexaNonVisibleComponentsPanel<>()), true,
        new ArrayList<String>(Arrays.asList("global")));

  palettePanel = new AlexaPalettePanel(this);
  palettePanel.loadComponents(new DropTargetProvider() {
    @Override
    public DropTarget[] getDropTargets() {
      List<DropTarget> dropTargets = root.getDropTargetsWithin();
      dropTargets.add(getVisibleComponentsPanel());
      dropTargets.add(getNonVisibleComponentsPanel());
      dropTargets.addAll(getAllOtherComponentPanels());
      return dropTargets.toArray(new DropTarget[0]);
    }
  });
  palettePanel.setSize("100%", "100%");
  componentDatabaseChangeListeners.add(palettePanel);
}
```

Overloaded constructor:

```java
public DesignerEditor(ProjectEditor projectEditor, S sourceNode,
                      V componentDatabase,
                      W visibleComponentsPanel,
                      boolean useTabs, ArrayList<String> convoStates) { // [CLAIRE] if
                           useTabs == true, wrap tabs around components, useTabs only
                          true in AlexaBlocksEditor
  super(projectEditor, sourceNode);

  this.sourceNode = sourceNode;
  this.componentDatabase = componentDatabase;

  //[CLAIRE] mapping between states and panels? between states and list of intents?
  //int selectedIndex = tabPanel.getTabBar().getSelectedTab();
  //getWidget(int index) --> Gets the child widget at the specified index.
  //   getWidgetIndex(Widget widget) --> Gets the index of the specified child widget
      .

  // Get reference to the source structure explorer
  sourceStructureExplorer =
      SourceStructureBox.getSourceStructureBox().getSourceStructureExplorer();
```

```
18
19    // Create UI elements for the designer panels.
20    Widget view;
21
22    //[CLAIRE] wrap tab around each set of visible + nonvisible components
23    if (useTabs) {
24      //statesConstructorHelper(visibleComponentsPanel, view) --> remove convoStates
          from this constructor, move everything in if statement into new function,
          overriding it in AlexaDesignEditor
25      this.statesPanelMap = new HashMap<String, SimpleNonVisibleComponentsPanel<?>>();
26      this.allOtherVisNonVisComponentsPanels = new ArrayList<DropTarget>();
27      this.convoStates = convoStates;
28      this.visibleComponentsPanel = visibleComponentsPanel;
29      nonVisibleComponentsPanel = visibleComponentsPanel.getNonVisibleComponentsPanel
          ();
30      componentDatabaseChangeListeners.add(nonVisibleComponentsPanel);
31      componentDatabaseChangeListeners.add(visibleComponentsPanel);
32      //Create an empty tab panel
33      this.tabPanel = new TabPanel();
34      for (String ele : convoStates) {
35        //create a set of visible + nonvisible components in each tab
36        //create contents for tabs of tabpanel
37
38        DockPanel componentsPanel = new DockPanel();
39        componentsPanel.setHorizontalAlignment(DockPanel.ALIGN_CENTER);
40        if (ele == "global") {
41          //use default visible/nonvisible components
42          this.statesPanelMap.put(ele, nonVisibleComponentsPanel);
43          componentsPanel.add(visibleComponentsPanel, DockPanel.NORTH);
44          componentsPanel.add(nonVisibleComponentsPanel, DockPanel.SOUTH);
45        } else {
46          //create new visible & nonvisible components
47          W newVisibleComponentsPanel = visibleComponentsPanel.copy();
48          SimpleNonVisibleComponentsPanel<T> newNonVisibleComponentsPanel =
              newVisibleComponentsPanel.getNonVisibleComponentsPanel();
49          this.statesPanelMap.put(ele, newNonVisibleComponentsPanel);
50          componentDatabaseChangeListeners.add(newNonVisibleComponentsPanel);
51          componentDatabaseChangeListeners.add(newVisibleComponentsPanel);
52          componentsPanel.add(newVisibleComponentsPanel, DockPanel.NORTH);
```

```
53          componentsPanel.add(newNonVisibleComponentsPanel, DockPanel.SOUTH);
54          allOtherVisNonVisComponentsPanels.add(newVisibleComponentsPanel);
55          allOtherVisNonVisComponentsPanels.add(newNonVisibleComponentsPanel);
56        }
57        componentsPanel.setSize("100%", "100%");
58        //create titles for tabs
59        String currtabTitle = ele;
60        //create tabs
61        tabPanel.add(componentsPanel, currtabTitle);
62      }
63      //add "+" widget that serves as add button
64      tabPanel.add(new Label(), "+");
65      tabPanel.addSelectionHandler(new SelectionHandler<Integer>() {
66        @Override
67        public void onSelection(SelectionEvent<Integer> event) {
68            if (event.getSelectedItem() == tabPanel.getWidgetCount() - 1) {
69              addState();
70            }else {
71              moveRoot();
72            }
73        }
74      });
75      //select first tab
76      tabPanel.selectTab(0);
77      // Add the widgets to the root panel.
78      view = tabPanel;
79    } else {
80      this.visibleComponentsPanel = visibleComponentsPanel;
81      nonVisibleComponentsPanel = visibleComponentsPanel.getNonVisibleComponentsPanel
          ();
82      componentDatabaseChangeListeners.add(nonVisibleComponentsPanel);
83      componentDatabaseChangeListeners.add(visibleComponentsPanel);
84      DockPanel componentsPanel = new DockPanel();
85      componentsPanel.setHorizontalAlignment(DockPanel.ALIGN_CENTER);
86      componentsPanel.add(visibleComponentsPanel, DockPanel.NORTH);
87      componentsPanel.add(nonVisibleComponentsPanel, DockPanel.SOUTH);
88      componentsPanel.setSize("100%", "100%");
89      view = componentsPanel;
90    }
```

```
91
92    // Create designProperties, which will be used as the content of the PropertiesBox
        .
93    designProperties = new PropertiesPanel();
94    designProperties.setSize("100%", "100%");
95
96    root = null;
97
98    initWidget(view);
99    setSize("100%", "100%");
100  }
101
102  public DesignerEditor(ProjectEditor projectEditor, S sourceNode,
103                        V componentDatabase,
104                        W visibleComponentsPanel) {
105    this(projectEditor, sourceNode, componentDatabase, visibleComponentsPanel, false,
        null);
106  }
```

Added functions to put mockIntent at the right tab:

```
1 //change root to display alexa widget
2   public void moveRoot() {
3     T rootPt = getRoot();
4     if (rootPt==null){
5       return;
6     }
7     DockPanel DockPt = (DockPanel) tabPanel.getWidget(tabPanel.getTabBar().
        getSelectedTab());
8     W VisComponentPt = (W) DockPt.getWidget(0);
9     VisComponentPt.setRoot(rootPt);
10    SimpleNonVisibleComponentsPanel<T> nonVisComponentPt = VisComponentPt.
        getNonVisibleComponentsPanel();
11    nonVisComponentPt.setRoot(rootPt);
12  }
13
14  //AddStateDesigner() --> add new state, update states list property, add a panel
15  //override in AlexaDesignEditor
16  public void addState() {
17    //throw new UnsupportedOperationException();
```

```
18      //update states list
19      String defaultStateName = "State" + this.convoStates.size();
20      String newStateName = Window.prompt("Please enter the state name: ",
            defaultStateName);
21      if(newStateName.isEmpty() || newStateName == null){
22        return;
23      }
24      this.convoStates.add(newStateName);
25      //add a tab panel
26      DockPanel componentsPanel = new DockPanel();
27      componentsPanel.setHorizontalAlignment(DockPanel.ALIGN_CENTER);
28      //create new visible & nonvisible components
29      W newVisibleComponentsPanel = visibleComponentsPanel.copy();
30      SimpleNonVisibleComponentsPanel<T> newNonVisibleComponentsPanel =
            newVisibleComponentsPanel.getNonVisibleComponentsPanel();
31      componentDatabaseChangeListeners.add(newNonVisibleComponentsPanel);
32      componentDatabaseChangeListeners.add(newVisibleComponentsPanel);
33      this.statesPanelMap.put(newStateName, newNonVisibleComponentsPanel);
34      componentsPanel.add(newVisibleComponentsPanel, DockPanel.NORTH);
35      componentsPanel.add(newNonVisibleComponentsPanel, DockPanel.SOUTH);
36      componentsPanel.setSize("100%", "100%");
37      //create titles for tabs
38      String currtabTitle = newStateName;
39      //create new tab panel
40      //tabPanel.add(componentsPanel, currtabTitle);
41      tabPanel.insert(componentsPanel, currtabTitle,
42                      tabPanel.getWidgetCount() - 1);
43      tabPanel.selectTab(tabPanel.getWidgetIndex(componentsPanel));
44      allOtherVisNonVisComponentsPanels.add(newVisibleComponentsPanel);
45      allOtherVisNonVisComponentsPanels.add(newNonVisibleComponentsPanel);
46      moveRoot();
47
48  public void createVisNonVisComponents(String panelName) {
49      this.convoStates.add(panelName);
50      DockPanel componentsPanel = new DockPanel();
51      componentsPanel.setHorizontalAlignment(DockPanel.ALIGN_CENTER);
52      W newVisibleComponentsPanel = visibleComponentsPanel.copy();
53      SimpleNonVisibleComponentsPanel<T> newNonVisibleComponentsPanel =
            newVisibleComponentsPanel.getNonVisibleComponentsPanel();
```

```
54   this.statesPanelMap.put(panelName, newNonVisibleComponentsPanel);
55   componentDatabaseChangeListeners.add(newNonVisibleComponentsPanel);
56   componentDatabaseChangeListeners.add(newVisibleComponentsPanel);
57   componentsPanel.add(newVisibleComponentsPanel, DockPanel.NORTH);
58   componentsPanel.add(newNonVisibleComponentsPanel, DockPanel.SOUTH);
59   allOtherVisNonVisComponentsPanels.add(newVisibleComponentsPanel);
60   allOtherVisNonVisComponentsPanels.add(newNonVisibleComponentsPanel);
61   componentsPanel.setSize("100%", "100%");
62   //create titles for tabs
63   String currtabTitle = panelName;
64   //create tabs
65   //this.tabPanel.add(componentsPanel, currtabTitle);
66   this.tabPanel.insert(componentsPanel, currtabTitle, convoStates.size() - 1);//
         tabPanel.getWidgetCount() - 1);
67 }
68
69 //returns a list of all visible and non-visible component panels other than the
       default set
70 public List<DropTarget> getAllOtherComponentPanels() {
71   return allOtherVisNonVisComponentsPanels;
72 }
73
74 //int selectedIndex = tabPanel.getTabBar().getSelectedTab();
75   //getWidget(int index) --> Gets the child widget at the specified index.
76   //   getWidgetIndex(Widget widget) --> Gets the index of the specified child widget
         .
77 public String getSelectedTabName() {
78   int selectedIndex = tabPanel.getTabBar().getSelectedTab();
79   String selectedTabName = convoStates.get(selectedIndex);
80   return selectedTabName;
81 }
82
83 //movetoPanel() --> move component to panel corresponding to correct state
84 //override in AlexaDesignEditor
85 public void movetoPanel(MockComponent m, String panelName) {
86   //throw new UnsupportedOperationException();
87   //SimpleNonVisibleComponentsPanel<?> oldParentComponentPanel =
88   m.removeFromParent();
89   SimpleNonVisibleComponentsPanel<?> newParentComponentPanel = this.statesPanelMap.
```

```
        get(panelName);
90    if(newParentComponentPanel == null) {
91      createVisNonVisComponents(panelName);
92      newParentComponentPanel = this.statesPanelMap.get(panelName);
93    }
94    newParentComponentPanel.addComponent(m);
95  }
96  }
```

A.1.3 AlexaNonVisibleComponentsPanel.java

Created new AlexaNonVisibleComponentsPanel class.

```
1 package com.google.appinventor.client.editor.alexa;
2
3 import com.google.appinventor.client.editor.designer.DesignerEditor;
4 import com.google.appinventor.client.editor.designer.DesignerRootComponent;
5 import com.google.appinventor.client.editor.simple.SimpleNonVisibleComponentsPanel;
6 import com.google.appinventor.client.editor.simple.components.MockComponent;
7 import com.google.appinventor.client.editor.simple.components.MockIntent;
8
9 public class AlexaNonVisibleComponentsPanel<T extends DesignerRootComponent> extends
      SimpleNonVisibleComponentsPanel<T> {
10  @Override
11  public <E extends SimpleNonVisibleComponentsPanel<T>> E copy() {
12    AlexaNonVisibleComponentsPanel<T> result = new AlexaNonVisibleComponentsPanel<>();
13    result.setRoot(this.root);
14    return (E) result;
15  }
16
17  @Override
18  public void addComponent(MockComponent component) {
19    super.addComponent(component);
20    if (root == null) {
21      // Loading in progress
22      return;
23    }
24    if (component instanceof MockIntent) {
25      MockIntent intentComponent = (MockIntent) component;
```

```
      DesignerEditor editorPt = root.getEditor();
      String stateName = editorPt.getSelectedTabName();
      intentComponent.changeProperty(MockIntent.PROPERTY_NAME_STATE, stateName);
    }
  }
}
```

A.1.4 SimpleVisibleComponentsPanel.java

Creates an empty copy of simple visible and non visible components.

```
@Override
  public <E extends SimpleVisibleComponentsPanel<MockAlexa>> E copy() {
    AlexaVisibleComponentsPanel copy = new AlexaVisibleComponentsPanel(
        getNonVisibleComponentsPanel().copy());
    //copy.setRoot(this.root);
    return (E) copy;
}
```

A.1.5 SimpleNonVisibleComponentsPanel.java

```
public <E extends SimpleNonVisibleComponentsPanel<T>> E copy() {
    //throw new UnsupportedOperationException();
    SimpleNonVisibleComponentsPanel<T> copy = new SimpleNonVisibleComponentsPanel<T>()
        ;
    copy.setRoot(this.root);
    return (E) copy;
}
```

A.1.6 MockIntent.java

```
@Override
    public void onPropertyChange(String propertyName, String newValue) {
        // TODO Auto-generated method stub
        super.onPropertyChange(propertyName, newValue);
        if(PROPERTY_NAME_STATE.equals(propertyName) && !newValue.isEmpty()) {
          DesignerEditor EditorPt = (DesignerEditor) this.editor;
```

```
        EditorPt.movetoPanel(this, newValue);
      }
    }
```

A.1.7 MockAlexa.java

```
@Override
public void onDrop(DragSource source, int x, int y, int offsetX, int offsetY) {
  super.onDrop(source, x, y, offsetX, offsetY);
  MockIntent intentComponent;
  if (source instanceof MockIntent) {
    intentComponent = (MockIntent) source;
    DesignerEditor editorPt = (DesignerEditor) super.editor;
    String stateName = editorPt.getSelectedTabName();
    intentComponent.changeProperty(MockIntent.PROPERTY_NAME_STATE, stateName);
  }
}
```

A.2 Blocks

A.2.1 voice.js

```
/**
 * TODO(CLAIRE): added entries to AI.Blockly.Alexa.generatableBlocks to alexa.js,
     blockly.js, blockly-all.js --> necessary?
 * (i.e., not inside an initialize block or anything else)
 * @type {Blockly.BlockSvg}
 * @this {Blockly.BlockSvg}
 */
Blockly.Blocks.voice_state_set = {
  category: 'Voice',
  helpUrl: Blockly.Msg.LANG_VOICE_SAY_HELPURL,//TODO Blockly.Msg.
      LANG_VOICE_STATE_SET_HELPURL --> msg.json
  init: function () {
    this.setColour(Blockly.VOICE_LAMBDA_CATEGORY_HUE);
    //this.fieldVar_ = new Blockly.FieldLexicalVariable(" ");
```

```
13    //this.fieldVar_.setBlock(this);
14    this.appendValueInput("STATE")
15      .setCheck(Blockly.Blocks.Utilities.YailTypeToBlocklyType("text", Blockly.Blocks.
          Utilities.INPUT))
16      .appendField("set current state to")
17      .setAlign(Blockly.ALIGN_RIGHT);
18    this.setPreviousStatement(true, null);
19    this.setNextStatement(true, null);
20    this.setTooltip(function () {
21      return Blockly.Msg.LANG_VOICE_SAY_TOOLTIP; //TODO Blockly.Msg.
          LANG_VOICE_STATE_SET_TOOLTIP
22    });
23  },
24  typeblock: []
25 };
```

A.2.2 alexa.js

```
1 "const PROJECT_NAME = '" + top.BlocklyPanel_getProjectName() + "';\n" +
2 "var current_state = 'global';\n" +
3 "var wrong_state_flag = false;\n";//set to true if the current state does not match
    intent state
4 ...
5 "\n" +//put WrongStateHandler, if wrong_state_flag is true, then speak some generic
    thing
6 "const WrongStateHandler = {\n" +
7 "   canHandle(handlerInput) {\n" +
8 "       return wrong_state_flag;\n" +
9 "   },\n" +
10 "   handle(handlerInput) {\n" +
11 "       const speakOutput = 'This intent cannot be invoked in the current state.';\n\n"
      +
12 "       wrong_state_flag = false;\n" +
13 "       return handlerInput.responseBuilder\n" +
14 "       .withShouldEndSession(false)\n" +
15 "       .speak(speakOutput)\n" +
16 "       .getResponse();\n" +
17 "   }\n" +
```

```
18   "};\n" +
19 ...
20 var state = top.BlocklyPanel_getComponentInstancePropertyValue(block.instanceName, 'State');
21 lambdaFxn += "const " + block.instanceName + "IntentHandler = {\n";
22 lambdaFxn += "  canHandle(handlerInput) {\n";// TODO add here!!! if name match and state not right, set var stateFlag to true
23 lambdaFxn += "      if (handlerInput.requestEnvelope.request.type === 'IntentRequest'\n";
24 lambdaFxn += "          && handlerInput.requestEnvelope.request.intent.name === '" + block.instanceName + "') {\n";
25 if (state != 'global') {
26     lambdaFxn += "        if (current_state == '" + state + "') {\n";
27     lambdaFxn += "          return true;\n";
28     lambdaFxn += "        }\n"
29     lambdaFxn += "        wrong_state_flag = true;\n";
30 
31  } else {
32         lambdaFxn += "        return true;\n";
33 ...
34 //TODO: add case 'voice_state_set'
35 case 'voice_state_set':
36     var state_input = currBlock.getInputTargetBlock("STATE");//need null exception if no block is plugged in
37     if (state_input == null) {
38         break;
39     }
40     func += "current_state = '" + state_input.getFieldValue("TEXT") + "';\n"; //need escape characters in case the state name has apostrophies in it
41     break;
42 }
```

A.2.3 components.js

Added the following code to DomtoMutation, definining the look of the block.

```
1 if (this.typeName == 'Intent' && !this.isGeneric) {
2      var state = top.BlocklyPanel_getComponentInstancePropertyValue(this.instanceName, "State");
```

```
3       this.appendDummyInput('STATEINFO').appendField('in state ').appendField(state);
4 }
```

Bibliography

[1] What is conversational ai? URL https://www.ibm.com/topics/conversational-ai.

[2] URL https://openai.com/blog/chatgpt.

[3] Amazon. Alexa skill development process. https://developer.amazon.com/en-US/docs/alexa/design/design-your-skill.html, 2021. Accessed: 2023-05-16.

[4] Amazon. Alexa developer console. https://developer.amazon.com/alexa/console/ask, 2022. Accessed: 2022-05-23.

[5] Amazon. Tutorial: Build an engaging Alexa skill. https://developer.amazon.com/en-US/alexa/alexa-skills-kit/get-deeper/tutorials-code-samples/build-an-engaging-alexa-skill, 2022. Accessed: 2022-05-18.

[6] Gabriele Barzilai and Lucia Rampino. Just a natural talk? the rise of intelligent personal assistants and the (hidden) legacy of ubiquitous computing. In Aaron Marcus and Elizabeth Rosenzweig, editors, *Design, User Experience, and Usability. Design for Contemporary Interactive Environments*, pages 18–39, Cham, 2020. Springer International Publishing. ISBN 978-3-030-49760-6.

[7] Jessica Van Brummelen, Mingyan Claire Tian, Maura Kelleher, and Nghi Hoang Nguyen. Learning affects trust: Design recommendations and concepts for teaching children – and nearly anyone – about conversational agents, 2022.

[8] Radhika Garg, Hua Cui, Spencer Seligson, Bo Zhang, Martin Porcheron, Leigh Clark, Benjamin R. Cowan, and Erin Beneteau. The last decade of hci research on children and voice-based conversational agents. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems*, CHI '22, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391573. doi: 10.1145/3491102.3502016. URL https://doi.org/10.1145/3491102.3502016.

[9] Google. Dialogflow cx. https://dialogflow.cloud.google.com/cx/, 2021. Accessed: 2021-10-15.

[10] Chandra Khatri, Anu Venkatesh, Behnam Hedayatnia, Raefer Gabriel, Ashwin Ram, and Rohit Prasad. Alexa prize — state of the art in conversational ai. *AI Magazine*, 39(3):40–55, Sep. 2018. doi: 10.1609/aimag.v39i3.2810. URL https://ojs.aaai.org/aimagazine/index.php/aimagazine/article/view/2810.

[11] Allison Koenecke, Andrew Nam, Emily Lake, Joe Nudell, Minnie Quartey, Zion Mengesha, Connor Toups, John R. Rickford, Dan Jurafsky, and Sharad Goel. Racial

disparities in automated speech recognition. *Proceedings of the National Academy of Sciences*, 117(14):7684–7689, 2020. ISSN 0027-8424. doi: 10.1073/pnas.1915768117. URL https://www.pnas.org/content/117/14/7684.

[12] Pradnya Kulkarni, Ameya Mahabaleshwarkar, Mrunalini Kulkarni, Nachiket Sirsikar, and Kunal Gadgil. Conversational ai: An overview of methodologies, applications & future scope. In *2019 5th International Conference On Computing, Communication, Control And Automation (ICCUBEA)*, pages 1–7, 2019. doi: 10.1109/ICCUBEA47591.2019.9129347.

[13] Kim Liu. Introducing voiceflow's gpt-powered ai builder: Voiceflow, May 2023. URL https://www.voiceflow.com/blog/introducing-voiceflows-gpt-powered-ai-builder.

[14] Duri Long and Brian Magerko. What is AI literacy? competencies and design considerations. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–16, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367080. doi: 10.1145/3313831.3376727. URL https://doi.org/10.1145/3313831.3376727.

[15] Gustavo López, Luis Quesada, and Luis A. Guerrero. Alexa vs. Siri vs. Cortana vs. Google Assistant: A comparison of speech-based natural user interfaces. In Isabel L. Nunes, editor, *Advances in Human Factors and Systems Interaction*, pages 241–250, Cham, 2018. Springer International Publishing. ISBN 978-3-319-60366-7.

[16] MIT. Responsible AI for social empowerment and education (RAISE). https://raise.mit.edu/index.html, 2021. Accessed: 2021-09-11.

[17] MIT App Inventor. Artificial intelligence with mit app inventor. https://appinventor.mit.edu/explore/ai-with-mit-app-inventor, 2020. Accessed: 2020-09-10.

[18] Robert J. Moore and Raphael Arar. *Conversational UX design: A practitioner's guide to the natural conversation framework*. ACM Books, 2019.

[19] Christine Murad and Cosmin Munteanu. Designing voice interfaces: Back to the (curriculum) basics. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, page 1–12, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450367080. doi: 10.1145/3313831.3376522. URL https://doi.org/10.1145/3313831.3376522.

[20] Donald A Norman. Natural user interfaces are not natural. *interactions*, 17(3):6–10, 2010.

[21] Hannah (Nicole) Pang. Computational action in action: Process and tools that empower students to make a real-world impact using technology. Master's thesis, Massachusetts Institute of Technology, 2022.

[22] Reddit. Alexa misunderstandings... https://www.reddit.com/r/amazonecho/comments/41159u/alexa_misunderstandings/, 2015. Accessed: 2021-09-01.

[23] William Seymour and Max Van Kleek. Exploring interactions between trust, anthropomorphism, and relationship development in voice assistants. *Proc. ACM Hum.-Comput. Interact.*, 5(CSCW2), oct 2021. doi: 10.1145/3479515. URL https://doi.org/10.1145/3479515.

[24] Sam Smith. Digital voice assistants in use to triple to 8 billion by 2023, driven by smart home devices. Scientific analysis or review, Juniper Research, Hampshire, UK, 2018. URL https://www.juniperresearch.com/press/digital-voice-assistants-in-use-to-8-million-2023.

[25] Caroline L. van Straten, Jochen Peter, Rinaldo Kühne, and Alex Barco. Transparency about a robot's lack of human psychological capacities: Effects on child-robot perception and relationship formation. *J. Hum.-Robot Interact.*, 9(2), jan 2020. doi: 10.1145/3365668. URL https://doi.org/10.1145/3365668.

[26] Google Cloud Tech. Introduction videos. https://cloud.google.com/dialogflow/cx/docs/video, 2021. Accessed: 2022-04-22.

[27] Mike Tissenbaum, Josh Sheldon, and Hal Abelson. From computational thinking to computational action. *Commun. ACM*, 62(3):34–36, feb 2019. ISSN 0001-0782. doi: 10.1145/3265747. URL https://doi.org/10.1145/3265747.

[28] David Touretzky, Christina Gardner-McCune, Fred Martin, and Deborah Seehorn. Envisioning AI for K-12: What should every child know about AI? In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 9795–9799, 2019.

[29] Giedrė Vaičiulaitytė. 25 funny tweets about Amazon Alexa that prove there's nothing artificial about her intelligence. https://www.boredpanda.com/funny-alexa-tweets/?utm_source=google&utm_medium=organic&utm_campaign=organic, 2018. Accessed: 2021-09-01.

[30] Jessica Van Brummelen. Conversational artificial intelligence development tools for K-12 education. In *2019 AAAI Fall Symposium*, 2019.

[31] Jessica Van Brummelen. Tools to create and democratize conversational artificial intelligence. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, 2019.

[32] Jessica Van Brummelen, Tommy Heng, and Viktoriya Tabunshchyk. Teaching tech to talk: K-12 conversational artificial intelligence literacy curriculum and development tools. In *2021 AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI)*, 2021.

[33] Jessica Van Brummelen, Viktoriya Tabunshchyk, and Tommy Heng. "Alexa, can i program you?": Student perceptions of conversational artificial intelligence before and after programming Alexa. In *Interaction Design and Children*, IDC '21, page 305–313, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450384520. doi: 10.1145/3459990.3460730. URL https://doi.org/10.1145/3459990.3460730.

[34] Voiceflow. Voiceflow. https://www.voiceflow.com, 2019. Accessed: 2023-2-15.

[35] David Wolber, Harold Abelson, and Mark Friedman. Democratizing computing with app inventor. *GetMobile: Mobile Computing and Communications*, 18(4):53–58, 2015.

[36] Benjamin Xie, Isra Shabir, and Hal Abelson. Measuring the usability and capability of app inventor to create mobile applications. *2015 ACM SIGPLAN Conference on Systems, Programming, Languages and Applications: Software for Humanity (SPLASH)*, 2015. doi: 10.1145/2824823.2824824.