SD Draw: A State Diagram Tool including Elm Code Generation for Interactive Applications

SD DRAW: A STATE DIAGRAM TOOL INCLUDING ELM CODE GENERATION FOR INTERACTIVE APPLICATIONS

BY

PADMA PASUPATHI, M.Sc .

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTING AND SOFTWARE

AND THE SCHOOL OF GRADUATE STUDIES

OF MCMASTER UNIVERSITY

IN PARTIAL FULFILMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

Master of Science

Master of Science (2021) McMaster University

(Computer Science) Hamilton, Ontario, Canada

TITLE:	SD Draw: A State Diagram Tool including Elm Code Generation for Interactive Applications
AUTHOR:	Padma Pasupathi M.Sc. (Computer Science) McMaster University, Hamilton, Ontario, Canada
SUPERVISOR:	Dr. Christopher Anand
NUMBER OF PAGES:	xiv, 57

Abbreviations

BMC Bramptom Multicultural Community Centre.

CAL Coding as Another Language.

CDF Cumulative Distribution Function.

CL Checklist.

CSE Computer Science Engineering.

DD Double Diamond.

DT Design Thinking.

EDP Event-Driven Programming.

ElmJr Elm Junior.

FIFO First in First out.

FSM Finite State Machine.

HCI Human-Computer Interaction.

HMW How Might We.

HTML Hypertext Markup Language.

IDE Integrated Development Environment.

IHP Integrated Haskell Platform.

K-12 from kindergarten to grade 12.

MDD Model Driven Development.

NFA Non-deterministic Finite State Automaton.

PAL Petri App Land.

PDF Probability Distribution Function.

RQ Research Question.

STEM Science, Technology, Engineering and Mathematics.

SVG Scalable Vector Graphics.

Dedication page

To my supervisor Dr. Christopher Anand and my husband Vijay.

I couldn't have this done with you both. Thank you for always supporting and believing me!

And most importantly, to the **God Almighty**!

Abstract

To make computational thinking appealing to young learners, initial programming instruction looks very different now than a decade ago, with increasing use of graphics and robots both real and virtual. After the first steps, children want to create interactive programs, and they need a model for this. State diagrams provide such a model, as observed previously by other researchers.

This thesis documents the design and implementation of a Model Driven Engineering tool, SD Draw, that allows even primary-aged children to draw and understand state diagrams, and create modifiable app templates. We have tested this with grade 4 and 5 students. In our initial test, we discovered that children very quickly understand the motivation and use of state diagrams using this tool, and will independently discover abstract states even if they are only taught to model using concrete states. To determine whether this approach is appropriate for children of this age we asked three questions: do children understand state diagrams, do they understand the role of reachability, and are they engaged by them. We found that they are able to translate between different representations of state diagram, strongly indicating that they do understand them. We found with confidence $p = 0.001$ that they do understand reachability by refuting the null hypothesis that they are creating diagrams randomly. And we found that they were engaged by the concept, with many students continuing to develop their diagrams on their own time after school and on the weekend.

Acknowledgements

Thank you to Ms. Celia Anand and Ms. Cohen, and especially her students at Saginaw Public School for their excellent feedback while creating and testing the SD-Draw application.
Thank you Chris, Nicole, Sarah for being a co-authors of the conference paper.
Thank you to the McMaster Computer Science Outreach members over the last several years who have worked, and continue to work, towards increasing student exposure and confidence in Computer Science topics.
I would also like to thank Department of Computing and Software for the opportunity and financial support.
Finally, last but not least, thank you to **Dr. Christopher Anand** for his patience, guidance, and confidence in my work over the last two years.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

McMaster Start Coding (http://outreach.mcmaster.ca) has introduced over 25,000 children in Grades 4 to 8 to computer science over the last 5 years. Over time, we have adopted programming in Elm and socially constructive learning that includes Design Thinking.

Design Thinking is a problem-solving process particularly suited to ill-defined problems involving people. This thesis is the report on a cycle of design thinking to improve our K-8 teaching. Following the Design Council's Double Diamond [Council, 2019], we must first define the right problem by observing users in their environment and interviewing them, as well as identifying best practices, competitors and lessons from the scientific literature. In place of direct observation and interview we interviewed instructors in our outreach program. They said, interaction is much harder for children to learn than animation, and the connection to the math curriculum is harder to explain to teachers, but children in Grades 4-8 whom we teach are highly motivated to learn interaction, and prior to COVID-19, we had some success using game maps (graphs). Consulting the literature, we found research on Event-Driven Programming (EDP), the importance of visualization while learning, the relationship between coding and other forms of literacy, advantages in using drawing as part of teaching and learning, and of team-based learning. This led to the *How might we?* (HMW) statement: **How might we better teach children in Grades 4 - 8 to design, implement, explain and modify programs with user interaction?**

Translating game maps into code was too difficult for beginning programmers, and hard for mentors to support. Nevertheless, there is no reasonable alternative mathematical structure for describing interaction. Inspired by the EDP and model-driven

development literature, we decided to make a better tool for state diagramming including code generation. The new tool adheres better to Norman's principles ([Norman, 2013]), generates complete programs (with working buttons for all transitions), and went through many iterations internally before we tried to use it with children from Grades 4 and 5. Most of our outreach activities are constrained, so our initial test was restricted to 3.5 hours per class of instruction, followed by one hour of challenges a week later. Surprisingly, we could answer affirmatively all of our questions about student understanding in this short intervention, including statistically significant results on their understanding of reachability, and the observation that they could both translate between different representations of state diagrams and spontaneously use abstract states states like "DragonIsDead" or "GameOver" versus concrete states like "Park" or "Mountain") in their diagrams.

1.1 Purpose:

As a part of our Outreach program, we provide a list of lessons that cover basic drawing, animations, interactions, comics, and adventure games using Elm programming language. State diagrams are taught as Lesson 8 in the syllabus, which introduces user interaction using states (state of the application) and transitions (responses to user actions). This helps children to visualize how a game is created with multiple interactions and changes in its state. Initially we taught these state diagrams on a (physical) white board, then we designed PALdraw for advanced developers who could use Petri nets, but children seemed to enjoy working with it as a simple Model Driven Development (MDD) tool. But mentors found teaching state diagrams using PALdraw awkward due to the complicated interface.
The main purpose of this app is to re-introduce the state diagram tool with a new user interface with familiar controls leveraging Norman's design principles for user interfaces. In addition to this, all the major operations like drawing, code generation are performed on the client end leaving less interaction with the server, and increasing scalability.

We also wanted to think about the place an improved tool could have in our outreach efforts, which can be summarized by the following research questions:

RQ1 Do grade 4-5 students demonstrate an understanding of State Diagrams by being able to translate between different representations?

RQ2 Do grade 4-5 students demonstrate equal facility for translating between different representations of state diagrams?

RQ3 Can grade 4-5 students understand the role of reachability? Assuming that students who did not understand the role of reachability would generate random graphs, what confidence do we have that the graphs are more reachable than random graphs?

RQ4 Are grade 4-5 students engaged by state diagrams and their applications to adventure games?

RQ5 Do grade 4-5 students understand abstract and concrete states equally well? Will students presented with concrete states generalize to abstract states without prompting?

1.2 Scope:

The scope of this thesis includes new UI/UX of the state diagram drawing tool with fewer buttons and keyboard interactions but adding mouse controls and drag and drop features. This version of the state diagram tool should also explore the presentation of associated data types.

1.3 Contribution:

Application contributions: The design and implementation of the Elm application is the work of the thesis candidate. The server back-end and serialization implementation is the work of undergraduate student Christopher W. Schankula.
Test: The lesson was taught and the tests were conducted by the thesis candidate, grad student Akshay A., undergrad students Christopher W. Schankula, Larry Yao and the thesis supervisor Christopher K. Anand.
Working Paper Contributions : A working paper was drafted including preliminary results of the study, written by the thesis candidate, Christopher W. Schankula, Nicole DiVincenzo, Sarah Coker and Christopher K. Anand. Many subsections from the working paper have been adopted for this thesis, such as Background (McMaster Start Coding Program, Functional Programming, Elm language, Related work, Event-driven programming in Education), State Diagrams (Formal Definition, Code

generation, Adding Graphics with Elm), Results of Evaluation (Qualitative Analysis, and Qualitative Analysis other than the reachability analysis), Discussion (Limitations, Pedagogical Improvements), State Diagram (Tool improvements).

(Sub)sections added for this thesis by the thesis candidate include Purpose, Scope, Background (NewYouthHack, Petri App Land, Lesson 8, Design Thinking, Elm Architecture, Algebraic Datatypes), Motivation, State Diagrams (State Diagram in General, State Diagram in SD Draw, Software Design, Norman's Principles, Self hosting, Recursive states, Nested states and User-Defined Algebraic Data types, Keyboard Shortcuts, Semantic versioning), Results of Evaluation (Reachability analysis), Discussion (Design Checklist), Conclusion.

Chapter 2

Background/ Literature Review

In this section we provide background of our program as well as relevant research on which we built our tool and instruction.

2.1 McMaster Start Coding Program

This McMaster University Outreach Program has been operating for the past decade. A mainly volunteer group of undergraduate and graduate students develop lesson plans and deliver free workshops to schools, public libraries, and community centres in the Hamilton, Ontario, Canada area [O'Farrell and Anand, 2017]. Before focusing on Elm programming, we used different approaches in teaching computer science fundamentals, including the development of iPad apps "Image 2 Bits" and "MacVenture" ([Brown, 2016]), which allowed children to create an adventure game by drawing a game map, which is in fact a state diagram, although described using different vocabulary ("places" and "ways" rather than "states" and "transitions").
During the COVID-19 pandemic, the program has shifted online and has taught a record number of students. Since 2016, we have taught over 25,000 students in nearly 1,000 classrooms. The goal of the program is to foster interest and ability in STEM subjects through coding, especially for those groups who are underrepresented in STEM subjects, such as girls and underprivileged youth.
To support these workshops, we have developed several tools, including:

1. An open-source Elm graphics library, GraphicSVG [Schankula and Anand,

2016-2019] which was recently updated in 2021.

2. An online mentorship and Elm compilation system incorporating massive collaborative programming tasks, including the Wordathon[1] and comic book storytelling[2].

3. A curriculum for introducing graphics programming designed to prepare children for algebra [d'Alves et al., 2018].

4. A type- and syntax-error-free projectional iPad Elm editor, ElmJr [Optimal Computational Algorithms, Inc., 2018].

In addition to local outreach, we have partnerships with post-secondary institutions in India and Iraq to establish similar outreach programs, such as Narasus Sarathy Institute of Technology, Vellore Institute of Technology, Cihan University.
To be successful in the long term, these programs must engage undergraduate students, who are interested in creating web applications. The long-term goal of this work is to create a tool which makes the specification and development of interactive web applications simple enough for children, but powerful enough to create the types of web applications of interest to potential undergraduate mentors.

2.1.1 NewYouthHack

In association with the Brampton Multicultural Community Centre (BMC) and Immigration, Refugees and Citizenship Canada, we built a project called NewYouthHack [Schankula et al., 2020] as shown in fig 2.1, which applied Design Thinking (DT) to reimagining settlement services in Canada. We also wanted to expose the youth to career pathways involving Software Design and Development. In the process, we built a technology platform which supports iterative development and the integration of novice programmers.
NewYouthHack started with a two-day hackathon program where participants were asked to use DT to improve the experience of settling in Canada. The hackathon activities were modelled as per the DT process, where they conducted interviews of each other, used How Might We (HMW) questions to frame a problem and iterated through their ideation process to refine a solution for the user. Since the youth involved were mostly high school students, they benefited by being able to apply DT

[1] http://outreach.mcmaster.ca/#wordathon2019
[2] http://outreach.mcmaster.ca/#comics2019

to their own learning. This is more of a designathon than a hackathon where each team came up with their own self-contained app ideas along with the prototypes for the ideas built during the coding workshops followed by the hackathon.

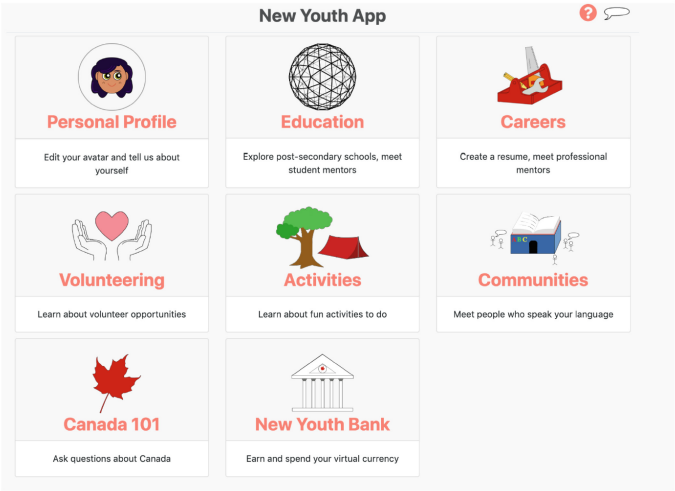


Figure 2.1: New Youth Hack application

At the coding workshop, the youth used their coding skills to code illustrations and animations that were incorporated into the app. For instance, students developed new features for an avatar creator, such as glasses and jewelry, and helped prototype the resume format, see fig 2.2. These coding sessions gave the students a means to directly apply their feedback on the app by coding their own contributions.

Figure 2.2: Customizable Avatar of a user and Resume template

Initially, the idea was to build multiple standalone applications as a result of practicing DT process. That could be achieved with skills from our existing Elm coding workshops (as discussed in the Elm architecture section). But most of their ideas required mentorship, sharing, communicating, and commenting on resumes which depends on client-server interaction. Thus we created a client-server framework called Petri App

Land (PAL) which is an expansion of normal state diagrams (which is discussed in next section).

2.1.2 Petri App Land

PAL [Schankula et al., 2020] is an open-source framework, which is the expanded version of state diagrams including the client server interaction. On the client-side PAL generates code which handles connecting to the server, encoding and decoding the outgoing and incoming messages respectively. On the server-side PAL uses a FIFO queue which is read in a state loop, processing the messages and updating the state automatically.

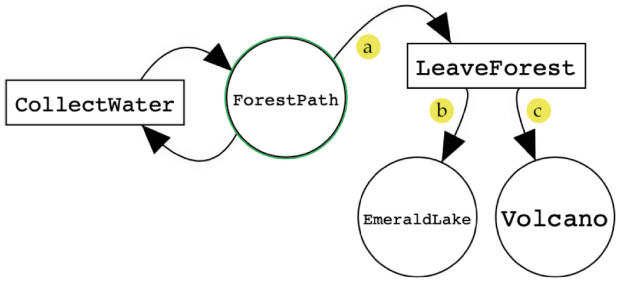


Figure 2.3: Example of a PAL using the PALDraw application.

Figure 2.3 shows a PAL drawn using the PALDraw application. Here, all circles represent places, the rectangles represent transitions and the directed arrows represent the relationship between the places and transitions. In this diagram, we have three places called **ForestPath, EmeraldLake, Volcano** and two transitions called **CollectWater, LeaveForest**. Here we do not have an obvious notation for the start place and upon LeaveForest, clients could be transitioned to EmeraldLake or Volcano places.
The Figure 2.4 shows the PALDraw application, with a State Diagram embedded in a place called Lesson8Demo. The graphical notation is similar to the notation for the overarching PAL, but transitions can only have single entry and exit tentacles. PALDraw has multiple buttons to create or delete states and transitions. Also, renaming or deleting a state or transition is accomplished using the text box on the right side of the screen after selecting the required state or transition. The Save button must be clicked to save the new name.

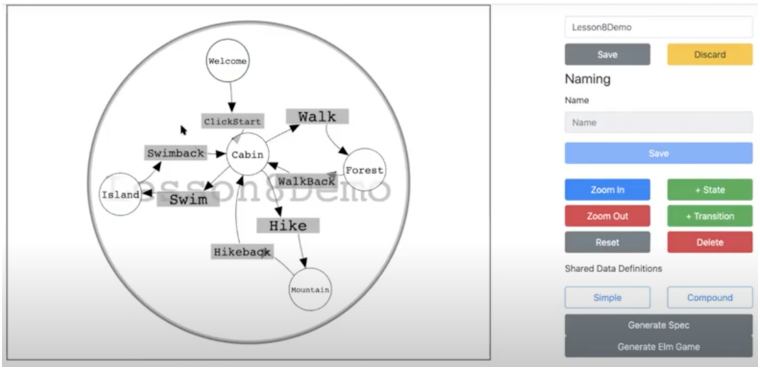


Figure 2.4: PALDraw application

2.1.3 Lesson 8: State Diagrams and Adventure Games

As a part of McMaster Start Coding club syllabus, we teach State diagrams and Adventure games in Lesson 8 (Youtube link) This lesson begins with explaining the animal sound game. Students are asked to make different animal noises based on number of arms raised by the mentor. A sample adventure game is explained with respect to a state diagram, comparing states and transitions of a diagram with a real adventure game. Then, introduction to PALDraw2.1.2 is provided to the students with a small example of moving to different places like Forest, Mountain etc. In this example, physical places like Forest, Mountain, Cabin, Island etc are considered as states and verbs like Walk, Swim, Hike etc are considered as transitions. After explaining this example, students were explained how Elm code can be generated from the PALDraw application and can be compiled in macoutreach.rocks website. Finally students were asked to add graphics to each place based on their previous lessons.

2.1.4 Design Thinking

Design Thinking (DT) [Council, 2019] is a human-centred methodology that focuses on the end-user and iterates rapidly through conceptual prototyping to produce innovative and creative solutions to complex problems. The DT process is designed to avoid the creation of technically perfect but unwanted or incomprehensible products by focusing on the end-user while creating the product, while considering three dimensions: technical feasibility, economic viability, and desirability to the user.
DT assumes that the end-user is complex and that an understanding of their needs requires experiment and inquiry. Working with end-users is not a validation process.

It is a discovery process. Hypotheses are not formulated as precisely, and are not about natural phenomena, but about the user's needs and experience.

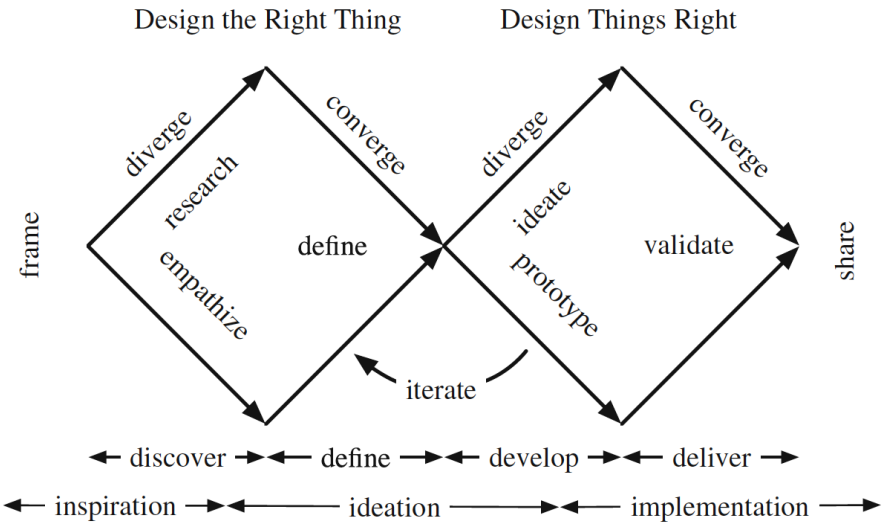


Figure 2.5: Double Diamond with the key process of Design Thinking.

DT can be viewed as double diamond process model developed at the British Design Council in 2005, see Fig.2.5. There are divergent thinking stages followed by convergent stages where ideas are narrowed down towards the best one. When designing, some people ignore the left side of the diamond, which leads them to focus on solving the wrong problem. This is why, in DT, discovering the problem through empathizing and research, as well as defining the right problem, are integral to the process. The develop stage involves developing prototype, testing, and iterating. Finally, the deliver stage is when the product is finalized, produced, and launched

2.2 Functional Programming

Functional programming [Krishnamurthi and Fisler, 2019] is a value-oriented programming paradigm, consisting of functions. Functions consume and produce values. There are no loops, and conditional expressions replace conditional statements, but functions are first-class values and can, e.g., be passed as parameters. There are two variations in functional programming languages: (1) typed or not and (2) eager or lazy. These variations lead to differences in programming style.

Many non-functional programming languages are adopting functional features, including Scala, Swift and Python.
Krishnamurthi and Fisler agrees with the common perception that writing programs in imperative programming languages is much easier as the state provides convenient communication channels between parts of a program, but this makes reasoning and debugging harder, whereas on the other hand functional programming has the opposite affordances. Students studying object-oriented programming are taught different skills and programming styles which reveal that the way of approaching programming and problem-solving differs in students studying different paradigms. Functional programming students perform better[Krishnamurthi and Fisler, 2019] by having high level structures and and composing solutions out of simpler functions than object-oriented students who try solving the entire problem in a single traversal of data. Students who learned Funtional Programming first also use built-in/higher order functions to implement subtasks which performed multiple passes over input data and had to release unwanted memory for intermediate data. Functional programming students create short functions for specific tasks, which create intermediate data. They also use `filter` and `map` rather than loops and non-general library functions. Thus, Krishnamurthi and Fisler say that a student who learns Java after learning functional programming may well program with different patterns than a student whose prior experience was entirely imperative.
Note that our experience is that functional programming with appropriate supports is easier for Grade 4 to 8 students. Before using Elm, our Outreach program taught using Python, Alice and Haskell. With Python and Haskell, students initial programming could easily trigger errors related to features of the languages they did not actually need causing frustration. With Alice, programming looks nothing like the mathematics they are learning, so although they enjoyed aspects of it, teachers could not see connections with other curricula. We also had trouble with bugs in the Alice environment, working around which required deep understanding of object oriented programming. Based on these experiences, and initial success with Elm, we designed GraphicSVG [Schankula and Anand, 2016-2019], a library for vector graphics which surfaces the concepts they are learning in math (coordinates, transformations, functions).

2.3 Elm Language

Elm (https://elm-lang.org) is a functional language designed for the development of front-end web applications [Czaplicki, 2012], and sold to front-end developers as a way of avoiding the many software quality issues which plague JavaScript programs. Its syntax, based on Haskell, is intentionally simple. For example, it has no support for user-defined type classes. In addition to strictly enforcing types, the Elm compiler also forces programmers to follow best practices, such as disallowing incomplete case coverage in case expressions. Elm apps use a model-view-update pattern in which users write pure functions and the run-time system handles side effects without the need for advanced concepts. Elm code compiles to JavaScript simplifying deployment and visualization.

While many consider that functional programming should be reserved for expert users, many of the features useful for experts (strict types, pure functions) are very useful for beginners. In addition to the practical implications of compiling to JavaScript, Elm's combination of simple syntax, strict typing, and purity which matches students' pre-existing intuition about math proves to be an asset to our program [O'Farrell and Anand, 2017]. These features allow the development of tools and curricula which would not otherwise be easy or possible in an imperative language with side effects such as Python.

2.4 Elm Architecture

All Elm programs follow a common architecture, "The Elm Architecture", with different variations enabling or restricting certain features as they are needed. These built-in "app" types interact with Elm's JavaScript-based runtime system, enabling pure code to interact with the outside world in a predictable manner, without runtime errors.

Elm's overall architecture consists of three main components: the *model*, the *view*, and the *update*.

The *model* of the program holds all states of the application. Here, state can a place, page (like pages in a website), level (like multiple levels of a game) or could even be a value (like score or temperature that keeps updating in an app) that changes upon interaction. A model could have one or more of all the above types of state. Each of them can have the same or different data types. The data types can be simple like `Int` or `String` or can be a complex data types. For example, a game that has three

levels could model the levels as follows,

```
type Level = Level1
           | Level2
           | Level3
```

The next component of the Elm architecture is the *View*. The View is a function that is used to visualize the model of the application. This function is used to display the current state of the application. View function (`view : Model -> Html Msg`) renders an HTML view of the application state and includes the mapping of user-generated events to messages used by the update function.

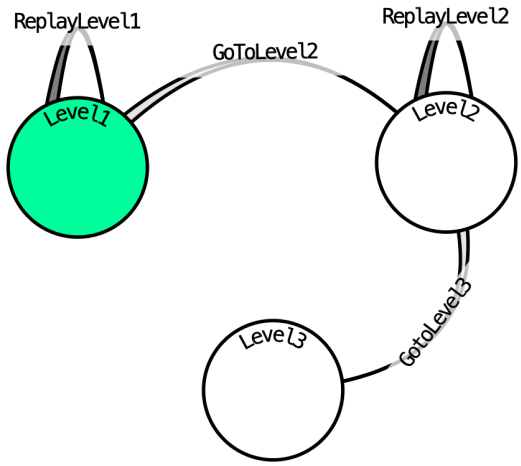


Figure 2.6: The interactions in an Elm app can be encoded as a state diagram, with circles being the states in the app and the transitions being the arcs between them, named by the message sent to initiate the transition. When describing an Elm app this way, the programmer has to take care to follow the state diagram in the logic of the program as it is not enforceable at the type level.

The final component of the Elm architecture is the *update* function, `update : Msg -> Model -> Model`, which is used to change the current state of application to a different state of application based on the message received from the run-time system based on either interaction or a system event (like receiving a network packet). Here the message `Msg` type for the game example could be as follows,

```
type Msg = ReplayLevel1
         | GoToLevel2
         | ReplayLevel2
         | GoToLevel3
```

From the above type of `Model` and `Msg` we can derive the state diagram, where states represent the model and transitions the messages that trigger the change from one state to another. Figure 2.6 shows how the corresponding state diagram would look. From the above diagram 2.6, it is clear that the game starts with Level1. User can either choose to replay Level1 or can go to Level2. Similarly in Level2, users can keep playing Level2 or can proceed to Level3. Level3 is the final state because there are no transitions emerging from the Level3. So, the components of the Elm architecture are sufficient to create state diagrams and with that state diagram, we can easily understand the flow of the application at a high level without looking deep into the code.

2.5 Algebraic Datatypes

The Elm language has algebraic data types. Algebraic Data Types are data types created using other data types. For example, a `List` is an algebraic datatype, a list can be a list of integers, list of strings or could be a list of any other data type. There are two types of algebraic data types, (1) Product types, and (2) Sum types.
A **Product** type value contains several values called fields and each value will have a combination of field types. It is the Cartesian product of the possible values and its field types. For example, consider a tuple of boolean values, here the boolean type has two possiblities `true or false`. As it is a tuple, it is a combination of these two values. So the values would be, $2 \times 2 = 4$, namely

$$\{\texttt{(true,true)},\texttt{(true,false)},\texttt{(false,true)},\texttt{(false,false)}\}.$$

As we are computing the values by multiplying, we call this a Product type.
A **Sum** type is often used in Functional programming, but is present in older languages as enumerated and union types. These are the types where its values must be one of a fixed set of options. For example, a value of type `Boolean` is either `True` or `False`.

```
type Boolean = True
             | False
```

So, the $(1 + 1 = 2)$ values would be $\{\texttt{True}, \texttt{False}\}$.
Here the values can only be one of the fixed options provided for this type and we can count the possibilities by adding them up, so we call this type as Sum type.

2.6 Related Work

Our background research aimed to identify prior work in the area of using state diagrams to teach computer science to K-12 students, as well as a more sweeping review of coding education and how it relates to other types of literacies.

2.6.1 Coding, Literacy, and State Diagrams

When it comes to child development and coding, looking at coding as a language is heavily emphasized in the literature. Again using Piaget's work, many believe that coding can change the way we think and experience the world around us [Bers, 2019]. Coding in itself is seen as a language [Bers, 2019]. Goldenberg and Carter believe that computer programming is just as important as English and should be taught in elementary school. Monteiro et al. also suggest that "programming can be the third language that both reduces barriers and provides the missing expressive and creative capabilities children need." Coding is a mix of English and math as the words allow for interaction with feedback [Goldenberg and Carter, 2021]. These two authors also bring up important facts when looking at programming as a language: "students can construct viable arguments and critique the reasoning with others, it eases the process of beginning with concrete examples and abstracting regularity, perseverance, using the proper tools strategically, and being precise." Finally, and akin to English, coding also involves problem-solving, a manipulation of a language, and symbols to create a shareable product [Goldenberg and Carter, 2021].

As discussed previously by Bers, previous systems for teaching coding based on Science, Technology, Engineering and Math (STEM) did not account for the intellectual maturation of school-age children. "Coding as Another Language (CAL)" considers coding development alongside language and literacy development. Using similar stages to those of learning reading and writing (emergent, coding and decoding words, fluency, new knowledge, multiple perspective, purposefulness), CAL uses literacy as both a parallel to develop programming curricula and a tool. Knowledge-constructing concept maps can allow for mental mapping of written stories and allow for a newer and easier method for students to record their ideas before starting the writing process [Anderson-Inman and Horney, 1996]. State diagrams mimic mind-mapping, a commonly used method of brainstorming in literacy teaching. Previous studies and curriculum formed around these standards have shown that it is very possible for students to have a basic understanding of coding upon leaving elementary school [Vico et al., 2019], similar to their level of reading and writing when entering high school.

Recent studies show that lessons in coding can also be useful in teaching mathematics at the elementary level [Suters and Suters, 2020]. By teaching computational thinking, or the thinking of a computer scientist, at a young age, students are provided with a deeper understanding of mathematical relationships necessary to perform algebra and calculus in later grades. All the above mentioned points align with our motto to get coding literacy to school students.

2.6.2 Visual Learning and Education

Our initial experience teaching basic programming using vector graphics in Elm shows strong student engagement. We expect that children will always be engaged with colourful outputs, but some of their engagement could be explained by the way that connecting textual and visual representations makes learning easier by using multiple aspects of working memory in parallel. We wanted to know what past research says about the use of visual information in facilitating learning in general, because this might help us understand our past success and guide our development of visual representations of state diagrams. Based on the keywords used, there seemed to be more literature on learning through drawing than learning through writing in relation to STEM. In the study done by Ainsworth and Scheiter, they were able to list advantages of drawing: i) limits abstraction ii) exploits "perceptual processes by grouping relevant information" iii) draws on problem-solving instead of memory, iv) provides focus for joint attention/group collaboration, v) increases attention, and vi) activates prior knowledge [Chang, 2012]. Park et al. also states that learning through drawings not only takes different perspectives into consideration but also exposes the child to other subject domains (such as math and literacy) when working in groups. To add, those who used learning-by-drawing scored higher on a test based on comprehension [Schmeck et al., 2014]. Chang found that when a child and an adult are partners in learning through drawing, that communication was associated with healthy language development and enabled the children to listen, think, and then speak. Interestingly, Cheng and Beal found that while students who drew had a significantly higher cognitive load than those who studied pictures, students were more willing to learn with provided pictures than drawing themselves. That being said, Kunze and Cromley also noted that drawing-to-learn was slightly less effective in "early secondary (i.e., children who are around 12-15) than in the bulk of literature."

2.6.3 Event-Driving Programming in Education

Before considering the visual representation of state diagrams, we wanted to understand the use of state diagrams explicitly or implicitly in previous teaching approaches. In the literature, this teaching approach is generically referred to as Event Driven Programming (EDP). As Lukkarinen et al. state in their literature on EDP in programming education, event-driven programming and computer programming are two separate entities; programming relies on *organizational characteristics* whereas EDP focuses on *behavioural characteristics*. For example, while computer programming is more procedural and object orientated, EDP forces the programmer to consider the consequences of the user's actions and how to react to them [Lukkarinen et al., 2021]. That being said, the main takeaway is that EDP and computer programming are two different concepts and therefore require two separate ways of teaching. On a similar note, Lukkarinen et al. also display the challenges of EDP within their literature review: it is hard to trace the computer programming from beginning to end which affects students abilities to fully understand, EDP has been linked to negative transfer effects when associated with event and non-event oriented programming environments, and "students who learn to program in an event-fashion do not develop some algorithmic skills that other students will have" [Lukkarinen et al., 2021]. Finally, in terms of challenges, Lukkarinen et al. state that no attempt has been made to alleviate these issues within EDP and learning.
Finally, in this literature review, we learned the most used software tools when teaching and learning EDP. Lukkarinen et al. found that Java was the most popular language with App Inventor, C++, and Scratch. Other tools for Java include DoodlePad and Squint Library [Lukkarinen et al., 2021].
Lukkarinen et al. questioned if any empirical results were recorded, to which they concluded that no pedagogical method (i.e., ways we teach EDP through abstract concepts or through video games for example). However, Lukkarinen et al. cautions researchers to view EDP and learning as more than "merely claiming in passing that some method or tool caused students to learn and giving some counts as statistics."

2.6.4 State Diagrams in Computer Science Education

Another way of arriving at our use of State Diagrams is via Model Driven Engineering or Development (MDE and MDD). In MDE and MDD, mathematical structures, such as state diagrams are used to model application behaviour, and code is generated from these structures, rather than just being used as documentation. This is

our goal for SDDraw. Often these mathematical structures have natural graphical representations, which ties back to visual learning.

Several authors have used state diagrams to introduce computer science and coding concepts at both the K-12 and university level. Czejdo and Bhattacharya discusses lessons using state diagrams to allow students to describe complex behaviours for robots, which then generated Python code to control the robots. They noted that the robots increased the students' engagement with the concepts and that the diagrams allowed the students to program more complex behaviours than would have been possible without them.

Kamada similarly used state diagrams to program behaviours of on-screen characters, for example programming a virtual fire truck to seek out water and then put out fires. They noted that "Enthusiastic children often run into the combinatorial explosion of states and transitions. Then it is the time for them to move on to the structured programming languages where they can use variables to represent states," which is aligned with our experience and is discussed in later in this paper. They also noted that "In some disappointing cases, a series of states are simply chained as if they continue forever" [Kamada, 2016], which is not something we observed in our albeit small study. This, however, is continued with the following statement: "We had better not recommend computer science to those children," with which we fundamentally disagree. We believe that this should instead be considered a teachable moment for the students and an area of improvement for the delivered lesson, instead of jumping to the rash decision that this indicates a fundamental lack of ability to be a computer scientist.

In contrast to the statement by Kamada, [Ben-Ari, 1998] instead states that "The science-teaching literature shows that performance is no indication of understanding. CSE research like Madison's, which elicits the internal structures of the student, is far more helpful than research that measures performance alone and then draws conclusions on the success of a technique. A student's failure to construct a viable model is a failure of the educational process, even if the failure is not immediately apparent." Thus, in our evaluations we must keep in mind that failures for students to apply state diagrams should be considered as important lessons for us as researchers in how to improve our lessons in the future.

Chapter 3

Motivation

In this section, we discuss the motivation for SD Draw. As discussed in the previous section, we had previously used State Diagrams embedded in coloured Petri Nets to specify interaction in client-server applications, and while the framework is open-sourced, the experimental visual editor was not. We soon discovered that children liked creating state diagrams despite the problems with the original interface. Our first motivation was simply to rewrite the State Diagram visual editor to simplify the user interaction. This is summarized by this checklist:

CL1 keyboard/mouse based editing, with a minimal number of buttons

CL2 in-place editing of names for places and transitions (live editing)

CL3 drag-and-drop where possible

CL4 minimize clicking

CL5 leave space for data in states and transitions

CL6 experiment with data types

CL7 snappy app (immediate feedback)

CL8 Elm code generation for buttons

CL9 Validate state and transition names. No two states or transitions can have the same name.

CL10 Widget and Server implementation

After strictly following these guidelines, we found that some buttons beyond the minimum were desired for Chromebook users who could not hold a key and move the mouse at the same time, and users who would forget keyboard+mouse actions. We realized that allowing, and even supporting, transitions with the same name better matched the Elm architecture. But overall, the checklist served us well.

Chapter 4

State Diagrams

4.1 State Diagrams in General

State diagrams are used to specify the behaviour of a system. For example, the state of a light bulb can either be ON or OFF, as shown in Figure 4.1. The state depends on the history of previous events or actions[1]. If the light switch is turned on, then the current state of the light bulb is ON, if the switch is turned off, then the current state of the bulb is OFF. These events are known as transitions. State diagrams can be used to represent finite state machines.

The components of a basic state diagram are listed in Table 4.1.

4.2 State Diagram in SD Draw

The state diagrams in this work are a type of deterministic finite state machine (FSM), which contains a set of states, transition labels and a partial function mapping a state and transition label to another state. Recall that a partial function is not

[1]The state of an Elm program, or any program for that matter, can be viewed as a fold which starts with an initial state and continually updates it according to a series of actions. Elm's architecture makes this especially easy to visualize. The series of actions is some arbitrary sequence of the aforementioned `Msg` type, and the function used to fold is the `update` function itself. For Elm programs without external commands, for a given starting state, `update` function and series of messages, the resulting state will always be identical.

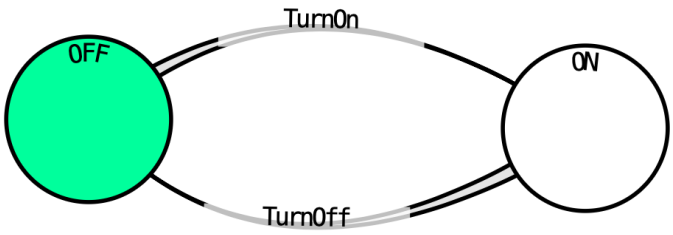


Figure 4.1: State diagram for a light bulb. Note that the transition functions are partial.

defined on its complete domain. In our case, if a transition function is not defined for a (state,transition) pair, it means that the view generated for that state will not have a button attached to that transition. Only if the user explicitly draws a transition, including a self-loop, will a button be generated. Also, unlike some definitions of FSMs, our diagrams do not have final states as they represent programs which continue to respond to user input until closed, rather than language recognizers. That being said, in the model-view-update pattern, the presence of buttons is determined by the view, but the update implements the transition functions, and before generating the update function, we extend the transition partial functions to total functions using the identity function. This is required by the Elm compiler, which does not allow case expressions to represent partial functions.

4.3 Formal Definition

Formally, our FSM is a 4-tuple, $N = (Q, \Sigma, \Delta, S)$ where

Q	is a finite set of state names
Σ	is a finite set of transition names
$\Delta : Q \times \Sigma \to Q$	is the transition partial function
$S : Q$	is the starting state
$Q \cap \Sigma = \emptyset$	

State	A state in a state diagram is used to represent the current state of the application, system or device. In SD Draw application, states are represented by circles.
Transition	A transition in a state diagram is used to represent an allowed change from one state to another. In a programming language, it is often implemented by a function which updates the state. In the case of human-computer interaction, these transitions may be triggered by a user interaction like a button click, moving the mouse to certain position and so on. Usually it is represented as directed arrows, in SD Draw application the line goes without an arrow head but its origin has thick line and it tapers towards its destination which encodes its direction.
Initial state	This state is the starting state of the state diagrams, which means the app would always start with this state. It can be represented in different ways; we represent it using green fill colour.
Final state	A state diagram may or may not have a final state. If the state diagram leads to a state which doesn't have any transitions to other states, then this could be called as Trap state and serves similar purpose of a final state In SD draw application, we do not have an explicit final state. If required, a state could be left with no originating transitions to simulate final state but it has no separate representation. The lack of final state represents how Elm programs run indefinitely without halting until they are closed by the user. Have a final state would serve no purpose other than to allow users to mark which of their states they consider final or "end-of-game" states.

Table 4.1: Components of a State Diagram.

As part of the code generation process, this FSM is completed by extending the partial transition functions by the identity, matching the usual definition from Kozen [2012], except that we do not label final states, since our diagrams represent infinitely running programs. Having trap states with no arrows out is one way of simulating a "final state" in the eventual game or application being modeled. Note that this

definition does not include associated data, the incorporation of which we leave to future work.

4.4 Graphical Representation and Tool

A state diagram can be represented graphically using a diagram where states are defined as labeled circles with transitions drawn as labeled arrows which define legal movements from state to state. Figure 4.2 shows an example state diagram representing the navigation of a school.

4.4.1 Software Design

To facilitate the creation of state diagrams, we have created a tool using the Elm language with a server backend written using IHP[2] in Haskell, which currently allows diagrams to be saved to a server and accessed later by logging in. Our state diagram tool allows students to easily draw their state diagrams by defining states and then attaching them with transitions. Each state and transition can be given a name where no pair of states or state and transitions can be named the same. Transitions can be named the same provided that no state has two transitions with the same name coming from it. Other invariants needed later for code generation are enforced, such as restricting state names to include only alphanumeric characters starting with a capital letter.
Figure 4.2 shows the SD Draw app's user interface. Here the right side of the screen has most of the controls of the app. (A) To add a new state, the first icon from the list should be clicked and the new state can be dragged to a desired position on the screen. By default the first state created will be the start state or initial state of the state diagram; if use wishes to change the start state, the second icon in the list can be dragged into the desired state to make it the start state. Only one state can be a start state in a state diagram.

(B) The upper icon in this section is used to Recenter, where it resets the position and zoomed in or zoomed out size. The next two icons are used to Zoom in and Zoom out the state diagram on the screen. One can either use these buttons or can use "Shift + Drag" on the screen to zoom in and zoom out. The icon with question mark is used to display the instruction page.

[2]https://github.com/digitallyinduced/ihp

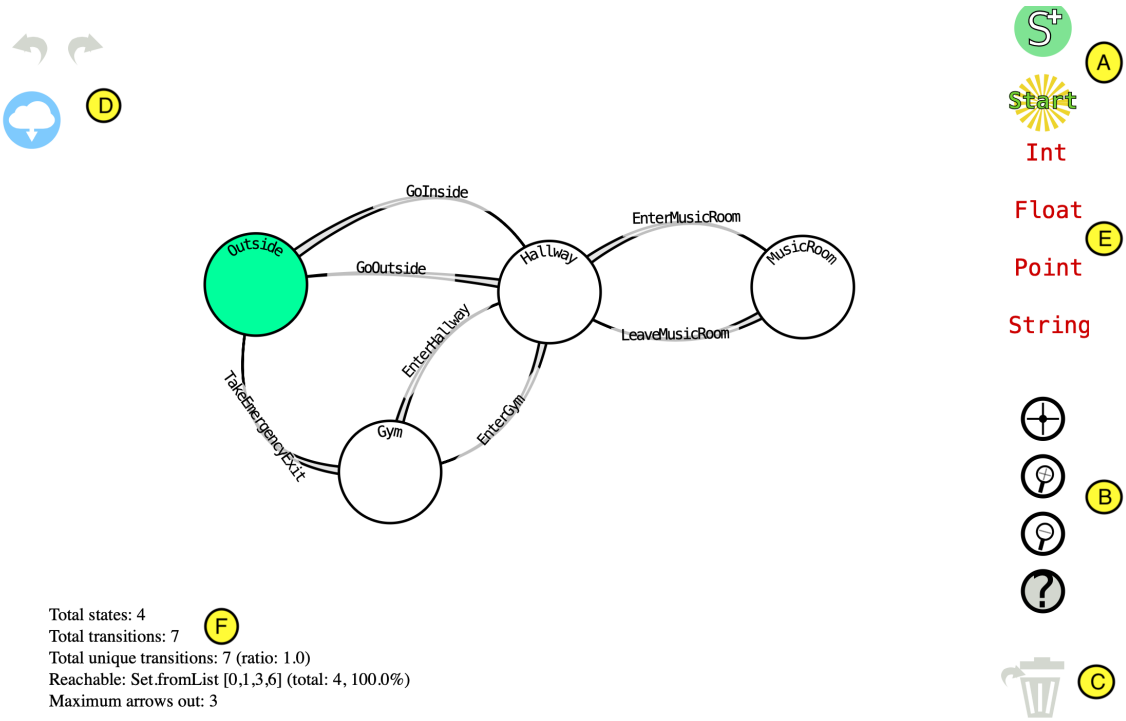


Figure 4.2: The interface of our web-based state diagram editor, with a diagram representing navigation through a school. (A) allows users to add states by dragging the "S" onto the canvas, or make a state the starting state by dragging the "Start". (B) has functions for recentering the screen, zooming and a help page. (C) allows states and transitions to be deleted. (D) provides undo / redo and code generation functionality. (E) shows representative data types which can be dragged to states and transitions, which we have included to get early feedback before completing the design, but have not explained in our teaching. (F) provides statistics of the state diagram including reachability. In the diagram, states are represented by circles with arrows (going from wide to narrow) representing transitions from one state to another. The green state is the starting state of the diagram.

(C) The last icon on the right is the trash bin which is used to delete states and transitions. If a user wishes to delete a state or a transition, they may drag it to this bin. When the item reaches the bin, the bin opens and turns into red color indicating it is ready to delete. Once the mouse lets go in the open trash bin, the dragged item will be deleted. A state or transition can also be deleted using the keyboard shortcuts of pressing and holding "Shift + Delete" keys and clicking on a state or transition to be deleted. This is difficult for children using Chrome-books. While deleting a transition, only that transition will be deleted from the list of transitions, but when a state is deleted, all the transitions associated with it will also be deleted along with the state.

(D) There are three icons on the left. The first icon is to perform Undo and Redo operations. In this app, Undo and Redo operations are implemented by keeping a list of undo and redo snapshots of the state of the program. Whenever a change is made on the diagram, the current state and transition will be inserted into the undo

list as a new entry. When this undo button is clicked, the most recently added item will be taken out from the undo list and the current state will be added to the redo list. If a new change is made, then the redo list will be made empty and usual undo list item will be added. The next blue icon on the left is used to generate the Elm code for the state diagram. When we click this icon for the first time, the code would be displayed on the screen. If the user wants to copy the entire code, then user could use "Ctrl+A" to select all and "Ctrl+C" to copy the selected code. Otherwise, user can click on this icon again to download the code as a text file. This generated code can be complied on the game slot of macoutreach.rocks website.

(E) The data types `Int`, `Float`, `Point`, `String` are the data types available in this version to add to states and transitions. Here, `Point` is the type alias for `(Float, Float)` to represent a position. Each state or transition has a list of associated (possibly repeated) data types. They are optional but adding data types to states or transitions can greatly increase the conciseness of the state diagram, by avoiding state explosion. For example, consider an example of a game which has different levels and the points earned in the previous level can be carried over to the next level. Here we can have an associated `Int` in the levels to indicate the score. We have not incorporated associated data types into our initial lessons, but included them so we can start having discussions with users to understand how they interpret them.

(F) We are currently displaying statistics at the bottom left of the screen including the number of states and transitions created, the reachability of all states using the available transitions,and the maximum number of transitions originating from a state. This information was initially used for debugging purposes and then to aid in test reporting, but when we worked with kids, this helped some children when their states were missing (moved off screen). A future iteration of an experiment with this tool should aim to see if the students can use the statistics to improve their diagrams in other ways, for example, identifying unreachable states which indicate that they have inaccessible parts of their app or game.

When a state is selected, the state will be highlighted in blue to show that it is selected and a small arrow will be displayed next to it. This little arrow is used to create the transitions from one state to another. When this arrow is clicked, a curve will start appearing on the screen and move as the mouse pointer moves. When this curve points to a state, the state will be highlighted in blue to indicate that this state would be the destination of the transition. A transition can be between two different states or can be a self transition (origin and destination of the transition would be the same state). In SD Draw, a transition is indicated using a curve whose originating end would be thicker and the destination end would be thinner. This design of curve was

chosen over conventional directed arrows because when the complexity of the diagram increases with a large number of states and transitions and the diagram is completely zoomed out, the direction of the transition will still be visible when the arrowhead is too small to see. We can also create a new transition using keyboard shortcuts, by holding the "Shift" key, clicking a source state and dragging. This creates the curve which looks for the destination state, then release over the destination state and let go the "Shift" key to create a transition between two states.
Each state and transition will be created with a default name appended with a number as a count. Every name can be renamed by simply selecting the state or transition. When it is selected, a blue cursor will appear at the end of its name, this indicates that the name can be edited by typing. We can delete the letters by pressing backspace and can type the new desired name. To make the eventual generated code compatible with Elm, there are few restrictions for the naming conventions as follows:

- Names can have only letters and numbers, no special characters or spaces are allowed
- Every name must start with an uppercase letter
- There are no character limitations for state and transition names but state names are drawn along the inside of the circle. It is recommended to use a maximum of 30 characters for a state to cover a full circle and 15 or fewer characters to have a better visibility of state names. Similarly, for a transition name it is better to have a verb that transitions between the two states on the transition.
- No two states can have the same name.
- States and transitions cannot have the same name.
- No two transitions originating from the same state can have the same name. If the user renames a transition from a state to an existing transition name from the same state, then the old transition will be deleted and the edited transition will be updated with the new name. So it is advised not to attempt to create two transitions with the same name starting from a state.
- Transitions from different states can have the same name, but if multiple transitions have the same name, when one transition is edited (like renaming or adding data types) all other transitions with same name will be similarly modified. This may be surprising to users at first, but was a design decision chosen

to make renaming groups of transitions easier. In order to "detach" a transition from the other ones named the same, the user must delete the transition and re-add it.

4.4.2 Norman's Principles

As per Don Norman's Design of Everyday Things, every app should follow the following design principles (Table:4.2) to improve user interaction and user experience. These principles were assessed by observing mentors using iterations of the app, resulting in many small improvements.

Visibility	The user should know the options available by simply looking at the application.
Feedback	Every action performed in the application should generate a visible response.
Signifier	It is the relationship between how something looks and how it is used.
Mapping	It is the spatial or temporal relationship between the control and its effect.
Constraints	These are the limits to an interaction or interface of the application.
Consistency	In an application, the same action should cause same reaction every time.

Table 4.2: Norman's Principles.

4.4.2.1 Visibility

In SD Draw, there are a minimum number of buttons and user can understand each of its functionality by clicking or dragging them like new state icon, Download button, Undo and Redo buttons, Zoom in and Zoom out buttons, etc. The visibility of these buttons are shown in the Figure 4.3.

4.4.2.2 Feedback

In SD Draw, every time we rename a state or transition, if the key pressed is in the allowed list of keys then it will be displayed on the screen immediately. There is no

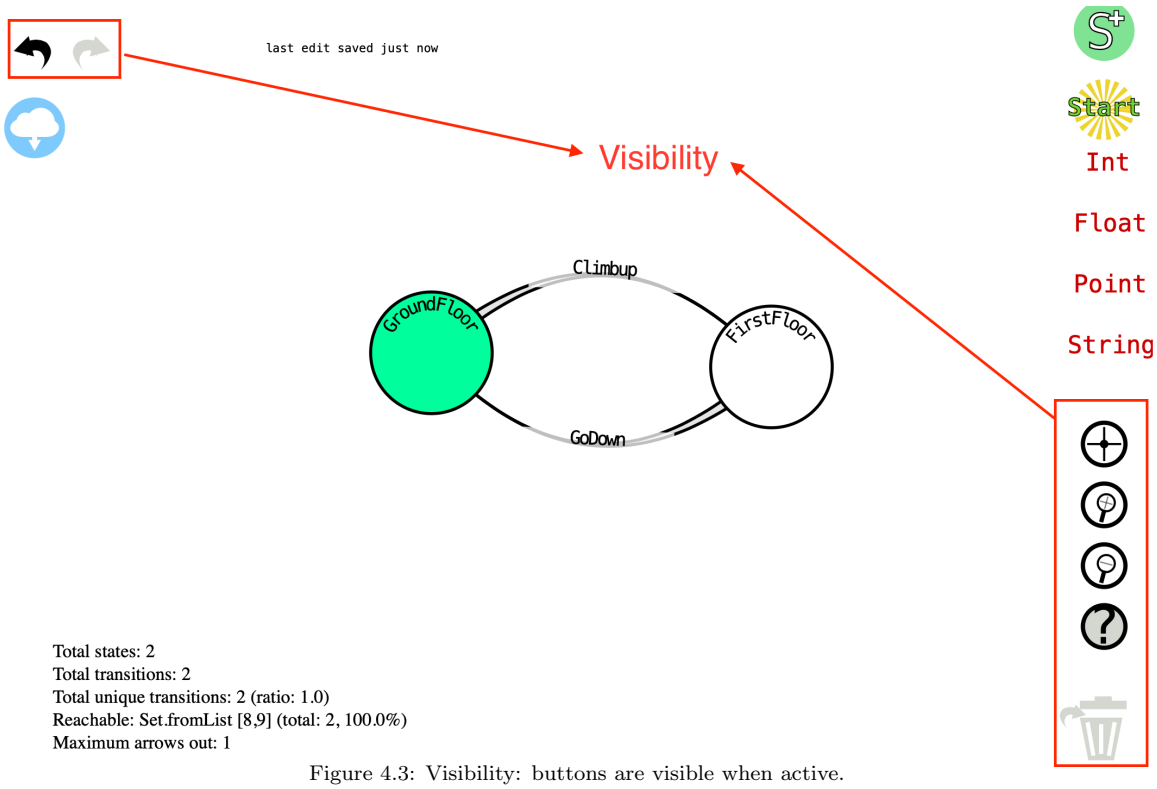


Figure 4.3: Visibility: buttons are visible when active.

need to have a separate save button to save and display like we had on the PALDraw application as seen in 2.1.2. Similarly whenever any other buttons or controls are clicked or dragged, it gives an appropriate response immediately. Similarly in the statistics section, we can see the number of states and transitions is updated upon creating or deleting them from the state diagram, which is represented in Figure 4.4.

4.4.2.3 Signifier

As mentioned above, we have a very limited number of icons whose display indicates their purpose. For example, a small arrow next to a state appears when a state is selected, as shown in Figure 4.5. It indicates that a transition can be create, originating from that state. And when the arrow is clicked, a curve is drawn out, which sticks to the mouse pointer and moves around the screen along with the mouse pointer as shown in Figure 4.6. This will create a transition if the curve is dragged to any state on the screen.

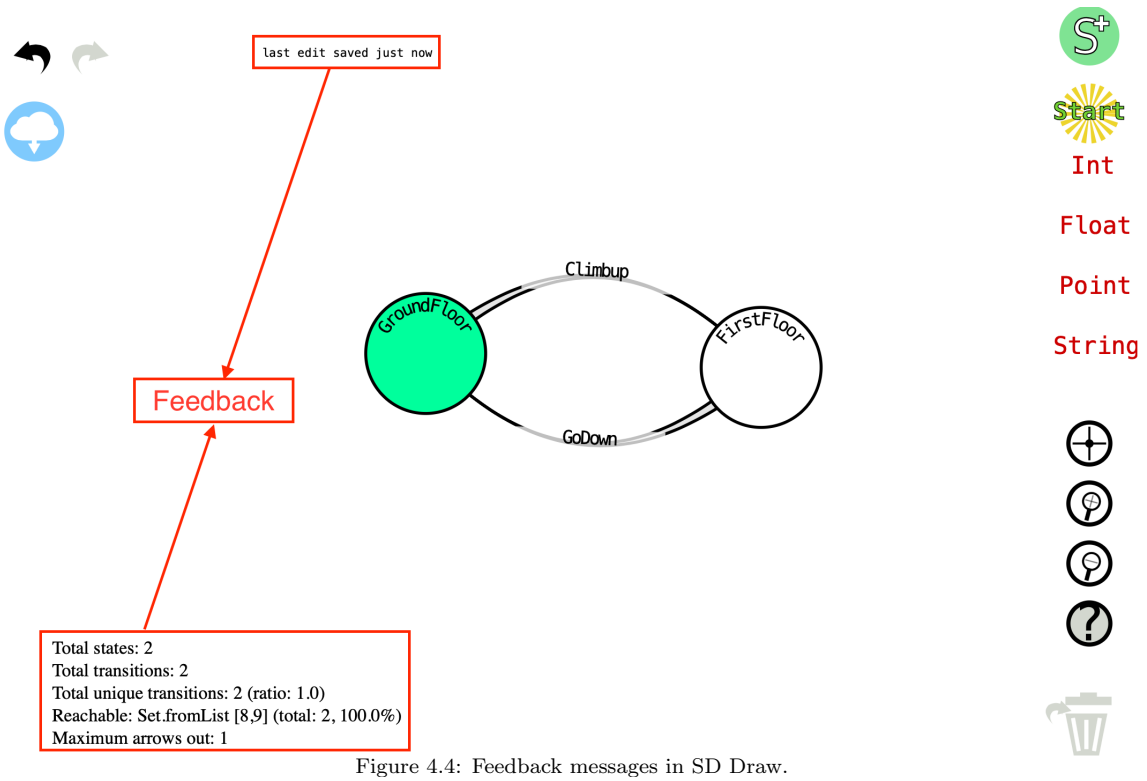


Figure 4.4: Feedback messages in SD Draw.

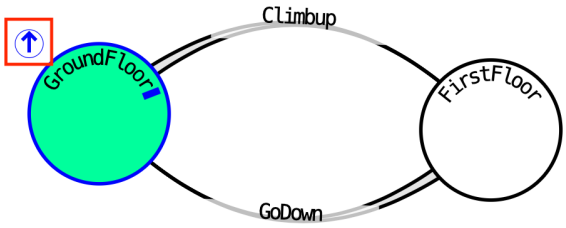


Figure 4.5: Signifier of transition start.

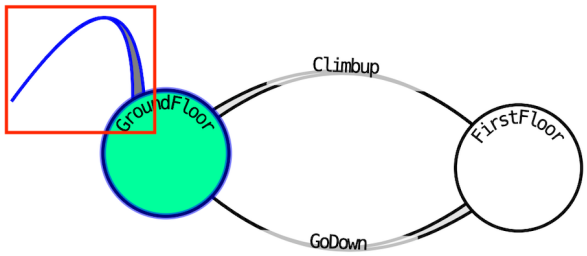


Figure 4.6: Signifier of transition creation.

4.4.2.4 Mapping

Mapping is the spatial or temporal relationship between the control and its effect. In SD Draw, the canvas mimics a physical plane, making it easy for children to learn. Here when we click and drag a state or a transition, it moves to the position where we move our mouse pointer on the screen. Similarly, when we select the start state button or any of the datatype listed, it sticks to the mouse pointer and it is moved across the screen as we move the mouse pointer until we drop them in a state or transition. See Figure 4.8 where the datatype is dragged to the screen as mentioned as start state button. In addition, when a state or transition is dragged to the trash bin's position, the bin opens and it turns to red in color indicating that it can be deleted, as shown in Figure 4.7. The dragged state or transition will be deleted only upon letting go of the mouse point when the bin is open, otherwise it would just treated as a regular dragging action.

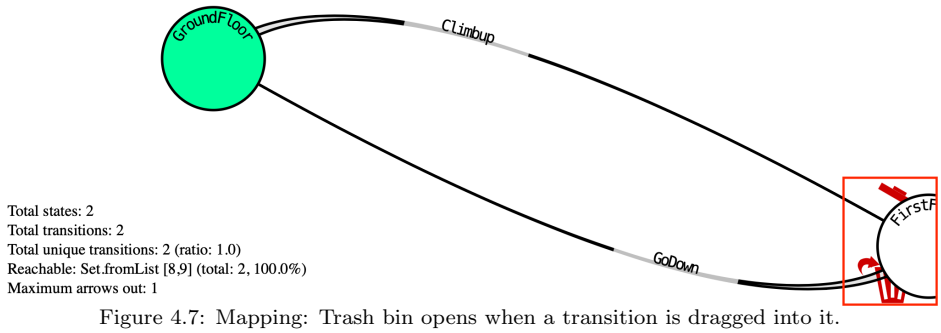


Figure 4.7: Mapping: Trash bin opens when a transition is dragged into it.

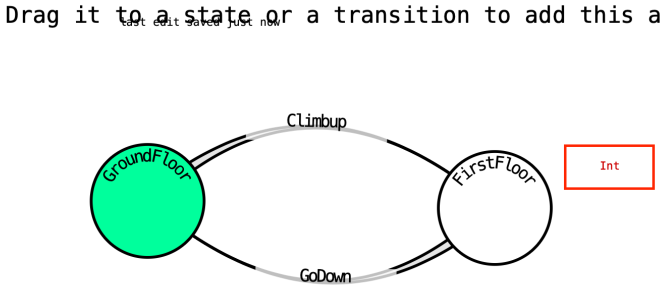


Figure 4.8: Mapping: Type sticks to the mouse when it is selected.

4.4.2.5 Constraints

These are the limits to an interaction or interface of the application. In SD Draw, each state and transition name should start with only upper-case letters and can have

alphanumeric in the rest of its name. When a name starts with a number or if it is left empty, an error message will be displayed, as shown in Figure 4.9. Similarly, a state and transition cannot have the same name, two states cannot have the same name but two or more transitions originating from different states can have the same name. The user is prevented from creating names violating these rules.

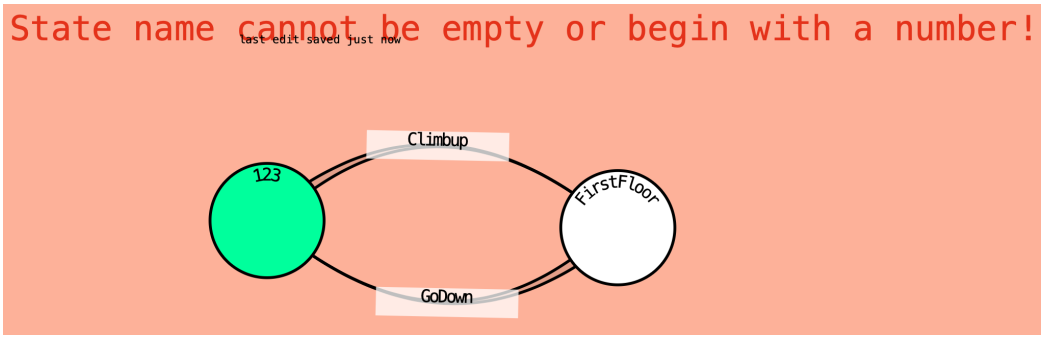


Figure 4.9: Constraints: Error message displayed when naming convention is violated.

4.4.2.6 Consistency

In an application, the same action should cause the same reaction every time. The position of the icons is consistent in all states of the application. When the application enters into the information states like the instruction page or code download page, a close button is provided on top to have a consistent feel of closing a popup in other applications (Figure 4.10).

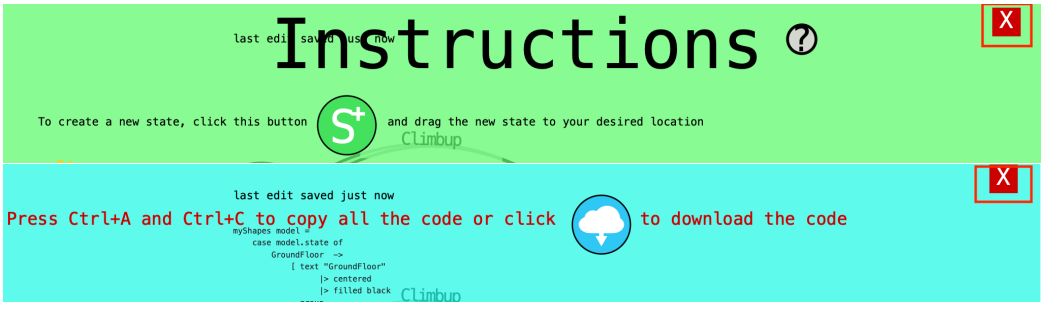


Figure 4.10: Consistency: Instruction and code-generation pages with close buttons.

4.4.3 Code Generation

With Elm's algebraic data types, generating the code from a state diagram is straightforward. The set of possible transitions is mapped to the `Msg` type and the set of possible states is mapped to the `State` type. Here, messages are transition labels and the curve is the transition. For instance, for the diagram shown in Figure 4.2, the data types generated are respectively:

```
type Msg = Tick Float GetKeyState
         | GoInside
         | EnterMusicRoom
         | LeaveMusicRoom
         | EnterGym
         | EnterHallway
         | TakeEmergencyExit
         | GoOutside

type State = Outside
           | Hallway
           | MusicRoom
           | Gym
```

Care must be taken not to generate duplicate transitions in the case where the same name is used for multiple transition arrows. The `Tick` message is platform-specific and sends the current time and keystrokes every 1/30th of a second, for easily making animations and taking keyboard input.
The `update` function is also generated based on the structure of the diagram which includes the logic for transitioning from state to state. Since all transitions are put into the singular algebraic data type `Msg` as constructors, there is no type-level safety for restricting which messages can be sent from which states. Instead, this logic is generated in the update, which pattern matches on the current state to determine which state to go to and defaults to keeping the current state the same if the message is sent from a state where it should not be according to the diagram. There is, of course, nothing stopping the users from modifying the code so as to no longer match the diagram, but generally beginners do not need to understand the update function to start creating their games.
Below is a snippet of the update function generated for the example in Figure 4.2:

```
update msg model =
    case msg of
```

```
GoInside  ->
    case model.state of
        Outside  ->
            { model | state = Hallway  }
        otherwise ->
            model
```

The `view` function is generated to display the correct state when in that state, as well as pre-populating each place with a basic text field to identify the current state and basic buttons for transitioning from state to state. Using knowledge from previous workshops, students can add graphics to each "page" of the app or change the buttons into more interesting objects, such as door handles or levers. See Figure 4.11 for an example of how the compiled code looks by default.

4.4.4 Adding Graphics with Elm

Once the code is generated, students can compile the code to get a bare-bones app with titles for each state and buttons representing each transition out of that place. Figure 4.11 shows the generated app for the example in Figure 4.2. Students can use their existing graphics knowledge to create pictures and/or animations for each place.

4.5 Tool Improvements

Given that most of our teaching has been forced online due to the ongoing COVID-19 pandemic, more features are planned for improving distance education. Instructors can currently view and make changes to students' state diagrams, but live editing and viewing and shared control by teams would significantly improve distance learning and teamwork.

The next step towards model-driven engineering requires the integration of the state diagram editor with our Web IDE, allowing the code generation button to automatically open a game slot with the generated code. Full model-driven development would add the ability to make changes in the state diagram and have them mirrored in the code. This is more work, but is important to support an iterative design thinking approach to development.

Collaboration could also be supported by allowing sub- or nested- state diagrams; that is, allowing an entire state diagram to exist within a state of a larger diagram.

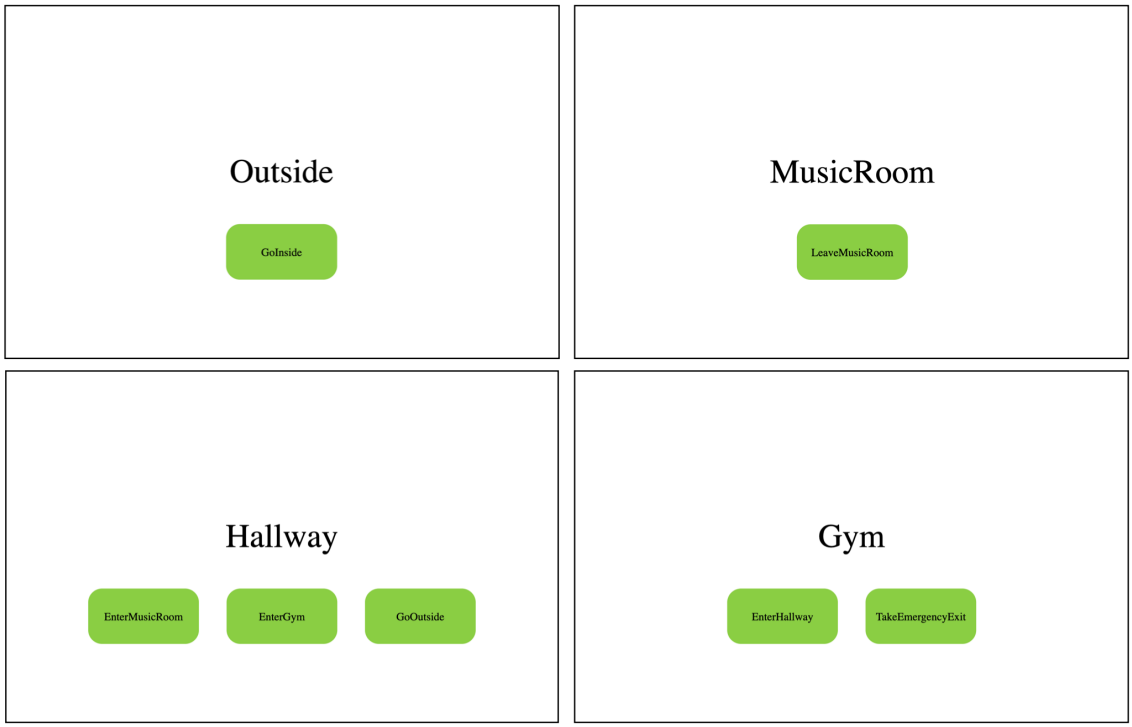


Figure 4.11: From the state diagram in Figure 4.2, a basic Elm application can be generated using the GraphicSVG library. Shown here are the four different "pages" the app can be in, one for each state in the diagram. Each place is given by default a basic title text and buttons for each transition, with the appropriate logic to transition to the correct state when clicked. Students can use existing knowledge from previous lessons to design graphics for each page, or even change the buttons themselves.

Students are interested in designing mini-games as part of a larger game, and in fact we encourage this with a summer camp. Nested state diagrams would allow students to integrate mini-games without mentor involvement, as is currently required[3]. This would require that sub-diagrams can be tested independently, similar to the support we already provide for individual frames in animated comics.

Beginner students can get very far with data-less states and transitions but eventually fall prey to what is known as a "state explosion," where students create many states and transitions to represent data which would make more sense as a data type like a Boolean, integer, etc. Even in our first 2.5-hour workshop we saw students making such diagrams, with states representing things like the amount of health an enemy has left and transitions for dealing with different damage values. We believe that students do benefit from the discussion and problem-solving that went into making

[3]http://outreach.mcmaster.ca/#camps and https://macoutreach.rocks/escapemathisland/

such complex diagrams, based on discussions we overheard during the session. Furthermore, generally students are not ready for things like integers being added to their states and transitions until they have had the chance to design their diagrams, generate the code, and discover the state explosion problem on their own. However, especially (or perhaps, only) when these tools are used in longer-term settings like a summer camp, eventually the basic "untyped" diagram is no longer powerful enough to support the students' ideas. Thus, future work includes finding the best way to introduce and teach these concepts, as well as support for user-defined algebraic data types and an interface to model such data. As previously mentioned, the lesson intended to introduce the concept of associating data with states and transitions must first motivate its need in the form of a student-generated problem and then present its solution as a much simpler diagram, even if this complicates handwritten (and generated, for that matter) Elm code.
One category of statistics evaluated above revealed that most of the children were able to use a transition to reach each state they created (see Figure 6.2). However, a few outliers showed that some of the students had difficulty with this, leaving certain states unreachable. In the future we should not allow code generation when some states are unreachable (or we could just warn them), but explain to the user that each state needs a transition leading to it.

4.6 Self-Hosting

Self-hosting means implementing a tool using the tool itself. Eventually, we would like to self host SD Draw. To discover the features we would need, we drew a state diagram based on the current implementation, see Figure 4.12 and made the following observations:

- Each drag state can be drawn as a state.
- Each message can be drawn as a transition.
- The actual application uses a product of states, i.e., drag state, trash bin state, code generation and information states. There is no way to support multiple state types in the current version. We will propose a solution to this problem using algebraic data types, including recursive types, in the next section.
- Currently mouse events encoded using `notifyTap`, `notifyMouseDown`, etc. are recorded as transitions. There is no way to support the keyboard interaction as

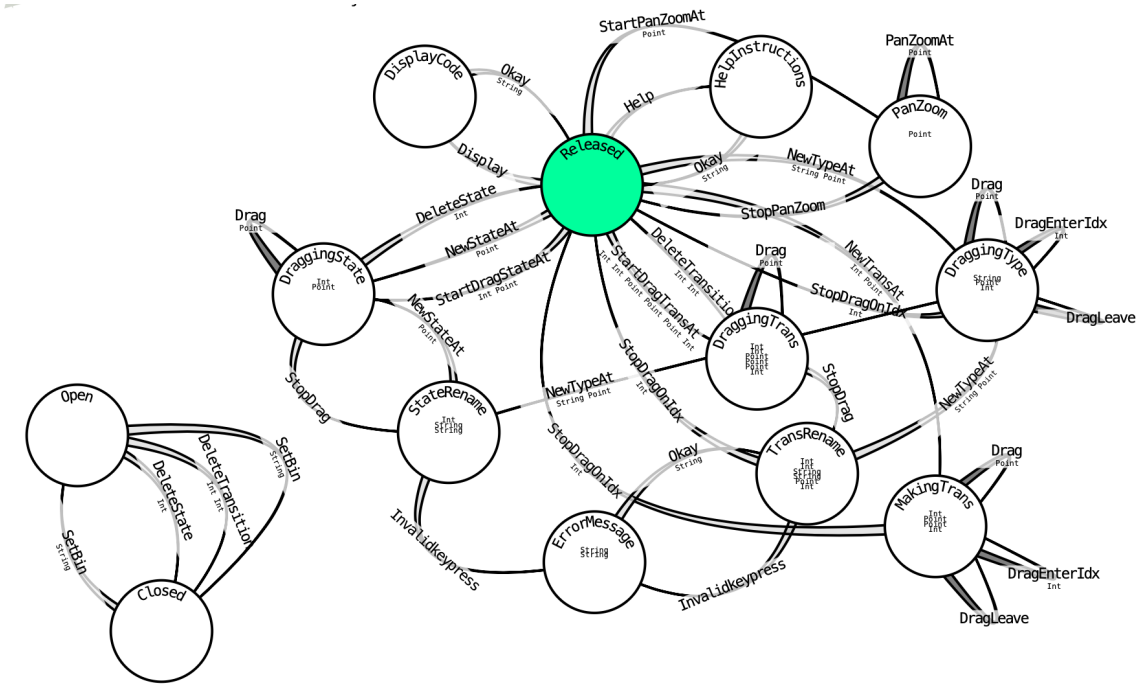


Figure 4.12: State diagram of SD Draw.

the app is currently implemented. This is partly because the app is implemented using GameApp, and a reimplementation as an html app would partly solve this problem.

- Undo and redo cannot be represented in the current SD Draw state diagram, because they use lists of the underlying state, and an error state is implemented as a constructor in the drag state with a reference to the underlying state. These would be handled by adding algebraic data type support, but users may benefit from standardized support for undo and error states, or more generic modal dialog windows.

4.7 Recursive States, Nested States and User-Defined Algebraic Data Types

As mentioned earlier, state diagrams with data like integers or booleans associated with states and transitions are more powerful than data-less state diagrams. In order

to gather feedback from users, our application currently supports four data types `Int`, `Float`, `String` and `Point` (a type alias for `(Float,Float)`). We propose to support user-defined data types by allowing users to use the type associated with another state diagram in the associated data for states and transitions. This would allow nested state diagrams, including sums and products, without adding to the visual language.

While theoretically interesting, this would be like programming without using libraries. For example, consider the case where the programmer is required to get list of inputs from the user. It would better practice for the programmer to use the existing `List` library in Elm. Other situations are best handled using `Dict` or `Set`. So these should be provided, but this raises the question: Should we allow for "state diagram constructors" which take type parameters or state diagram parameters, in the same way as `List` takes a type parameter? This is an interesting question for future research.

Currently we have the Error state, where the user can go back to the previous state to avoid the cause of the error or else the user cannot go to other states. As mentioned in the discussion of self-hosting, this requires the use of recursive data types, including embedding the state type for the current state diagram within the diagram itself. If this feature is added, we could add standard support for dialogs with different buttons leading to different states, such as a warning state, where the user will have OK and CANCEL buttons. Upon clicking the OK button, the user would be taken to another state, upon clicking the CANCEL button, the user would return to the previous state. The data constructor for such a Warning state could be defined as shown below,

```
type State = Start
           | OtherStates
           | WarningMessage
                 String -- message to display in modal dialog
                 State  -- state to go back to if cancel
                 State  -- state to go forward to if ok
```

To summarize, a state diagram defines two algebraic data types, one for states and one for messages. We should be able to refer to the types associated with the current state diagram, creating a recursive data type. We should be able to refer to the types associated with other state diagrams, creating nested state diagrams. Since we are only defining the structure in the state-diagram editor, we do not need to refer to individual transition functions, only types, at this level. In the generated code, the programmer will need to refer to the individual state and message constructors and functions implementing the transitions.

Also currently we may only add the datatypes to the states and transitions, we cannot edit the order or delete the types added to them. This will be a priority task when implementing algebraic datatypes.

4.8 Keyboard Shortcuts

Support for keyboard alternatives to buttons and drag operations is limited in both user-generated code and in SD Draw itself. This could be addressed by changing the structure of generated code. The generated update function includes state-updating logic. We could instead separately generate the user interface related code as a dedicated functions called twice from the update function, i.e., once for handling mouse events (via transitions named in the state diagram) and once for keyboard events (in the `Tick` message required by GameApp). We could also use the recursive call of update function if necessary in future.
The handling of keyboard events could be automatically generated if the SD Draw app allowed users to specify keyboard shortcuts in the transitions.

4.9 Semantic Versioning

In the current SD Draw app, state diagrams are not saved with versioning. This could become a problem when state diagrams can include each other via their associated types. We could save the state diagrams with different versions that could either be versioned using timestamps or by manual naming conventions. The best method of doing this is a question for future research.

Chapter 5

Design of the Teaching Experiment

This section discusses the design of the lesson and the challenges given to students.

5.1 Lesson Design

We developed a lesson plan prior to teaching the Grade 4 students and radically simplified it when teaching Grade 5 group based on feedback from the classroom teacher that it was too long and included too many examples. For Grade 4, we introduced the concept of a state diagram using the Moo-Quack game (see [d'Alves et al., 2018]). Here, the instructor shows a state diagram in which the states are animal noises and transitions number of raised arms. Students are often confused at first but catch on quickly when classmates start making animal noises. We then explained the concepts of a states (places) and transitions (actions) using examples drawn in our tool of states of matter, Canadian provincial land borders, and school navigation (see Figure 4.2). Finally, we showed the children how to generate and run the code in a game slot in our Web IDE. We then split the class into groups, assigned each to a breakout room, and challenged them to make their own game.
We then received feedback from the classroom teachers. For the Grade 5 class, we focused on presenting state diagrams as the map of a concrete adventure game and used the vocabulary of "places" and "ways." From there, we split the children up into groups and asked each group to chose one person to edit the map while sharing their screen, and we gave them a Google Slides template to use in identifying tasks and then assigning them to group members. One of the teachers asked them to think

about the scale of their game in terms of enjoyment in reading a story. We found that if it is too long, people will lose interest.
After class, we were able to retrieve the state diagrams, and approximately assign them to the two grades (i.e., Grade 4 and 5). Due to the fact our program uses randomly generated logins, we do not keep identifying information, and many of the students continued working on their state diagram after class, we expected to report on the two classes as a whole. Statistics on the state diagrams are reported below.

5.2 Challenge Design

We visited the Grade 5 students the next week for an additional hour and gave them four challenges to measure the impact of our teaching on their understanding on state diagrams and their ability to translate from one representation/implementation to another. Each challenge had two variations; the first one, based on the state diagram in Figure 4.2, contained only concrete place names. The second variation had abstract states: a closed box with a scratching noise, an open box with a dragon flying around, and a closed box. The challenges were to:

1. Draw a state diagram using our tool from an English paragraph:

    a Your task is to make a state diagram to help a new classmate find their way around inside your school. The classmate starts Outside. From the Outside, they can go inside to the Hallway. From the Hallway, they can enter the Music Room. From the Music Room they can leave and go back into the Hallway. From the Hallway they can also enter the Gym. From the Gym they can leave the Gym and go back to the Hallway, or in an emergency they can take the emergency exit to go back Outside. Your classmate cannot enter the school through the emergency exit.

    b You are designing a state diagram for a video game about a dragon. The game starts with a Closed Chest, with a scratching noise inside. The player can open the chest, which will cause a dragon to start flying around. The player can then close and open the chest as many times as they want, but the dragon will still be flying around and the chest will remain empty. There are many correct answers: remember, just do your best!

2. Draw a state diagram using our tool from English bullet points.

    a • One place you can be is Outside.

- One place you can be is in the Hallway.
- One place you can be is in the Music Room.
- One place you can be is in the Gym.
- From Outside you can go inside to the Hallway.
- From the Hallway you can go back Outside.
- From the Hallway you can also go into the Music Room.
- From the Music Room you can go back into the Hallway.
- From the Hallway you can also go into the Gym.
- From the Gym you can go back into the Hallway.
- From the Gym you can also go through the emergency exit to go back Outside.
- You cannot enter through the emergency exit.
- You start Outside.

b
- One thing that can happen in the game is the chest is open and a dragon flies out of it.
- One thing that can happen in the game is the chest is closed but the dragon is still flying.
- If the chest is closed and you hear a scratching sound, you can open the chest.
- If the chest is open and the dragon is flying, you can close the chest, but the dragon will still be flying.
- If the chest is closed and the dragon is flying, you can open the chest again.
- The dragon never enters the chest again.
- The game starts with the closed chest with the scratching noise.

3. Describe a state diagram using English based on a diagram drawing using our tool.

4. Draw a state diagram using our tool given a generated (and compiled) game, e.g. Figure 4.11.

Chapter 6

Results of Evaluation

In this chapter, we analyze the results of our test both quantitatively and qualitatively. Quantitatively, we use descriptive statistics for reachability, and test the null hypothesis that the diagrams created by children come from the distribution of randomly generated directed (multi-)graphs, and can say with confidence $p = 0.001$ that this is not true, so children do understand the concept of reachability, and can design it into their diagrams. Qualitatively, we present the "median" results of the eight different translation exercises. We also note that students who received the abbreviated introduction to state diagrams based on adventure game, without examples showing travel between provinces of Canada and states of matter, came up with better diagrams. To understand the prevalence of abstract states, we also use descriptive statistics.

6.1 Quantitative Analysis of State Diagrams

Figure 6.1 shows a scatter plot of the total number of transitions versus states. The line $y = x$ is plotted as a dashed line, and $y = 1.5x$ plotted as a dotted line. Points below $y = x$ indicate more states than transitions and a disconnected graph. The points near the x-axis are likely abandoned diagrams. Points near $y = x$ indicate diagrams with close to one transition per state, e.g. a tree. Points above $y = x$ indicate more complex games with multiple paths. The line $y = 1.5x$ represents the visual centre of the diagrams with multiple choices, and represents having three transitions for every two states. There are eight above the line (with more transitions)

and eleven below the line (with fewer), so the visual centre is actually slightly above the median.

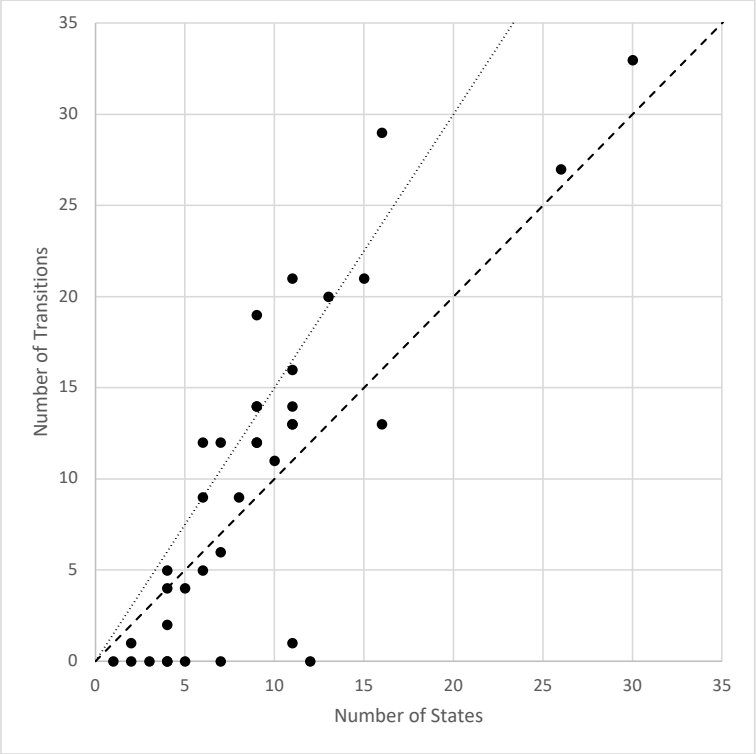


Figure 6.1: A scatter plot of the numbers of transitions and number of states for each diagram. The line $y = x$ is plotted as a dashed line, and $y = 1.5x$ plotted as a dotted line. Points below $y = x$ indicate more states than transitions and a disconnected graph. The points near the x-axis are likely abandoned diagrams. Points near $y = x$ indicate diagrams with close to one transition per state, e.g. a tree. Points above $y = x$ indicate more complex games with multiple paths. 6.2.

Figure 6.2 shows a scatter plot of reachable vs total states. Points on the diagonal ($y = x$) indicate that all states are reachable from the starting state. The points on the line $y = 1$ probably correspond to abandoned diagrams, since only the starting state is reachable.

Figure 6.3 shows a scatter plot of abstract versus concrete states. The dotted line has slope -2.9 which suggests that the effort required to add a concrete state is three times the cost of adding an abstract state.

Consistent with our observation that many diagrams (10 out of 38) were abandoned after students realized that only one of the diagrams created by their group could be used in the next step, we note a cluster of disconnected diagrams, but otherwise the diagrams indicate strong understanding of and engagement with the material. There was widespread adoption of abstract states, even though most students were only

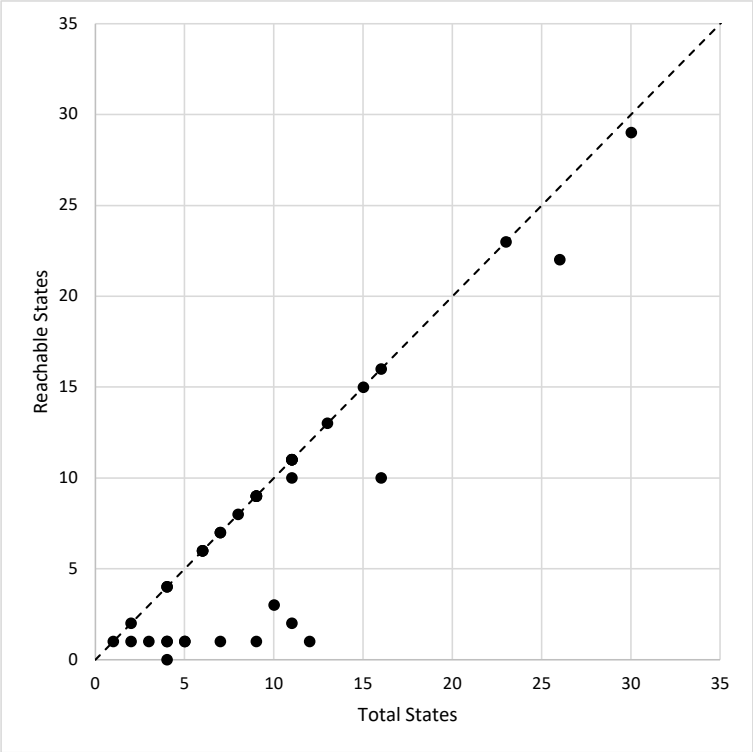


Figure 6.2: A scatter plot of reachable versus total states in students' diagrams. Points on the diagonal ($y = x$) indicate that all states are reachable from the starting state. The points on the line $y = 1$ probably correspond to abandoned diagrams, since only the starting state is reachable.

taught to think about and create concrete states. In fact, the most productive groups produced many more abstract than concrete states.

To answer **RQ3**, we have tested the hypothesis that the diagrams created come from the distribution of randomly generated diagrams. To use the Anderson-Darling single-sample test, we need to know the cumulative probability distribution (CDF) for the number of states reachable from the starting state. We approximate this discrete by generating random diagrams. Using the empirical CDF, we can randomly generate samples of diagrams and evaluate the Anderson-Darling statistic to approximate its distribution, and estimate the p-value for the child-created diagrams.

In Tables 6.1, 6.2, and 6.3, we display the probability distribution for reachability for each diagram with respect to the number of states and transitions in the diagram. Considering that the initial state is always reachable, the minimum reachability is one. Given a random state diagram, we calculate the reachability using Dijkstra's algorithm. We approximate the probability distribution function for the number of

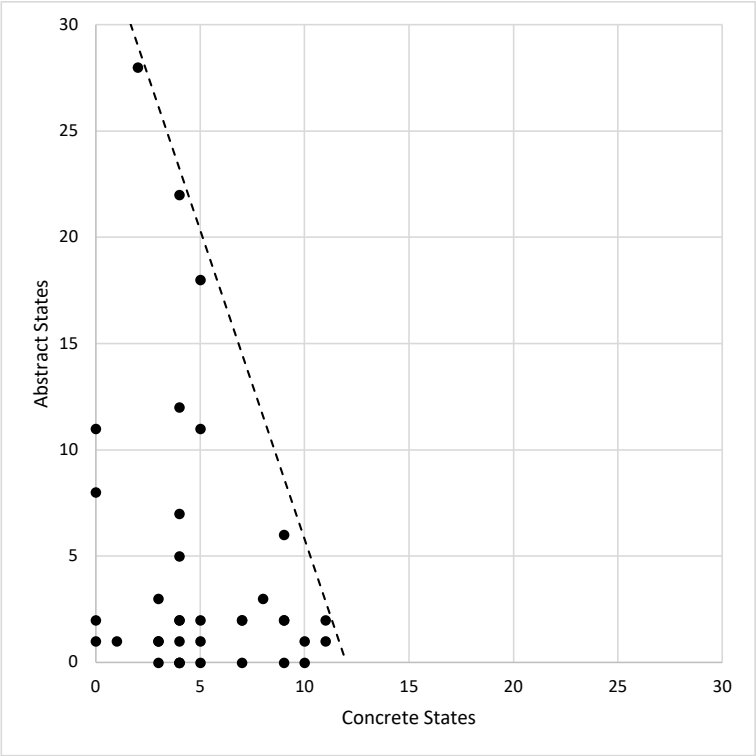


Figure 6.3: A scatter plot of concrete versus abstract states in students' diagrams. The dotted line has slope -2.9 which suggests that the effort required to add an concrete state is three times the cost of adding an abstract state.

states reachable from the initial state using a normalized histogram. Before computing the Anderson-Darling test, we note that the histograms for randomly generated diagrams are not consistent with the distribution of student-generated diagrams. In Table 6.1, we see the probabilities for the diagrams with 11 states, as well as the observed reachability in the five diagrams created by students. In the first three rows, the maximum probability for any observed reachability is 1% (rounded to the nearest percent). Although the probability of full reachability grows with the number of transitions, it is never high. The maximum probability for full reachability is in the third row of Table 6.2, for diagrams with 9 states and 19 transitions, at 18%. In Table 6.3, we show two cases with large numbers of states and transitions, which illustrate that for larger diagrams, a large number of states are needed to make reachable graphs at all likely to arise randomly.

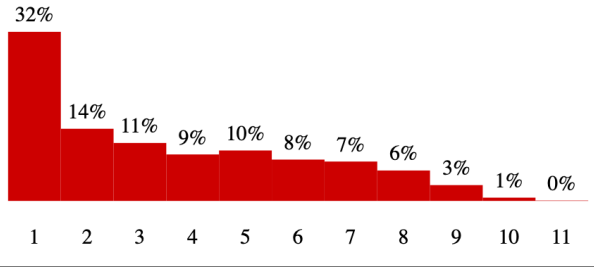
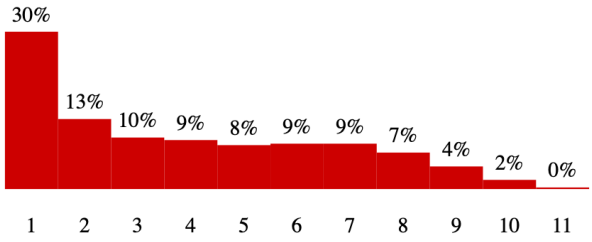
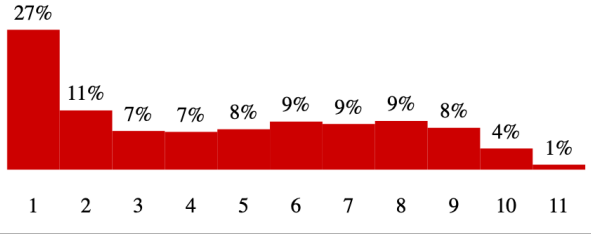
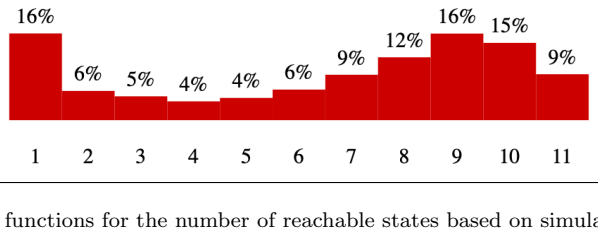
States	Transitions	Probability Distribution Function	Observed
11	13		10,11
11	14		11
11	16		11
11	21		11

Table 6.1: Probability Distribution functions for the number of reachable states based on simulation of 4000 random diagrams with 11 states. The observed column has the number of reachable states in the students' state diagrams with 11 states and its corresponding transitions.

The formula for calculating the Anderson-Darling test statistic is,

$$A^2 = -n - S, \tag{6.1}$$

$$S = \sum_{i=1}^{n}(((2i-1) \div (n))[\log F(Y_i) + \log(1 - F(Y_n + 1 - i))]). \tag{6.2}$$

where, A is the Anderson-Darling test statistic, F is the Cumulative Distribution Function, and n is the number of elements(diagrams). This statistic measures the

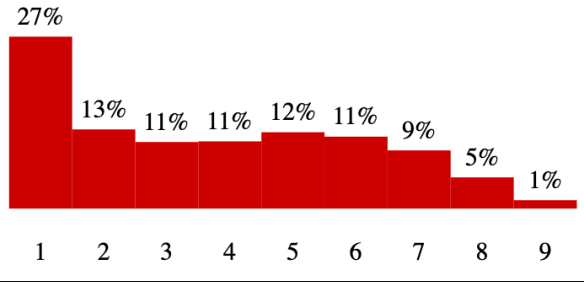
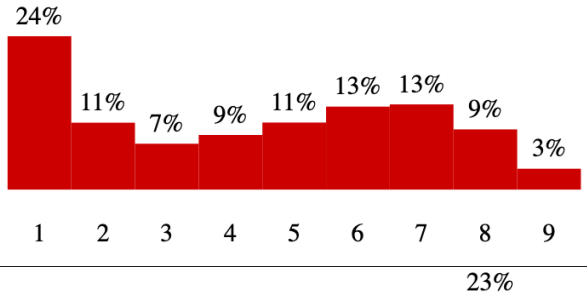
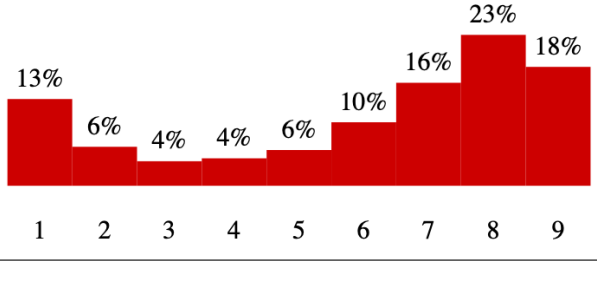
States	Transitions	Probability Distribution Function	Observed
9	12		9
9	14		9
9	19		9

Table 6.2: Probability Distribution functions for the number of reachable states based on simulation of 4000 random diagrams with 9 states. The observed column has the number of reachable states in the students' state diagrams with 9 states and its corresponding transitions.

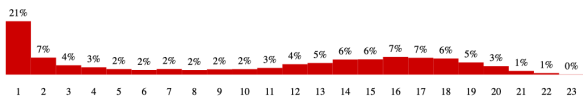
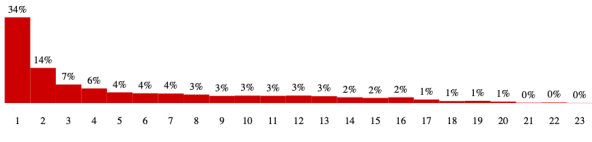
States	Transitions	Probability Distribution Function	Observed
23	38		23
30	33		29

Table 6.3: Probability Distribution functions for the number of reachable states based on simulation of 4000 random diagrams with 23 and 30 states. The observed column has the number of reachable states in the students' state diagrams with 23 and 30 states and its corresponding transitions.

match between a sample and a distribution. To use this test statistic, we need a sample of diagrams from one distribution. We have at most two diagrams with the

same number of states and transitions, but we can reduce our confidence level by assuming that the diagrams with 11 states and 13, 14 and 16 transitions all have 16 transitions, and hence come from one distribution. The CDF F is displayed in the third row of Table 6.1, but to compute the next steps we used 40K samples to get a better approximation of the true distribution. In Figure 6.4, we show the distribution of test statistics for 4000 sets of 4 randomly generated diagrams, and the test statistic for the sequence of reachabilities $10, 11, 11, 11$, i.e., 33.8. Since this is an empirical distribution based on 4000 samples, and the test statistic for these four diagrams is well outside the distribution, we are confident that $p < 0.001$, refuting the null hypothesis based on four diagrams. In the future, it would be good to calculate the confidence value for the full set of diagrams by using Fisher's combined probability test to combine the p values for each set of diagrams with a common number of states and transitions.

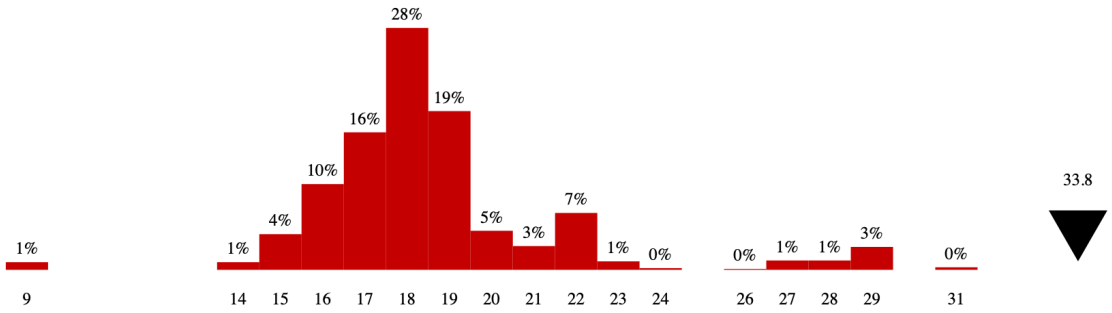


Figure 6.4: Anderson-Darling Test Statistic Distribution for 4 diagrams each with 11 states and 16 transitions, approximated using 4000 randomly generated sets of 4 diagrams. The horizontal axis shows A^2, the Anderson-Darling test statistic, and the vertical axis shows the probability. The black triangle shows the test statistic for the 4 diagrams produced by children with 11 states and 16 or fewer transitions (33.8). Since 4000 sets were used to generate the histogram, and 33.8 is well outside the randomly generated tests, we estimate that the confidence value $p < 0.001$.

6.2 Qualitative Analysis of Challenges

We report the results of the challenges qualitatively because (1) to understand where we needed to improve instruction without asking for too much student time we had 3-5 students completing each challenge, (2) we asked some children to write paragraphs, and (3) we asked some children to create state diagrams. In all cases, we could identify a "median" response, which we report here.

In the concrete school challenges, we found confusion about whether the emergency

exit was a state or a transition, as seen in Figure 6.5 when translating from a paragraph description.

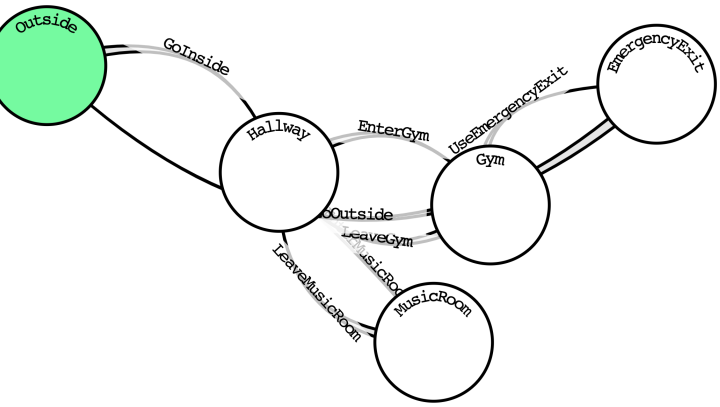


Figure 6.5: Median result for challenge 1a. Note the extra EmergencyExit state.

Whereas given a point-form description, they were less likely to leave off or add additional states or transitions, as seen in Figure 6.6.

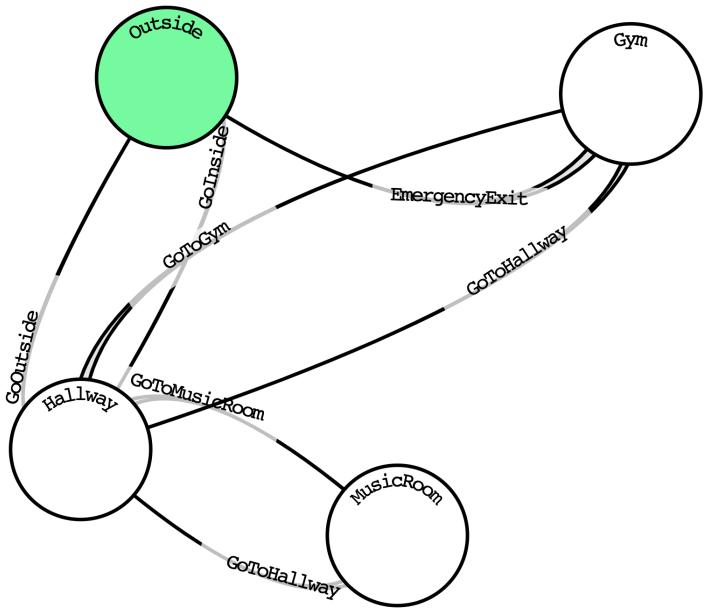


Figure 6.6: Median result for challenge 2a matches expectation.

When asked to describe the state diagram in English, most students hesitated to get started, but then used narrative to thread together a description. Many students

asked "when should I stop?" because they realized that the narrative could go on forever due to a cycle, giving a glimpse into how they were systematically analyzing the diagrams. Our median response for Challenge 3a was: *You start outside the school. If you go inside through the door, you'll be at the hallway. Here, you can access all the different rooms or exit the hallway to go back outside. The music room is the room labeled "music," and you can enter and exit it through the door. From the hallway, you can also enter the gym room, and exit it back through the door. If you are in the gym and there is an emergency, you can take the emergency exit instead of running back to the hallway and exiting that way. There is no emergency exit in the music room, since there is nowhere to go after you leave.* There was not median response to the final challenge due to miscommunication about the need to work individually.

The abstract dragon-in-a-box challenge was more difficult, but they did best when translating a working app into a diagram, see Figure 6.7, perhaps because they did not have to think about the semantics of the abstract situation. When they had to reason about the English description, they were uncertain, but still seemed to understand the basic definitions and task, see Figures 6.8 and 6.9.

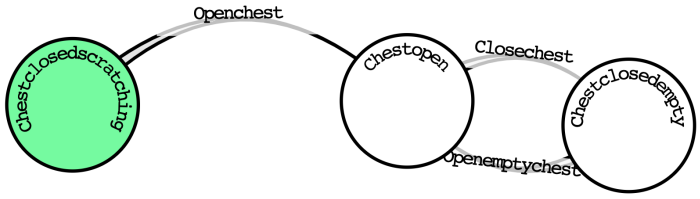


Figure 6.7: Median result for challenge 4b matches expectation.

Going the other way, they again showed their understanding of the concept, but felt the need to add narrative bridges, perhaps because we used narrative in our descriptions, or because that is how they understand them. Our median solution: *You start with a chest, and you hear a faint scratching noise inside. If you run away, you will find that the room has no escape, and you have to open the chest. When you open the chest, The dragon flies out, and all you have to do is to close the chest, and you win.* The inclusion of narrative elements to explain properties of the state diagram was surprising and shows their understanding of the state diagram model. In this response, *you will find that the room has no escape* shows that they understand how the diagram models the allowable transitions at a given place. The addition of *and you win* shows another common observation we saw with their responses: the student adding their pre-existing knowledge of video games to their responses,

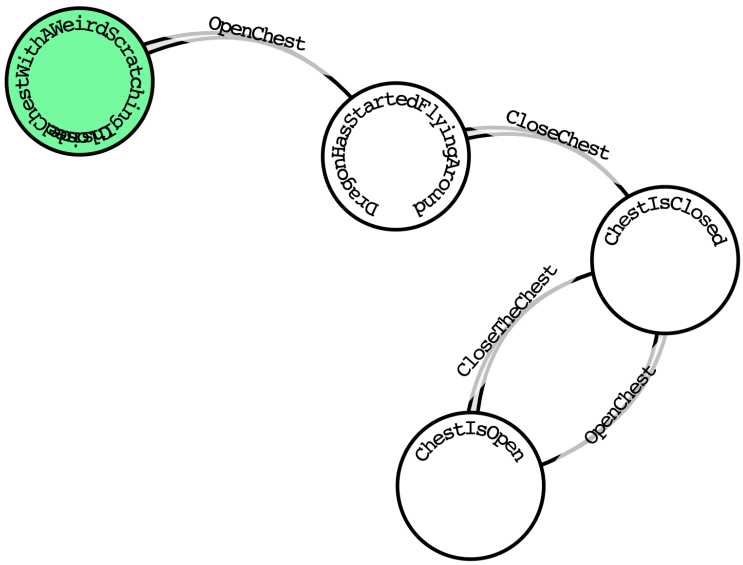


Figure 6.8: Median result for challenge 1b with an extra state emphasizing the start of the dragon flying around.

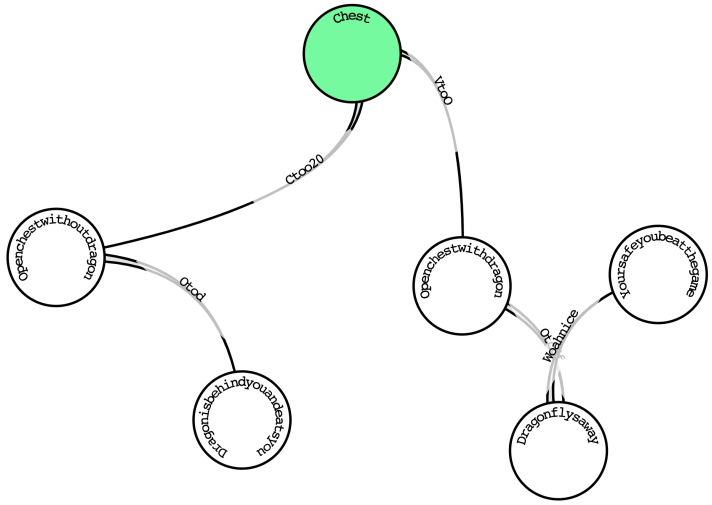


Figure 6.9: Median result for challenge 2b without a circuit. Linear narratives do not have circuits, and the additional states include interpretation beyond the specification which serve to make the narrative more interesting.

whether or not that information is encoded in the diagram, which in this case it is not.

Chapter 7

Discussion

The primary purpose of this study was to test the tool and pedagogy of teaching state diagrams. Our design checklist and identified future work is presented in this section.

One of the more interesting results of the classes was that the students began to differentiate between concrete states (places one could go) and abstract states (new states of being), and apply different levels of both. For example, in a game where a character can enter a barn, they find a dragon. The character can transition to a new state by feeding the dragon. The barn itself would be a concrete state, as it is a place the character can enter. The fed dragon would be an abstract state, as you cannot return to the state of the dragon being unfed. Both types of states were given in the examples provided in class; however, it was never explained that there might be a difference between the two.

Once the results of what the students had created was reviewed, it was hypothesized that they could be sorted by complexity in two forms: number of states and transitions, and number of concrete and abstract transitions. The results showed few games of moderate complexity, instead showing that students favoured either high complexity or low complexity. Students who used more abstract states also had higher rates of states and transitions, while those with more concrete states had less states and transitions. Further research might be able to explain why this trend was seen.

It was also noted, when checking in on the state diagrams created by the students, that some continued their work after the classes had ended. Though these results were not analyzed due to the small participation number, it seemed that those who showed more complexity by using abstract states and a higher number of states and

transitions overall were the same students who continued working on their projects in their own time. This showed the investment the students had in their stories that had not been predicted. As mentioned previously, the ability to visually map out ideas through concept mapping or state diagrams has been shown to improve the efficiency of a student's writing. However, this finding also suggests the potential for higher engagement, interest and initiative when learning to code, and create culminating projects fusing coding and other curriculum areas.

7.1 Pedagogical Improvements

We found that students were generally able to translate between different equivalent specifications of state diagrams, but that they were more successful and (anecdotally) more comfortable with point-form specifications than paragraphs. This suggests designing and testing a staged curriculum in which translation between diagrams and point-form specifications should be taught first, followed by paragraph descriptions, and the advantages of translating from working game to diagram to point-form to paragraph should be measured. Teachers are always trying to find ways of engaging reluctant writers, and we hypothesize that this is one way of leveraging children's engagement with video games. Different teaching styles should be investigated for different age groups. Furthermore, demographic information collected beforehand can allow the foundation for comparing genders, ages and developmental differences.
Knowing that many students created abstract states without prompting, see Figure 6.2, we should add a follow-up lesson after children have produced one (or more) state diagrams to introduce abstract states to all children. We should then design new challenges to evaluate whether all children are able to understand and use abstract states, and whether there are differences based on developmental level.

7.2 Limitations

Firstly, our results are based on approximately 38 diagrams from a total of 70 students from an enrichment program. While we cannot draw conclusions about the general school population, our experience in piloting curriculum with this group over 15 years suggests that this new activity will also work in all classrooms, and for longer engagements.

Secondly, we did not follow the same lesson plan when teaching the Grade 4s as when teaching the Grade 5s. After one lesson with the Grade 4 students, a streamlined teaching plan was used for the Grade 5 students, with the main differences being the skipping of explanations of state prior to designing a game together, and the reiteration of the importance of choosing one team member to share their screen while creating one shared state diagram. Because we use anonymized accounts, we cannot segregate and compare their results. Similarly, Grade 5s are both more mature and cognitively advanced than Grade 4s, and therefore the difference in how they were taught on days one and two of this study may have contributed to the differences within our results rather than it being their age or grade. That being said, a more concrete and tailored lesson plan for both grades would be helpful in increasing reliability and validity.
Finally, the online platform we used was not compatible with tablets (i.e., a small number of iPads), and students used a range including Chromebooks, with and (mostly) without mice. The lack of control around which device was being used may affect whether a concrete or abstract diagram was created, but device heterogeneity is more common than not in the classrooms we visit.

7.3 Design Checklist

CL1 *Keyboard/mouse based editing, with a minimal number of buttons.* The main goal was to minimize the number of buttons on the application. The PAL-Draw application had around 15 buttons and around 10 buttons were used for drawing basic state diagrams without any data types attached. During the first iteration of the SD Draw we had only 3 buttons to download the generated code, Undo and Redo. All other functionalities like making a new transition, deleting a state or transition, zoom-in and zoom-out were carried out using the keyboard and mouse interactions. When we taught school children to use it, though they loved making state diagrams for their adventure games, they had some difficulties accessing some controls using keyboard and mouse interactions (especially simultaneously). During the next iteration, we introduced a trash bin for deleting states and transitions by simply dragging them over the trash bin, a recenter button to bring back all the states and transitions to the original scale and position, and in the last iteration, a new small control to click and drag to make a new transition, Zoom-in and Zoom-out buttons were added along with the Instructions screen button. Now, SD Draw has 7 buttons which

are fewer and using much less screen space compared to PALDraw.

CL2 *In-place editing of names for places and transitions (live editing).* In PALDraw, we had to select a state or transition on the left window and type its name on a text box on the right panel to rename it. In SD Draw we can select a state or transition and can edit its name directly and it gets updated in real-time.

CL3 *Drag-and-drop where possible* In SD Draw, we cannot just drag and drop the states and transitions, we can also add a start state, four different data types to a state or transition by drag and drop. Whereas PALDraw had separate buttons for changing a start state and to add data types.

CL4 *Minimize clicking* As mentioned earlier, SD Draw has fewer buttons and most of its functionality is based on keyboard and mouse interactions, we have fewer button clicks. For example, saving the name of a state or transition can be done by pressing the "Enter" key instead of using a separate button like in PALDraw.

CL5 *Leave space for data in states and transitions* Every state and transition has some space on it to add and display the data types associated with it.

CL6 *Experiment with data types* Currently SD Draw application supports four data types. We better understand the importance of supporting algebraic data types after trying to self-host the application. Hence we are planning to introduce the custom data types as discussed in Section 4.7.

CL7 *Snappy app (immediate feedback)* While editing is snappy for small state diagrams, performance still needs to be improved for large diagrams. It may be necessary to cache some computations in the model so they are not recalculated on every screen refresh.

CL8 *Elm code generation for buttons* Unlike PALDraw, SD Draw generates an app with working buttons. This makes teaching a lot simpler because kids observe the flow of the diagram matching generated code.

CL9 *Validate state and transition names. No two states or transitions can have the same name.* We have added multiple validations for the naming convention of the states and transitions as mentioned in 4.4.1 and it does not let users create invalid names which would causes compiler errors in the generated code.

CL10 *Widget and Server implementation* The SD Draw application is hosted on the macoutreach.rocks server, and ready to be integrated into the web development environment.

Chapter 8

Conclusion

In this chapter, we will summarize and conclude our work. This was a proof-of-concept study, but we are satisfied that it already provides enough value to be offered to a wider school population. That said, we have a plan for improvements and future studies. Further study is required to match the teaching to the development level of the students. In Section 4.5, we have outlined the further advancements of SD Draw, and the ways in which our research group will anticipate using it. This includes our direction on the additional teacher's application as introduced throughout this work, and the general benefits we hope to achieve from our entire system when the teacher's application is finished.

The initial goal of SD Draw application is to re-implement the application with a new, simpler interface, using the screen space better and requiring fewer clicks. We have completed two main goals so far, (1) re-implemented it with a new interface and (2) tested it with kids, testing whether they can use it and whether they understand the underlying state diagrams, giving answers to our main research questions. We have used Design Thinking methodology to develop this application and we went through several iterations of application improvement, and are not finished. We came up with a list of improvements to be made in the new SD Draw application as mentioned in the chapter 3. We did achieve most of those tasks, as explained in Section 7.3, and learned a lot.

Research Questions

We have answered all of the research questions:

RQ1 *Do grade 4-5 students demonstrate an understanding of State Diagrams by being able to translate between different representations?* Yes, when students were given different representations and asked to convert them, they were able to do so. See Section 6.2.

RQ2 *Do grade 4-5 students demonstrate equal facility for translating between different representations of state diagrams?* No. When a state diagram is given and students are asked to write a description about it, students were confused how much to write, especially in the case of a cycle in the state diagram. Students also found it easier to interpret point-form specifications rather than paragraphs, and found the conversion of a working app into a State Diagram easiest of all. See Section 6.2.

RQ3 *Can grade 4-5 students understand the role of reachability? Assuming that students who did not understand the role of reachability would generate random graphs, what confidence do we have that the graphs are more reachable than random graphs?* We have a confidence of $p < 0.001$ that they understand reachability enough not to draw random diagrams, based on an monte carlo simulation of the Anderson-Darling statistic for four of the diagrams with 11 states. See Section 6.1.

RQ4 *Are grade 4-5 students engaged by state diagrams and their applications to adventure games?* Yes, students were engaged by state diagrams. They expressed interest in creating the state diagrams during class, and showed even more interest in designing the levels or difficulty of the levels of their game than designing the graphics for their game. Several groups continued working after school. See, e.g., Table 6.3.

RQ5 *Do grade 4-5 students understand abstract and concrete states equally well? Will students presented with concrete states generalize to abstract states without prompting?* For the grade 5 students, we did not explain about concrete and abstract states. But when we asked them to draw their own State Diagrams for their favorite game, some students came up with abstract states. See Figure 6.3.

Bibliography

Shaaron E Ainsworth and Katharina Scheiter. Learning by drawing visual representations: Potential, purposes, and practical implications. *Current Directions in Psychological Science*, 30(1):61–67, 2021.

Lynne Anderson-Inman and Mark Horney. Computer-based concept mapping: Enhancing literacy with tools for visual thinking. *Journal of adolescent & adult literacy*, 40(4):302–306, 1996.

Mordechai Ben-Ari. Constructivism in computer science education. *Acm sigcse bulletin*, 30(1):257–261, 1998.

Marina Umaschi Bers. Coding as another language: a pedagogical approach for teaching computer science in early childhood. *Journal of Computers in Education*, 6(4):499–528, 2019.

Helen Brown. Macventure: An iPad application design for social constructivist e-learning. Master's thesis, McMaster University, http://hdl.handle.net/11375/20478, 2016.

Ni Chang. The role of drawing in young children's construction of science concepts. *Early Childhood Education Journal*, 40(3):187–193, 2012.

Li Cheng and Carole R Beal. Effects of student-generated drawing and imagination on science text reading in a computer-based learning environment. *Educational Technology Research and Development*, 68(1):225–247, 2020.

British Design Council. What is the framework for innovation? design council's evolved double diamond, Sep 2019. URL https://www.designcouncil.org.uk/news-opinion/what-framework-innovation-design-councils-evolved-double-diamond.

Evan Czaplicki. Elm: Concurrent FRP for Functional GUIs. *Senior thesis, Harvard University*, 2012.

Bogdan Denny Czejdo and Sambit Bhattacharya. Programming robots with state diagrams. *Journal of Computing Sciences in Colleges*, 24(5):19–26, 2009.

Curtis d'Alves, Tanya Bouman, Christopher Schankula, Jenell Hogg, Levin Noronha, Emily Horsman, Rumsha Siddiqui, and Christopher Kumar Anand. Using elm to introduce algebraic thinking to k-8 students. In Simon Thompson, editor, Proceedings Sixth Workshop on *Trends in Functional Programming in Education,* Canterbury, Kent UK, 22 June 2017, volume 270 of *Electronic Proceedings in Theoretical Computer Science*, pages 18–36. Open Publishing Association, 2018. doi: 10.4204/EPTCS.270.2.

E Paul Goldenberg and Cynthia J Carter. Programming as a language for young children to express and explore mathematics in school. *British Journal of Educational Technology*, 52(3):969–985, 2021.

Masaru Kamada. Islay—an educational programming tool based on state diagrams. In *2016 International Conference on Advances in Electrical, Electronic and Systems Engineering (ICAEES)*, pages 230–232. IEEE, 2016.

Dexter C Kozen. *Automata and computability*. Springer Science & Business Media, 2012.

Shriram Krishnamurthi and Kathi Fisler. Programming paradigms and beyond. *The Cambridge Handbook of Computing Education Research*, 37, 2019.

Andrea Kunze and Jennifer G Cromley. Deciding on drawing: the topic matters when using drawing as a science learning strategy. *International Journal of Science Education*, pages 1–17, 2021.

Aleksi Lukkarinen, Lauri Malmi, and Lassi Haaranen. Event-driven programming in programming education: A mapping review. *ACM Transactions on Computing Education (TOCE)*, 21(1):1–31, 2021.

Ana Francisca Monteiro, Maribel Miranda-Pinto, and António José Osório. Coding as literacy in preschool: A case study. *Education Sciences*, 11(5):198, 2021.

Don Norman. *The design of everyday things: Revised and expanded edition*. Basic books, 2013.

Bill O'Farrell and Christopher Anand. Code the future!: teach kids to program in elm. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, pages 357–357. IBM Corp., 2017.

Optimal Computational Algorithms, Inc. ElmJr (1.0). iOS App Stores: https://apps.apple.com/ca/app/elmjr/id1335011478, 2018.

Joonhyeong Park, Jina Chang, Kok-Sing Tang, David F Treagust, and Mihye Won. Sequential patterns of students' drawing in constructing scientific explanations: focusing on the interplay among three levels of pictorial representation. *International Journal of Science Education*, 42(5):677–702, 2020.

Christopher Schankula, Emily Ham, Jessica Schultz, Yumna Irfan, Nhan Thai, Lucas Dutton, Padma Pasupathi, Chinmay Sheth, Taranum Khan, Salima Tejani, et al. Newyouthhack: Using design thinking to reimagine settlement services for new canadians. In *International Conference on Innovations for Community Services*, pages 41–62. Springer, 2020.

Christopher W Schankula and Christopher K Anand. Graphicsvg [elm package], 2016-2019. URL http://package.elm-lang.org/packages/MacCASOutreach/graphicsvg/latest.

Annett Schmeck, Richard E Mayer, Maria Opfermann, Vanessa Pfeiffer, and Detlev Leutner. Drawing pictures during learning from scientific text: Testing the generative drawing effect and the prognostic drawing effect. *Contemporary Educational Psychology*, 39(4):275–286, 2014.

Leslie Suters and Henry Suters. Coding for the core: Computational thinking and middle grades mathematics. *Contemporary Issues in Technology and Teacher Education*, 20(3):435–471, 2020.

F Vico, M Molina, D Orden, J Ortiz, R Garcia, and J Masa. A coding curriculum for k-12 education: the evidence-based approach. In *Proceedings of the 11th annual International Conference on Education and New Learning Technologies*, pages 7102–7106, 2019.