Real-time Collaboration for a Block-based Programming Environment

by

Denis Li

B.S., Massachusetts Institute of Technology (2017)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of
Masters of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2018

© Massachusetts Institute of Technology 2018. All rights reserved.
Real-time Collaboration for a Block-based Programming Environment

by

Denis Li

Submitted to the Department of Electrical Engineering and Computer Science on May 25, 2018, in partial fulfillment of the requirements for the degree of Masters of Engineering in Computer Science and Engineering

Abstract

StarLogo Nova is an online education platform for programming and visualizing 3D simulations and games. It has a custom programming language, a compiler, and graphical user interface for shows systems implemented with the programming language. Unlike most programming languages, however, StarLogo Nova uses a block-based programming front-end. This unique front-end leads to interesting challenges when adding new features to StarLogo Nova’s programming editor. For this thesis, I designed and implemented a proof of concept real-time collaboration system that supports multi-user undo/redo. The related works, design, implementation, and future work are discussed in this thesis.

Thesis Supervisor: Eric Klopfer
Title: Director of MIT Scheller Teacher Education Program
Acknowledgments

I would like to thank my advisor Daniel Wendel for giving me this great project opportunity and guiding me along this challenging project. When I stepped into the lab during the beginning of the school year, I never thought I would have found a project that would be sufficiently challenging, applicable, and interesting. Also, without his initial guidance and suggestion of the Wantaim real-time collaboration system, I would have wallowed hopelessly through academic papers trying figure out what is going on.

I would also like to thank Lisa Stump for her help in letting me understand more about the StarLogo project and how truly complex it is. The work I have done here is an iteration on multiple failed attempts that failed to grasp all of the intricacies of StarLogo.

I would also like to thank my friends Eric and Martin for being in Boston during my last year at MIT when everyone I used to hang out with at MIT has left. It has been a lonely year, but it is good to know that they are here.

Lastly, I would like to thank my family and other friends for getting me to where I am. Their love and support has kept me going through my tough years at MIT.
# Contents

1. **Introduction to StarLogo Nova** 21
   1.1 Introduction ............................................. 21

2. **Motivations and Contributions to StarLogo Nova** 23

3. **Discussion Overview** 25

4. **Technical Background on StarLogo Nova** 27
   4.1 Technical Terminology ................................. 27
      4.1.1 Blocks ............................................... 27
      4.1.2 Arguments ............................................ 29
      4.1.3 Breeds and Traits ................................... 32
      4.1.4 Drawer ............................................... 33
      4.1.5 Widgets .............................................. 33
   4.2 User Interactions with the Editor ...................... 36
      4.2.1 Dragging and Connecting Blocks ..................... 36
      4.2.2 Deleting and Copying Blocks ....................... 39
      4.2.3 Argument-Related Interactions ..................... 41
      4.2.4 Breed-Related Interactions ......................... 45
      4.2.5 Trait-Related Interactions ......................... 45
      4.2.6 Widget-Related Interactions ....................... 48
      4.2.7 Series-Related Operations ......................... 49
   4.3 Current Implementation of StarLogo Nova ............... 49
4.3.1 Outline ......................................................... 49
4.3.2 ScriptBlocks ................................................. 51
4.3.3 WLWidgets .................................................... 52
4.3.4 WebLogo ....................................................... 52
4.3.5 WLSite .......................................................... 53

5 Problem Background ........................................... 55
5.1 Terminology – Operations and Edits .......................... 55
5.2 Single-User Undo/Redo Problem Statement .................. 55
5.2.1 Related Works ............................................... 56
5.3 Real-time Collaboration Problem Statement ................ 58
5.3.1 Related Works ............................................... 61
5.3.2 Clarification .................................................... 64

6 Abstract Single-User Undo/Redo System ....................... 65
6.1 Design ............................................................. 65

7 Real-time Collaboration .......................................... 67
7.1 Design ............................................................. 67
7.1.1 Merging Algorithm .......................................... 68
7.1.2 Communication Layer ...................................... 81
7.1.3 Session Manager ............................................. 82
7.2 Implementation .................................................. 82
7.2.1 Fixing IDs ..................................................... 82
7.2.2 Operations and Edits ........................................ 83
7.2.3 Undoing an Edit .............................................. 85
7.2.4 Vector Timestamp Comparison ............................. 86
7.3 Test Results ....................................................... 86
7.3.1 Block Tests ..................................................... 86
7.3.2 Procedure-Related Tests ................................... 91
7.3.3 Breed and Trait-Related Tests ............................. 92
7.3.4 Widget-Related Operations

8 Efficient Undo/Redo

8.1 Analysis of Past Work

8.2 Design

8.3 Implementation

8.3.1 UndoMethodBuilder

8.3.2 Applying the UndoMethodBuilder

8.3.3 Library Function Calls

8.3.4 Refactoring to Foundation Functions

8.3.5 Quirks of sb.Argument.addArgument()

8.3.6 Masquerading

8.4 Single-User Test Results

8.4.1 Procedure-Related Tests

8.4.2 Breed and Trait-Related Tests

8.4.3 Widget-Related Tests

8.5 Multi-user Test Results

8.5.1 Block-Related Tests

9 Intention Preservation with Tombstones

9.1 Design

9.2 Implementation

9.2.1 BlockOrderingTree

9.2.2 Applying the BlockOrderingTree

9.2.3 Undo/Redo

9.3 Test Results

9.3.1 Inserting after a Concurrently Deleted Block

9.3.2 Undoing Creation of a Block that is Inserted After It

9.3.3 Inserting into a Concurrently Deleted Stack

9.3.4 Connecting to a Concurrently Deleted Parent Block
List of Figures

1-1 Block-based programming environment. On the left of the page, where there is a gray background, is the drawer containing blocks that a user can drag to create more blocks. The tan space to the right is a page where blocks are placed to form a program. The currently shown page is the “The World” page. Clicking on the other tabs would switch over to showing those pages. To the top right of a page, we can see some more icons for features like copying, cutting, pasting, and trashing blocks.

1-2 Visual interface for learning about a system, or a game. There are numerous buttons for setting up and changing the state of a simulation. The graph shows statistics about the simulation. The green space to the right shows the actual simulation.

4-1 Boolean block examples

4-2 List block examples

4-3 Generic data type block examples

4-4 Command block examples

4-5 Before-after relationship

4-6 Arguments with values set via text

4-7 Parent-child relationships. The inner blocks are the children

4-8 Examples of list arguments. Each dropdown is a list argument. The options shown for the create-do block are based on the existing breeds.

4-9 Example of button arguments. Each of the squares with a “-” or “+” is a button argument.
4-10 Breeds and traits. The top row shows the trait UI when it is hidden, while the bottom row shows the trait UI when it is shown.

4-11 Blocks whose arguments are related to breeds or traits. The options for each argument are fetched based on factors like traits of a breed, the list of breeds, and more.

4-12 Drawer with the “Agents” category shown.

4-13 Information widgets. Top left shows a graph widget. Top right shows a table widget. Bottom left shows a label widget. Bottom right shows a databox widget.

4-14 Control widgets. Top left is a push-button widget. Top right is a toggle-button widget. The bottom is a slider widget.

4-15 Blocks whose arguments are related to widgets. The options for each argument are fetched based on the existence of widgets of each type.

4-16 Dragging a block from a drawer. We started a dragged on the delete-agent factory block which creates a delete-agent block.

4-17 Dragging a block from an existing stack. This breaks off the stack and makes us drag the entire lower stack. The if block is currently being dragged.

4-18 Dropping off the stack of blocks in our Figure 4-17 example. The stack is dropped onto the delete-agent block within our on-collision-with block.

4-19 Possible connections. The first row shows an after-connect, child-connection, child-connection, and before-connection, in that order. The second row shows that you can do an after-connection and child-connection with a stack of more than one block, but a before-connection is not possible.

4-20 Hovering our dragged blocks above the trash-icon. The icon turns red to indicate that it being hovered over. Dropping the blocks here will destroy them.
4-21 Outcome from using **delete** key on the first if block inside the **on-collision-with** block from our Figure 4-18 example. ................................. 39

4-22 Outcome from using **Ctrl+X** key combination on the first if block inside the **on-collision-with** block from our Figure 4-18 example. ................................. 39

4-23 Right-click menu. The menu has three options: “Duplicate”, “Cut Blocks”, and “Copy Blocks”. ................................................................. 40

4-24 Copying and pasting a block. The first if block in the **in-collision-with** block has been copied and then pasted. Now the clone is being dragged. 41

4-25 Changing argument value. The first block is changing an argument’s value by inserting a block. Remember that child slots are arguments even though they do not look like a typical one. The second block is the same but for a different argument. The third block changes a list argument’s value by selecting “Turtle” for it. Lastly, the last block changes an argument value with text. ................................. 41

4-26 Adding/removing parameters from procedure block. First block is a block in which nothing about its parameters has been changed. Second block is a block in which two parameters has been added. Third block is a block in which three parameters have been added and the first parameter has been removed. ................................. 42

4-27 Side-effects from adding/removing parameters. First row is the initial state. Second row shows state after removing a parameter. After removing parameter named “Parameter 1”, the arguments for the call blocks are removed and blocks in those arguments are removed as well. Third row shows state after adding a parameter. After adding a parameter named (by default) “Parameter 3”, the an argument is added to each call block to correspond with it. ................................. 43

4-28 Difference in number of arguments for procedures with different return types. The left block has a return type of **nothing**, so it does not have an argument for its return value. The right block has a return type of **data**, so it has an argument for its return value. ................................. 44
4-29 The return type of a procedure is changed from data to none. Since this is a valuable example for illustrating side-effects, we call this the return-type-to-none example.

4-30 Breed renaming. First the downwards-facing triangle is clicked to bring up a menu. Rename breed is clicked, and that makes the tab an editable field. The user then edits the name. A blur event will then trigger the rename.

4-31 Warning when trying to rename a breed to an invalid name (existing breed name or empty string).

4-32 Dialog which shows up when a user tries to delete a breed. Here the user tried to delete the breed named “New Name”.

4-33 Trait UI which lets the user work with traits.

4-34 Trait UI after editing.

4-35 Menu for creating widgets.

4-36 Widget editor for a slider. The widget can be renamed by changing the field next to the “Name” label. The other properties of the widget can similarly be set by editing the field.

4-37 Widget editor for a series widget (graph). A series can be renamed by changing the text field. A series can be added by clicking on the “Add Series” button. A series can be removed by clicking on the trash-can icon. The color of the series can be changed by clicking on the color picker icon.

5-1 User 1 sees O1 then O2. User 2 sees O2 then O1. Since changes are typically not commutative, this inconsistent ordering leads to inconsistent state for our users.

5-2 User 1 applies O1 and O2 locally in that order. They should be applied in that order, but User 2 sees O2 then O1 because of out-of-order delivery.
5-3 The intended output string is “orld”, but User 1 sees “ordl” instead.

This is partly a convergence problem, but also an intention preservation problem.

7-1 Undo/do/redo scheme. E1, E2, E3 are Edits in the Change Log. E4 is a remote Edit that arrives and has been determined to be between E1 and E2. E3 has already been undone, and so will be effectively ignored throughout the process.

7-2 Multi-user undo. The Edits are color-coded by the user (ID of user also indicated in parentheses). In particular, User 1 has color orange. It has been notified that User 1 is undoing. As we can see, there is an undo phase (including the target Edit) followed by a redo phase.

7-3 Multi-user redo. The Edits are color-coded by the user (ID of user also indicated in parentheses). In particular, User 1 has color orange. It has been notified that User 1 is redoing. As we can see, there is an undo phase followed by a redo phase (including the target Edit).

7-4 Concurrently inserting after the same block. The first row shows two concurrent inserts. The second row shows the final outcome of both inserts.

7-5 Concurrently inserting child blocks into same argument. The first row shows the state for both users before inserting. The second row shows the state when the two users concurrently try to insert. Lastly, the third row shows the converged state.

7-6 Receiving an Edit during a Drag. User 2 drags a stack of blocks. The first row shows the state when both users are concurrently doing something (but User 1 has finished his Edit and User 2 is in the middle of it). The second row shows the state when User 2 has received User 1’s Edit. Note that User 1 will not receive User 2’s Edit until User 2 has finished the drag.
7-7 Pasting after cutting a stack with a concurrently inserted block. The first row shows the state after User 1 performs his insert. The second row shows the state when User 2 concurrently cuts the state. Lastly, the third row shows what happens when User 2 pastes.

7-8 Cyclic insertion attempt. The rows successively show the steps to attempt a cyclic insertion.

7-9 Inserting after a concurrently deleted block in a stack. The first row shows the concurrent Edits. The second row shows the converged state.

7-10 Created a call-block and adding a parameter to the same corresponding procedure concurrently. The first row shows the concurrent action. The second row shows the converged result.

7-11 Concurrently adding and removing a parameter to a procedure. Notice that since the remove parameter occurred first, the add parameter that follows could potentially use the name of the parameter that was removed. That happens in this example.

7-12 Concurrently renaming two breeds to the same name. The first row shows the concurrent renaming. The second row shows the converged result.

7-13 Concurrently renaming a breed and one of its traits. The first row shows the concurrent renaming. The second row shows the converged result.

7-14 Concurrently renaming a trait and changing its value. The first row shows the concurrent action. The second row shows the converged result.

7-15 Deleting a widget with its widget editor open. The first row shows the state when User 2 has deleted the widget, but User 1 has yet to receive the delete. After getting the delete, both the widget and the editor are removed.
8-1 Example where a representative event (sb.Block.SOME_EVENT) of a function is fired/captured in an awkward place. The representative event implies that this.fieldToReset should be reset, but it is fired after another change that fires events. Since sb.Block.SOME_EVENT is captured last here, this.fieldToReset is undone first, which is not actually applying inverses in reverse order.

8-2 Undoing return-type change of procedure block from data to none. The first row shows the initial state. The second row shows the change.

8-3 Undoing deletion of a parameter from a procedure block. The first row shows the initial state. The second row shows the change.

8-4 Page of the breed that we undo the delete of.

8-5 Undoing trait rename. The first shows the initial state. The second row shows the change.

8-6 Undoing series rename. The first shows the initial state. The second row shows the change.

8-7 Undo creation of block that has another block inserted after it. User 1 inserts the green block. User 2 inserts the red block after User 1’s green block. The first row shows the outcome after these two inserts. The second row shows the outcome after User 1 undoes. Note that User 2 has yet to receive User 1’s undo message.

9-1 The first row shows the setup. The second row shows the actions taken by two users. The third row shows the outcome from using Wantaim. Lastly, the final row shows the expected results.

9-2 Inserting after a concurrently deleted block. User 1 deletes the green block while User 2 concurrently inserts the red block after it. This concurrent action is shown in the first row. The converged outcome is shown in the second row.
9-3 Undoing creation of a block that is inserted after it. The first row shows the state after User 1 and User 2 inserts their blocks. The second row shows the state after User 1 undoes the creation of his block. Note that User 2 has yet to receive User 1’s undo message.

9-4 Inserting into a concurrently deleted stack. User 2 inserts a red block after the green block, but User 1 deletes the stack rooted on the green block. This concurrent action is shown in the first row. The second row shows the final converged outcome.

9-5 Concurrent delete and parent connect. The first row shows the concurrent action of User 1 and User 2. The second row shows the final converged outcome.
List of Tables

7.1 List of Operations for creating and deleting blocks. . . . . . . . . . . 74
7.2 List of Operations for connecting, disconnecting, and dragging blocks. 75
7.3 List of Operations related to arguments. . . . . . . . . . . . . . . . . 76
7.4 List of Operations for breeds and traits. . . . . . . . . . . . . . . . . . 77
7.5 List of Operations for handling widgets. . . . . . . . . . . . . . . . . 78
7.6 List of Operations for handling series. . . . . . . . . . . . . . . . . . 79
Chapter 1

Introduction to StarLogo Nova

1.1 Introduction

StarLogo Nova (also affectionately called StarLogo) is an online educational platform developed by the MIT Scheller Teacher Education Program [7] to enable users to program and visualize 3D systems and games. This an educational endeavor to help students study diverse concepts in science and math and develop computational thinking [6].

StarLogo supports visual learning by giving users a visual interface for understanding the interaction of multiple decentralized agents in a system. This interface is shown in Figure 1-2. In addition to this, StarLogo helps students build skills in computational thinking by letting them implement the systems themselves. To make development of computational thinking the focus of students’ learning, StarLogo provides a high-level, block-based programming environment that facilitates programming without worrying about low-level details [5]. This programming environment is shown in Figure 1-1.

At the time of writing, StarLogo is being used by over 12,000 users and has over 60,000 projects [6].
Figure 1-1: Block-based programming environment. On the left of the page, where there is a gray background, is the drawer containing blocks that a user can drag to create more blocks. The tan space to the right is a page where blocks are placed to form a program. The currently shown page is the “The World” page. Clicking on the other tabs would switch over to showing those pages. To the top right of a page, we can see some more icons for features like copying, cutting, pasting, and trashing blocks.

Figure 1-2: Visual interface for learning about a system, or a game. There are numerous buttons for setting up and changing the state of a simulation. The graph shows statistics about the simulation. The green space to the right shows the actual simulation.
Chapter 2

Motivations and Contributions to StarLogo Nova

The StarLogo development team is always aiming to provide new features to improve user experience. A feature that has surprisingly been missing from StarLogo is undo/redo in the block-based programming environment. This feature is usually taken for granted in editors since it is so ubiquitous and powerful. However, it turns out to not be a simple feature, and its implementation was briefly attempted by a former MEng student Ami Suzuki for a part of her thesis [16]. Hinging myself on her work, I have come up with a design for efficiently undoing and redoing in StarLogo Nova’s block-based programming environment.

In addition to undo/redo, I worked on real-time collaboration (supporting multi-user undo/redo). Given popular productivity tools like Google Docs [3] available today, it is not difficult to imagine how real-time collaboration could benefit our users from a productivity aspect. However, collaborative editing has educational benefits as well. It touches upon several main practices of computational thinking proposed by Brennan and Resnick: being incremental and iterative (iterating upon each other’s work immediately), reusing and remixing (remixing in real-time), and abstracting and modularizing (splitting work into independently-implementable components) [10]. The feature would add the collaboration dimension to education, which is currently not easily supported through StarLogo.
My work on it is the first attempt within the StarLogo development team. This has previously been tried by Google for their block-based programming library Blockly[1], but they were not able to succeed. My work for this thesis provides the design and implementation of a proof-of-concept real-time collaboration system for StarLogo’s programming environment. This is a direction towards block-based programming environments one day supporting real-time collaboration as an actual feature.

Although single-user undo/redo is not directly implemented, its efficient implementation can be arrived at from simplifying the implemented real-time collaboration system.
Chapter 3

Discussion Overview

The main discussion of this thesis will be divided into the following parts:

1. Technical background on StarLogo Nova
2. Description of undo/redo and real-time collaboration as technical problems
3. Proposed solution to problems

StarLogo Nova’s editor is powerful in that it takes away the details and lets programmers focus on the concepts. This actually makes the editor fairly complex and intricate. To understand the problems and solutions mentioned in this thesis, a decent amount of terminology and understanding of the functionalities provided by the StarLogo Nova editor will first need to be understood. Therefore we begin our discussion with a technical background on StarLogo Nova.

The next part of our discussion will explain in more detail what “undo/redo” and “real-time collaboration” even mean as problems. We will also discuss some of the related works and see how applicable they are in the context of StarLogo Nova. Ideas from these related works will help us gain intuition or building blocks for the designs presented in this thesis.

Lastly, we will discuss the system designs created for solving the problems. The first design we discuss is an abstract, straw-man system for single-user undo/redo. This is a system that does not explain how changes are actually undone (a problem which we call Edits). Rather, it describes the general process for what happens when
a user makes changes or undo/redoes. This will ease our way into understanding the types of problems that need to be dealt with that are specific to StarLogo. The design for efficient Edits will be discussed in a later chapter. Note, however, that the design of a real-time collaboration system supporting multi-user undo/redo is heavily influenced by the constraints imposed by undo/redo. In other words, our Edits design might not necessarily conform to the workings of our real-time collaboration system. For this reason, we will describe our real-time collaboration system before the efficient Edits design, and, in describing it, provide rationalization for the overall design and what constraints this has on Edits. As it turns out, we are fortunate that there ends up being no constraints. This is by no means obvious, but real-time collaboration and Edits turn out to be two modular components that can be designed independently. We will then describe our efficient Edits design. Lastly, we patch up our initial real-time collaboration system design with an improvement. The initial design does not actually provide strong guarantees on intention preservation. We will use a technique called tombstoning to provide what we believe are fairly complete guarantees on intention preservation.
Chapter 4

Technical Background on StarLogo Nova

4.1 Technical Terminology

4.1.1 Blocks

The basic unit of programming in StarLogo Nova is the block. They are analogous to abstract syntax tree nodes (AST nodes) in program analysis. Blocks are classified into several data types, and are shaped accordingly so that this static typing information is visually presented. The current supported types are:

1. **Internal (knob)**. In programming languages, these blocks are equivalent to an expression. These blocks have a knob to the left side to mark the kinds of child-connection they can belong to. These blocks are classified into even more specific types based on the shape of the knob.

   - **Boolean (half-circle)**. These represent boolean expressions, and have a half-circle knob shape. Example boolean blocks are shown in Figure 4-1.
   - **List (triangle)**. These represent lists, and have a triangular knob shape. They are actually not fully supported, so they will not be considered too much. Example list blocks are shown in Figure 4-2.
• **Generic data types** (square). These represent all other expressions, and have a square knob shape. The value could be a number, agent, time, or many other objects. Example generic data type blocks are shown in Figure 4-3.

2. **Command** (flat). In programming languages, command blocks are equivalent to a statement or function declaration. These blocks can also be further specified by the types of connections that they have. A command block can either have: 1. before and after connections, 2. only a before connection, or 3. no connection at all. A command block with a before connection can be linked after another command block with an after connection. A command block with an after connection can have another command block (with a before connection) come after it. Examples of these are shown in Figure 4-4. Blocks with only a before connection are also known as *terminator blocks*. 

---

**Figure 4-1: Boolean block examples**

**Figure 4-2: List block examples**
4.1.2 Arguments

Blocks can be connected to each other in insertion points referred to as sockets. Blocks with a before connection have a corresponding before socket. Blocks with an after connection have a corresponding after socket. Command blocks with the necessary connections can be inserted at these sockets. These connections suggest a sibling relationship and are shown in Figure 4-5. This is in contrast to a parent-child relationship that arises from inserting into sockets corresponding to arguments.

Arguments are yet another component of a block. It is difficult to precisely define what an argument is, but in general they do as their name suggests. Most arguments are to blocks like parameters are to a function – they specify additional information. For example, an add block has two arguments – one for each addend. Similarly to parameters for a function, arguments work by having their values set. In the add block example, a user can set the value of the two addends by typing in two num-
This usage of an argument is shown in Figure 4-6. Alternatively, the user could insert expression blocks (with generic data type) into the arguments; the argument’s value is thus set to the inserted block. The types of block insertion allowed into an argument depend on the argument’s socket type. If the argument has an internal socket type, then internal blocks with a matching shape with the argument can be inserted. If the argument has a command socket type, then command blocks can be inserted into the argument using the command blocks’ before connection. This type of connection forms a parent-child relationship (where the inserted block is the child and the block owning the argument is the parent). Figure 4-7 examples of this type of block relationship.

A different type of argument called list argument has its value set via a dropdown menu (also called selection). The options for the selection are updated based on the block and context of the project. For example, a create-do block has a list argument for selecting the breed. The options come from the list of user-created breeds. Figure 4-8 shows an example of a list argument.

The last type of argument is a button argument. A user cannot change the value of a button argument. Instead, a button argument exists to let users be able to click on it. For example, a user can add or remove parameters from a procedure block. This is done by clicking on the appropriate button arguments. Figure 4-9 shows examples of button arguments.
Figure 4-8: Examples of list arguments. Each dropdown is a list argument. The options shown for the `create-do` block are based on the existing breeds.

Figure 4-9: Example of button arguments. Each of the squares with a “-” or “+” is a button argument.
4.1.3 Breeds and Traits

A program can be implemented across more than one page of code. Each page has the code for a breed. A breed is essentially the equivalent of a class in most programming languages. There are two special breeds that exist by default called The World and Everyone. The World is analogous to a static class, where all methods and fields are static. Everyone is essentially a base class by which all newly created breeds extend. This is the only case in which StarLogo Nova currently supports inheritance.

Just like a class, breeds have an analogy to class fields called traits. The World and Everyone breeds do not come with default traits. However, every custom breed that is created has default traits: shape, color, and size. These are fixed traits whose values can change, but they cannot be deleted or renamed. An element called the trait UI that can be hidden or shown holds the display of a list of traits for the current page’s breed. Figure 4-10 shows breeds and the trait UI for both cases of when the trait UI is hidden or shown.

There are a number of blocks whose arguments are related to breeds and traits. These examples are shown in Figure 4-11. The options in the list arguments are based on existing breeds, or traits of breeds.
Figure 4-11: Blocks whose arguments are related to breeds or traits. The options for each argument are fetched based on factors like traits of a breed, the list of breeds, and more.

4.1.4 **Drawer**

We have already seen how blocks can belong on a page or inside other blocks. Lastly, a block can belong in the **drawer**. A drawer is the scrollable container on the left of a workspace which holds all of the **factory blocks** from which we can drag and create more blocks. Figure 4-12 shows a view of the drawer.

4.1.5 **Widgets**

A **widget** is one of the tools for interfacing with the program runtime. These all reside in a space called the **widget space**. Widgets can be divided into two classes: **information widgets** and **control widgets**. Information widgets are used to display information about the runtime. These are: the **label** widget, **databox** widget, **graph** widget, and **table** widget. Figure 4-13 shows all the types of information widgets. Control widgets are used to control the program runtime state. These are: the **push-button** widget, **toggle-button** widget, and **slider** widget. Figure 4-14 shows all the types of control widgets.

There are also a number of blocks whose arguments refer to widgets. These examples are shown in Figure 4-15.
Figure 4-12: Drawer with the “Agents” category shown.
Figure 4-13: Information widgets. Top left shows a graph widget. Top right shows a table widget. Bottom left shows a label widget. Bottom right shows a databox widget.

Figure 4-14: Control widgets. Top left is a push-button widget. Top right is a toggle-button widget. The bottom is a slider widget.

Figure 4-15: Blocks whose arguments are related to widgets. The options for each argument are fetched based on the existence of widgets of each type.
Figure 4-16: Dragging a block from a drawer. We started a dragged on the \textit{delete-agent} factory block which creates a \textit{delete-agent} block.

4.2 User Interactions with the Editor

4.2.1 Dragging and Connecting Blocks

The most fundamental feature of StarLogo is that we can compose programs by dragging and connecting blocks. We can drag a block from the drawer to create a new block. This block can then be connected to a number of locations. It could be connected onto the page or inside another block depending on whether or not the sockets match. Figure 4-16 shows an example of dragging from the drawer.

Dragging a block from a stack of blocks will split the stack into two where our target block is, and let us to drag the bottom-half stack. Figure 4-17 shows an example of this.

For completeness, Figure 4-18 shows the same example where the dragged stack is finally dropped onto a connection.

There are actually four different ways in which a dragged stack of blocks can be connected. They can be connected after a block, as a child of a block, onto a page, or before a block. Connecting after a block is only possible for command blocks with a before-connection. Connecting as a child of a block is possible only if it fits the socket-type for an argument. Connecting onto a page is always possible. Connecting before a block is only possible when dragging a single-block stack with an after-connection,
Figure 4-17: Dragging a block from an existing stack. This breaks off the stack and makes us drag the entire lower stack. The if block is currently being dragged.

Figure 4-18: Dropping off the stack of blocks in our Figure 4-17 example. The stack is dropped onto the delete-agent block within our on-collision-with block.
Figure 4-19: Possible connections. The first row shows an after-connect, child-connection, child-connection, and before-connection, in that order. The second row shows that you can do an after-connection and child-connection with a stack of more than one block, but a before-connection is not possible.

Figure 4-20: Hovering our dragged blocks above the trash-icon. The icon turns red to indicate that it being hovered over. Dropping the blocks here will destroy them.

and only to a command block with a before-connection that is connected to the page. Possible connections are indicated in the UI by a yellow-orange highlighting of the socket area. Figure 4-19 shows examples of possible connections.

Blocks can also be deleted by dropping it to the right locations. Dropping blocks into the black-space above the workspace will destroy the dragged blocks. Dropping blocks into the trash-can icon on the top-right of the workspace will also destroy the blocks. Figure 4-20 shows how the UI indicates that blocks will be trashed when dragged above the trash-can icon.
4.2.2 Deleting and Copying Blocks

Besides dragging blocks to the trash-can or black space, there are other ways to delete blocks. Blocks can be deleted by focusing on the block (clicking on it) and using the delete or Ctrl+X key shortcut. The delete shortcut will isolate out a block from a stack and delete it (including its children). Figure 4-21 shows an example of what happens when the delete key is pressed. When the Ctrl+X key shortcut is used, the stack from the focused block and below is destroyed (and it is copied). Figure 4-22 shows an example of what happens when the Ctrl+X key combination is pressed.
Figure 4-23: Right-click menu. The menu has three options: “Duplicate”, “Cut Blocks”, and “Copy Blocks”.

In order to make block deletion a more accessible feature to our users, there are two more ways to do this. One way is to focus on a block and then clicking on the scissor-icon on the top-right of the workspace (also labeled as “cut”). Clicking on this has the same effect as doing a Ctrl+X. The other way is to right click and then click on the “Cut Blocks” option in the resulting menu. This also has the effect of a Ctrl+X. Figure 4-23 shows the menu from a right click.

As the right-click menu shows, we can also copy and paste blocks. Unlike deleting, we can only copy a stack of blocks (i.e. copy the stack below our focused block). The “Copy Blocks” would copy the blocks onto the clipboard so that they can later be pasted. The actual pasting can be done via the Ctrl+V key shortcut. This will create a clone of the stack where the user’s mouse is and set it to dragging state until the user clicks again at a drop point. Not surprising, both clicking the icon on the top-right labeled “Cut”, and using Ctrl+C has the same functionality as clicking on the “Copy Blocks” option. Clicking on the icon on the top-right labeled “Paste” has the same behavior as Ctrl+V. Lastly, the “Duplicate” option from the right-click menu actually has the same behavior as a Ctrl+C and Ctrl+V combination. Figure 4-24 shows what happens when a paste is done.
Figure 4-24: Copying and pasting a block. The first if block in the in-collision-with block has been copied and then pasted. Now the clone is being dragged.

Figure 4-25: Changing argument value. The first block is changing an argument’s value by inserting a block. Remember that child slots are arguments even though they do not look like a typical one. The second block is the same but for a different argument. The third block changes a list argument’s value by selecting “Turtle” for it. Lastly, the last block changes an argument value with text.

4.2.3 Argument-Related Interactions

Arguments can be added, removed, or have their value changed. Argument value changes happen in a number of ways. The user types the value in, inserts a block into it, or changes the selection if it is a list argument. Figure 4-25 shows examples of each of these three.

Adding and removing arguments are primarily triggered by two special blocks: procedure blocks and list blocks. A user can add parameters to the procedure block by clicking on the “+”-labeled argument. This will create an argument for the name of the parameter, an argument for the type of the parameter, and a “-”-labeled argument
Figure 4-26: Adding/removing parameters from procedure block. First block is a block in which nothing about its parameters has been changed. Second block is a block in which two parameters has been added. Third block is a block in which three parameters have been added and the first parameter has been removed.

which can be clicked to remove the parameter. When a “-”-labeled argument for a parameter is clicked, all of the arguments added by clicking on the “+”-labeled argument are removed. Figure 4-26 show examples of what a procedure block could look like after adding or removing parameters.

List blocks do essentially the same thing when a list item is added. The difference is that each list item added also has a “+”-labeled argument so that a list item can be inserted at a specific position.

Adding/removing parameters is more complex than the example shown in Figure 4-26. This is one of the many cases where something known as a side-effect happens. Basically, a side-effect is equivalent to a refactoring in most modern programming IDEs. For example, in these IDEs, when a method name is renamed, refactoring would rename every method call as well. StarLogo, by default, chooses to refactor the program with side-effects.

For adding/removing parameters, this implies that call blocks corresponding to the affected procedure block should also reflect the addition/deletion of the parameters. Each added/removed parameter would map to an added/removed argument in the call block. To make matters worse, a parameter that was removed could have corresponded to an argument that was already set with values via block insertions.
Figure 4-27: Side-effects from adding/removing parameters. First row is the initial state. Second row shows state after removing a parameter. After removing parameter named “Parameter 1”, the arguments for the call blocks are removed and blocks in those arguments are removed as well. Third row shows state after adding a parameter. After adding a parameter named (by default) “Parameter 3”, the an argument is added to each call block to correspond with it.

This means that those blocks are removed as well when a parameter is removed. Figure 4-27 shows an example of this described behavior.

There are some more unusual ways of adding or removing arguments. When we change the return type of a procedure block, it can gain or lose an argument. When the procedure block has been set to return nothing, it will be missing an argument for the return value. When the procedure has been set to return anything else (like data), it will have an argument for the return value. Therefore, argument changes to the return type of the procedure will change the number of arguments it has. Figure 4-28 shows the difference of two procedure blocks when they have different return types. As a matter of fact, the effects of an argument value change is actually greater than this example shows. There could be side-effects where a return-call block is affected from the fact that it can no longer call the procedure. Figure 4-29 shows an example of this. Since we will be referring to this example, often we will be naming it the return-type-to-none example.
Figure 4-28: Difference in number of arguments for procedures with different return types. The left block has a return type of `nothing`, so it does not have an argument for its return value. The right block has a return type of `data`, so it has an argument for its return value.

Figure 4-29: The return type of a procedure is changed from `data` to `none`. Since this is a valuable example for illustrating side-effects, we call this the `return-type-to-none` example.
4.2.4 Breed-Related Interactions

A user can do three things to a breed: rename, create, and remove. To rename a breed, a user clicks on the downwards-facing triangle. This will bring up a menu with “Rename Breed” as an option. Selecting it will make the tab editable. After filling in the field with a new name, a blur event then triggers the rename. Figure 4-31 shows these steps for renaming. Note that a breed cannot be renamed to an existing breed name or the empty string. This will give a warning as shown in Figure ??.

To create a breed, the button labeled “Add Breed” next to the page tabs should be clicked on. This will create a new breed with an automatically generated unique name. Furthermore, once created, the tab is set to be editable so that the user can rename a newly created breed without clicking on the downwards-facing triangle first. Lastly, we have breed deletes. The breed can be deleted by bringing up the menu by clicking on the downwards-facing triangle and then clicking on the “Delete Breed” option. This option will bring up a dialog which confirms whether or not user wants to delete the breed. Of course, clicking the “Yes” button will delete the button. Figure 4-32 shows the dialog when the user wants to delete a breed.

Unsurprisingly, these interactions also come with side-effects. For example, a create-do block that had its argument value set to a breed will have the value changed to match the name of a breed accordingly. Likewise, if the breed is deleted, then the create-do block will set its argument value to null.

4.2.5 Trait-Related Interactions

Traits of a breed can be added, renamed, have their value changed, or removed. All of these interactions are done via the trait UI as shown in Figure 4-33. As expected, clicking on the “Add Custom Trait” button will create a new custom trait. A trait can be renamed by changing the left column. The value of a trait can be changed by changing the right column. Figure 4-34 shows an example of what happens the user adds, renames, or changes the value of a trait.

Removing a trait can be done by clicking on the trash-can icon on its left. This
Figure 4-30: Breed renaming. First the downwards-facing triangle is clicked to bring up a menu. Rename breed is clicked, and that makes the tab an editable field. The user then edits the name. A blur event will then trigger the rename.

Figure 4-31: Warning when trying to rename a breed to an invalid name (existing breed name or empty string).

Figure 4-32: Dialog which shows up when a user tries to delete a breed. Here the user tried to delete the breed named “New Name”.

46
Figure 4-33: Trait UI which lets the user work with traits.

Figure 4-34: Trait UI after editing.
A user can do the following with a widget: create it, rename it, remove it, move it, set its property, and resize it. All of the various types of widgets can be created by selecting the desired widget from opening up the menu under the “Create Widget” button in the widget space. Figure 4-35 shows the menu from which the desired widget can be chosen to be created. After creating a widget, it can be moved by dragging it with the mouse. Pulling at the ends of the widget can be used to resize the widget.

A widget can be renamed or have its properties set with its widget editor. This widget editor is opened by clicking on the widget. This will show properties of the widget which can be edited on the editor. Figure 4-36 shows an example of the widget editor.

Lastly, a widget can be deleted by clicking on the “Delete Widget” button from the widget editor. A dialog box will be shown requesting confirmation on whether or not the user wants to delete the widget (just like for breeds and traits). Clicking the “Yes” button will complete the deletion of the widget.

Furthermore, as we can see above the widget space, there is a “Clear All” button. This button can be used to clear all of the widgets currently on the space.
4.2.7 Series-Related Operations

There are a few specific widgets called series widgets (table widget and chart widget) which hold series. With these widgets, a user can also add a series, remove a series, or change the color of the series. These changes are done via the widget editor of the series widget. Figure 4-37 shows how this editor looks for a series widget.

4.3 Current Implementation of StarLogo Nova

4.3.1 Outline

StarLogo Nova is a huge project created with multiple repositories. It builds on top of the ScriptBlocks library which is used for making other block-based programming environments. The library provides block rendering, block dragging/connection logic, and interface methods for modifying/fetching properties of blocks, drawers, pages, and arguments. However, the ScriptBlocks library lacks any understanding of a programming language. For example, the ScriptBlocks library does not handle the side-effects associated with changing a procedure block’s return type. It is only aware of changing the argument value just for the return type list argument, and,
Figure 4-37: Widget editor for a series widget (graph). A series can be renamed by changing the text field. A series can be added by clicking on the “Add Series” button. A series can be removed by clicking on the trash-can icon. The color of the series can be changed by clicking on the color picker icon.

for example, will not change the selection in return-call blocks currently selecting the procedure to null. To make ScriptBlocks know how to propagate the effects of the change, the WebLogo repository maintains programming language logic and calls ScriptBlocks to make the remaining changes. WebLogo basically serves as the main client-side code for StarLogo Nova.

Widgets and widget space are implemented in the WLWidgets repository. This is effectively another library just like ScriptBlocks. It handles widget UI, widget dragging, and interface methods for modifying/fetching properties of widgets. WebLogo serves as a channel between WLWidgets and ScriptBlocks so that blocks can reference and react to changes in widgets.

Lastly, the server-side is implemented in WLSite. The server maintains a database for users, projects, as well as handles a number of other features. The most relevant part of the server for this thesis is it stores projects and sends it to the client when a project is loaded.

StarLogo Nova also has a separate repository for handling the rendering in the
program output window. Since this is completely independent of the work done in this thesis, we will not discuss it. The following sections will go into more specifics for the relevant parts of the other aforementioned repositories.

4.3.2 ScriptBlocks

ScriptBlocks is a library that holds the interface to blocks – basically providing the logic for a workspace skeleton. This includes block rendering, block dragging/connection logic, and interface methods for modifying/fetching properties of blocks, drawers, pages, and arguments. In a bare-bone application built with ScriptBlocks, a user would be able to drag blocks, view different pages, scroll the drawer, and change argument values (via typing, selection, or block insertion). ScriptBlocks would detect all the user input and make simple changes to model, view, and controller (to the extent in which they exist in the implementation) in accordance with the input. However, the library by itself does not directly implement the side-effects. For example, ScriptBlocks does not directly change return-call block selections when a procedure block changes return type to nothing; it would change the return-type argument value to nothing but do no more. Side-effects are handled indirectly by firing an event (sb.ArgumentEvent.VALUE_CHANGED in this case), to inform the main application of the fact that an argument has changed value and of all associated details such as which argument and to what value. The main application listens for these events and can then decide the appropriate side-effect behavior. In our example, the main application would then know to call the ScriptBlocks library function for changing the argument value of all return-call blocks that had the value set to refer to the procedure block to instead refer to null.


The classes sb.Block, sb.Argument, sb.ListArgument, sb.Page, sb.Drawer, and sb.Workspace represent the fundamental objects in ScriptBlocks and StarLogo. They have methods for modification fetching and modification. For example, we can
connect a block after another by calling `sb.Block.connectAfterBlock()`.

The `sb.ViewManager`, `sb.BlockDragDropManager`, and `sb.ObjectManager` classes are all manager classes that maintain some global state. The `sb.ObjectManager` keeps track of all of the objects above by mapping the object ID to a reference to the actual object itself. Objects deleted or created go update the `sb.ObjectManager`. The `sb.ViewManager` updates the view. This listens to events triggered and then update the view accordingly. Along with `sb.BlockDragDropManager`, it also handles the dragging logic.

### 4.3.3 WLWidgets

WLWidgets is essentially ScriptBlocks but for widgets. The main difference is that it has access to some minimal part of WebLogo and was never designed for any other application, and therefore is not a pure library. It handles widget UI, widget dragging, and interface methods for modifying/fetching properties of widgets. Modifications on widgets should trigger side-effects. For example, renaming a widget should also rename all of its uses as an argument value in blocks. Instead of using event firing to let WebLogo handle side-effects, WLWidgets adds listeners to block arguments. This side-effect behavior is defined entirely in WLWidgets.

### 4.3.4 WebLogo

WebLogo is the main client-side code. It holds a number of managers for maintaining procedures, breeds, traits, variables, lists, and other event-listening purposes. The most major manager here is the `sb.ProcedureManager`.

The `sb.ProcedureManager` keeps track of every `procedure` block, `call` block, and `return–call` block. Changes to a procedure or call block causes the `sb.ProcedureManager` to update the listeners of the blocks. Basically, this is responsible for handling the side-effects associated with parameters adds and removals, and return type changes.
4.3.5 WLSite

WLSite is the server-side code. The relevant piece of WLSite is quite small. It has a route handler for project loading GET requests, and another handler for project saving PUT requests.
Chapter 5

Problem Background

5.1 Terminology – Operations and Edits

Before describing related works, I would like to first bring up an important terminology difference used in this thesis versus other papers. StarLogo Nova is unlike typical applications in that not every instant of user input corresponds to an intended change. For example, there is no intended change completed while a user drags a block. The change is only complete and intended when the user drops the block somewhere. Many academic papers assume that intended changes happen at every user input, and they refer to these changes as “Operations”. For us, we call an intended change an Edit (not to be confused with Edits which is a design on how to undo/redo a change). An Edit is composed of changes that are generated at every user input called an Operation. Readers should take care not to be confuse our usage of the term “Operation” with usage of it by other papers. For this chapter alone, we use both definitions of Operation, but hopefully either the context is sufficient or the very subtle difference will make the distinction not matter.

5.2 Single-User Undo/Redo Problem Statement

Our definition of the single-user undo/redo feature is not that different from the behavior expected in most editors. Undo should revert the effects of the last applied
change so that the state is the same as it was before the change. Redo should revert the effects of the last undo so that state is the same as it was before the undo. That is essentially our goal, and we avoid foolish attempts at trying to specify our problem even further – especially since our specification is actually very loose. We instead mention some sample behavior which opens up more potential solutions that might not have been thought of. In general, we expect that the view the user sees should always be updated to reflect the state of the project after undo/redo. However, this does not necessarily mean that we have to, or even should, restore back to the same exact view. Consider text editors, for example. The position on the page we are scrolled to is not restored on undo/redo. Instead, the typical handling is to scroll the page enough to see the change. For us, this is not a requirement, but a nice bonus feature. In fact, we do not necessarily even guarantee that blocks end up exactly where they were before. Our goal is primarily to give the user the sense that the project state has been reverted on undo. Still, we should still make the necessary UI updates so that the user cannot perform illegal actions and break the state of the project. For example, keeping the widget editor open for a widget whose creation is undo would allow a user to fiddle with a non-existent widget. Lastly, one of our few requirements is efficiency. As we will see in the following section 5.2.1, there are simple solutions to undo/redo that are inefficient. Our goal for Edits is such that undo/redo can be roughly as fast as the change it affects.

5.2.1 Related Works

A common approach to solving the undo/redo problem is the command pattern [4]. In the command pattern, changes that a user makes are encapsulated into an object containing enough information to define the change. This lets us record changes and be able to redo them. In many editors, this also turns out to be enough information to undo. A general solution can therefore be described rather simply. Have an undo stack for holding past changes, and a redo stack for holding past undoes. To undo, the undo stack is popped, the popped change is undone, and the change is pushed onto the redo stack. To redo, the redo stack is popped, the popped change is redone,
and the change is pushed onto the undo stack. Along with a few more clean-up steps, this more or less suffices as an undo/redo design for simple applications.

For more complex applications, which StarLogo Nova happens to be, the information does not turn out to be enough to describe undo. For example, it should be easy to specify which breed to delete, but recovering the breed after deletion requires caching information like the blocks that were on the page.

A solution for situations like is the memento pattern. The entire state before applying a change is stored. This is a great technique for changes that cannot be inverted. For example, it appears that the Photoshop team uses the memento pattern since operations like blur are irreversible [4]. This technique could also be applied to StarLogo.

The problem with the memento pattern is that it is slow for large projects. Sometimes changes in StarLogo are small. Ideally, an undo would not require re-loading the entire project from a stored state. Furthermore, many of the changes in StarLogo Nova actually seem invertible. The only challenge is that we require bookkeeping. For example, consider our return-type-to-none example ?? . It seems that we just need to figure out where to change the value of an argument back, where to re-create a block, and where to re-connect blocks as a child. If we can keep smaller constituent changes that are invertible, then we can undo the entire change.

There have been two major past works in this direction: 1. Google on their block-based programming library called Blockly [1], and 2. initial work on Edits for StarLogo Nova in Suzuki’s Masters thesis [16]. Google has managed to successfully implement undo/redo for their system. Suzuki approached the problem similarly to Google, but did not manage to complete undo/redo for all potential user interactions. Both of them noticed that side-effects were indicated in the form of an event. For example, removing blocks from an argument necessarily fired an sb.BlockEvent.CHILD_DISCONNECTED event, and changing an argument value necessarily fired an sb.ArgumentEvent.VALUE_CHANGED event. The key idea is therefore to capture these events to represent the constituents of a change. Undo can then be implemented as applying inverses of the constituents in reverse order. However, due
to the designed architecture of StarLogo, Suzuki found it confusing and fragile.

5.3 Real-time Collaboration Problem Statement

Real-time collaboration can be thought of as providing the illusion to users that they are all editing on the same document. When a user is editing locally, changes should be administered to the document immediately. Furthermore, changes received from remote users should be merged onto the document seamlessly. This is basically the goal of our real-time collaboration system.

The technical conditions for solving real-time collaboration are more involved and complex. According to Sun et al. the three properties that real-time collaboration needs to guarantee are: convergence, causality preservation, and intention preservation. Convergence refers to the condition that workspaces with all changes applied should arrive at the same state. Causality preservation refers to the condition that if change B was first generated on a workspace with already change A applied, then change B must also be applied after change A for every site. It is not surprising why these are the properties we need to guarantee. Consider for example, a real-time collaboration system which simply applies a received change to the local workspace. Now say User 1 does Operation O1 and User 2 does Operation O2. User 1 will apply the Operations in the order O1, O2 while User 2 will apply the Operations in the order O2, O1. Since we cannot in general assume that Operation composition is commutative, the two users have a diverged workspace. Figure 5-1 Now say instead that User 1 does two Operations O1 and O2. If O2 arrives to User 2 before O1, then we have causality violation. Figure 5-2 illustrates an example of the causality violation problem.

Convergence and causality preservation are solved problems for the most part. Their solutions are explained in the following section 5.3.1

The last property Sun mentions, intention preservation, is significantly more interesting, and is a focus point of many academic papers, including this thesis.

Intention preservation, as one would expect, is the property that changes each
Figure 5-1: User 1 sees O1 then O2. User 2 sees O2 then O1. Since changes are typically not commutative, this inconsistent ordering leads to inconsistent state for our users.

Figure 5-2: User 1 applies O1 and O2 locally in that order. They should be applied in that order, but User 2 sees O2 then O1 because of out-of-order delivery.
Figure 5-3: The intended output string is “orld”, but User 1 sees “ordl” instead. This is partly a convergence problem, but also an intention preservation problem.

have their “intentions preserved” even after merging. As far as I am aware, there is no good general definition for intention preservation – most of them say essentially what is expected. Since the notion of intention will vary from application to application, it might not even be possible to provide a good general definition. There are, however, well-known examples and applications with standard definitions for intention preservation. For example, text editing with only the insert and delete character operations have a standard definition. Intention preservation there means that insertion ordering is preserved regardless of other operations being applied. For example, consider that we are given the string “word”, and two users concurrently modify this string. User 1 deletes “w” while user 2 inserts the character ‘l’ between “r” and “d”. The expected string would be “orld” since the intention is to delete “w” and insert ‘l’ between “r” and “d”. It turns out this behavior is non-trivial to implement. For example, text documents are often represented as an array or list of characters. The change by User 1 would be denoted as \texttt{del(0)} (deleting the character at index 0) and the change by User 2 would be denoted as \texttt{ins(“l”, 3)} (inserting the character ‘l’ at index 3). Applying the changes directly upon receiving them would lead to incorrect behavior. Figure 5-3 shows how this happens.

For StarLogo Nova, there is no standard definition of intention preservation. However, there are reasonable definitions that we can arrive at – some based on common sense and some based on existing standards for similar scenarios. Most of the changes
allowed in StarLogo Nova fall in the category of “don’t do if you can’t” for intention preservation. For example, a user cannot rename a trait that has been deleted by another user, but otherwise simply renaming the trait is intention preserving. As another example, a user cannot set the value of an argument whose owner block was deleted by another user, but otherwise value setting should happen as long as the argument exists. The more troublesome family of changes are those pertaining to blocks. A stack of blocks is analogous to a string of characters. This means that insertion ordering of blocks needs to be preserved. Furthermore, since blocks can be disconnected and reconnected, the relationship between blocks in our tree structure can change in an intractable number of ways. Since a formal proof of intention preservation here is difficult, we focus more on the correctness of several important cases and ensure that broken states cannot be attained. These will be explained more in the sections describing real-time collaboration.

There is another reason why proving intention preservation is especially difficult for us. We would also like to support undo/redo for real-time collaboration, which would further increases the space of possibilities significantly. When only a single user is working on a project, undo/redo should behave the same way as specified for Edits. When there are multiple users, undo should exclude the change done by the undoing user without affecting the changes of other users unless they are dependent. As with intention preservation, the meaning behind this is not entirely clear. However, we at least guarantee that independent changes are preserved.

5.3.1 Related Works

Real-time collaboration has been tackled with a few approaches for the past few decades. One of the most popular approaches that has for real-time collaboration is operational transforms [15].

An operational transform approach typically uses a Change Log (history buffer) for storing changes. Every change that a user does is encoded into something called an “Operation” which is stored in the Change Log.

As mentioned earlier, convergence and causality preservation are relatively simple,
and approaches involving a Change Log typically solve this using a structure called a *vector timestamp*. A vector timestamp is essentially a tuple of $N$ integers used for “time” comparison, where $N$ is the number of user sites. Every user locally keeps track of a vector timestamp where each integer at index $i$ of the tuple is the number of changes from the user at site $i$ that is applied locally. Every time a change is done by user at site $i$, the user increments the count at his timestamp’s $i$-th index and stores it into the change. This timestamp can be used to determine if an Operation arrived earlier than it should. Denote the timestamp $T_O$ as the timestamp of an operation and let $s$ be its source site. Denote timestamp $T_d$ as the timestamp in a destination site. Then the Operation has arrived in order and can be applied locally at site $d$ if and only if

1. $T_O[s] = T_d[s] + 1$ and
2. $T_O[i] \leq T_d[i]$, for all $i \in \{0, 1, ..., N - 1\}$ and $i \neq s$.

Basically, the first condition means that we have not skipped an Operation from site $s$, and the second condition means the same thing for the remaining sites. Operations that are out-of-order are withheld from being applied locally until its preceding Operations arrive. This solves the causality violation problem (i.e. we now have causality preservation). To solve convergence, the timestamps are used to order the Operations. The sorting key is determined by the sum of the counts in the vector along with the site index from which an Operation originates is used as a tiebreaker for equal sums. Since this gives a total ordering, Operations received can have a determined insertion point into the Change Log. Applying this change to satisfy the ordering maintained by the Change Log can be achieved with an *undo/do/redo* scheme. First, all proceeding Operations (relative to the insertion point) are undone, then do the received Operation, and lastly redo all of the proceeding Operations. Since a single ordering is enforced and each workspace reflect the state from applying changes in that ordering, convergence is handled. [13]

The power of operational transforms is its ability to deal with intention preserva-
tion. As the name of the technique suggests, it transforms Operations so that applying the new Operation satisfies intention. Recall our example where we have a starting string “word” and User 1 does \texttt{del(0)} and User 2 does \texttt{ins(‘l’, 3)}. If the deletion occurs first, then operational transforms the insertion such that it instead inserts at index 2. In other words, the shift in index is compensated for via a transformation.

Another popular approach is CRDTs (conflict-free data types). CRDTs do not inherently try to deal with intention preservation. Instead, it has mathematical properties such that Operations applied to the CRDT guarantee convergence and causality preservation. However, there have been successful attempts at implementing CRDTs so that Operations applied to them also preserve intention [14]. One prominent example is WOOT (With-Out Operational Transforms) [13]. Instead of transforming Operations, WOOT cleverly designs the underlying model so that Operations defined on top of that model satisfy intention without requiring transformation.

There is a third approach called Wantaim [12] which does not truly satisfy intention preservation, but instead trades it for simplicity. Wantaim is similar in essence to WOOT in that it also avoids use of operational transforms. This is done by changing the underlying model to be context-based. This way block inserts are not affected by shifts in index (because there are no indices).

Thinking along the lines of the data structure a StarLogo Nova project is implemented in, the WOOT or Wantaim approach is more promising. Operational transforms is more suited for linear structures. Furthermore, it is difficult to even imagine the designs of transforms required to deal with the sweeping side-effects of operations in StarLogo Nova.

With regards to actual implementation attempts at real-time collaboration for block-based programming, Google’s Blockly is the only attempt I am aware of. The implementation had a number of bugs when simultaneously editing on the same block [2]. Also, having personally tried the demo [11] in a non-collaborative environment, it seems that the undo/redo feature is either broken or unsupported. Google tried using the Google Real-time API to implement real-time collaboration. It is unclear how they went about doing this, or what the operational transforms even
were. For reasons brought up earlier regarding why operational transforms is ill-suited for block-based programming environment, it is not surprising to me that the demo turned out buggy. Zero Robotics, which is built with Blockly, also has its own real-time collaboration environment [9], but it is difficult to tell whether or not this is different from Google’s attempt. From personal testing, it was possible to create inconsistencies between editors when applying concurrent operations like inserting to a deleted block.

5.3.2 Clarification

Many of the papers for real-time collaboration are for peer-to-peer networks. For StarLogo, we have a central server instead. Changes from a user are therefore first sent to the server and then relayed to the other users.
Chapter 6

Abstract Single-User Undo/Redo System

6.1 Design

A typical single-user system would use an undo and redo stack as described in 5.2.1. Taking into account compatibility with real-time collaboration, we will instead use a single Change Log. The related works 5.3.1 talk about how generic changes are inserted into the Change Log (albeit for a real-time collaboration setting which we are not concerned about right now). However, StarLogo works with Operations and Edits. We will now explain in specifics how Change Log handling is done for a single-user.

Whenever a user performs a change, Operations corresponding to the change are inserted into the Change Log. If the change is also one that completes user intention, then all of the Operations included in the part of the Edit are removed from the Change Log, and the Edit is inserted in their place.

For undo/redo, we keep in mind that an Edit is the unit of user intention. Therefore, undo/redoes triggered by the user should affect an entire Edit, not just an Operation.

When a user undoes, the Change Log is scanned backwards until we find an Edit that is not marked as undone (which there should none for the first undo). The found
Edit is the one that we want to undo. It is undone and then the Edit is marked as such.

When a user redoes, the Change Log is scanned backwards until we find an Edit that is marked as undone. The found Edit is the one that we want to redo. We simply re-apply the Edit and unmark it.

There are two important edge cases that we have left out. The first of these is something common in many editors. After doing a change (that is neither undo nor redo), the user should not have any changes to redo. In other words, undone changes should be flushed out. We handle this by removing all marked Edits after completing an Edit and inserting it into the Change Log. The second edge case is what happens when a user tries to undo/redo while in the middle of an Edit (i.e., it is storing Operations into the Change Log and has yet to replace them with a representative Edit). We avoid this problem by simply not allowing a user to perform undo/redo while in the middle of an Edit.
Chapter 7

Real-time Collaboration

7.1 Design

Our real-time collaboration system involves three main components: a merging algorithm, communication layer, and session manager. The merging algorithm involves designing our Operations and Edits, as well as determining how Edits and undo/redo changes received from remote users can be merged into the local workspace. The communication layer handles communication between clients and the server. Project initialization for users use this layer by requesting for all changes of a project. The main responsibility of this layer is to combat a slew of network issues such as out-of-order message delivery, duplicate messages, and failed sends. The session manager maintains information about the connected users.

One of the major focus points for this thesis is the merging algorithm – the exploration of unusual states and how they can be handled to preserve intention (or at least get reasonable behavior) is our main interest. Networking and server architecture, on the other hand, will take a back-seat. Although it is no simple task to implement the networking or server architecture, there are numerous ways to deal with networking problems. Furthermore, for the most part, they may be considered mostly independently of the merging algorithm. In discussing the design, we will therefore first describe the merging algorithm under the assumption of no unexpected network disturbances like out-of-order delivery or duplication. The only issue that we consider is
latency. We then lift these assumptions later by explaining the communication layer and session manager.

7.1.1 Merging Algorithm

From the related works \[5.3.1\] and the previous chapter \[9\] we have the basic underpinnings for designing a merging algorithm. On top of the single-user undo/redo system, we now add on support for multi-user functionality. Changes from other users (which get stored on the central server) are assumed to be fetch-able via the communication layer. Just like in the related works \[5.3.1\] vector timestamps are employed for ordering changes. To apply changes in the ordering respected by the Change Log, the undo/do/redo scheme is used. We now go into the specific details of this high-level description.

Workflow

After loading a project (to be explained later), each user starts with a Change Log of Edits and a local vector timestamp matching the number of changes (Edit or undo/redo) received from each user. Furthermore, the user is given by the server an ID to identify the user, and an index corresponding to his index in the vector timestamp.

As usual, the intended changes are Edit and undo/redo. These are therefore the changes that we actually want to send to the server (and then relayed to the other clients). Operations, on the other hand, should only be applied to the local workspace.

When an Operation is done locally, the Operation is inserted to the end of the Change Log. If the Operation completes an Edit, then the Operations part of the Edit are removed from the Change Log and replaced with their representative Edit. In addition to this, the local timestamp is updated so that the value for the user’s index is incremented (to indicate the number of intended changes done by the user). The stored Edit is then also tagged with both updated timestamp and the ID of the user. A message is lastly constructed for the Edit and sent to the server.
Figure 7-1: Undo/do/redo scheme. E1, E2, E3 are Edits in the Change Log. E4 is a remote Edit that arrives and has been determined to be between E1 and E2. E3 has already been undone, and so will be effectively ignored throughout the process.

When an Edit is received from a remote user, the local timestamp is updated based on the Edit’s tagged timestamp. The new local timestamp is such that at each index, the maximum between the old local timestamp and received Edit’s timestamp is used. Using the timestamp of the received Edit, its insertion point in the Change Log is determined such that the Change Log preserves ordering of the Edits by timestamp. The state of the workspace is then changed to reflect the Change Log via the undo/do/redo scheme. Note, however, that the undo/do/redo scheme described in the related works is actually a bit incomplete. During the undo phase, we should take care to ignore Edits that were already undone. During the redo phase, those ignored Edits should be ignored again. Figure [7-1] shows an example of how the undo/do/redo scheme works when receiving a remote Edit.

Recall that in our single-user undo/redo system, Edits marked undone are dis-
Figure 7-2: Multi-user undo. The Edits are color-coded by the user (ID of user also indicated in parentheses). In particular, User 1 has color orange. It has been notified that User 1 is undoing. As we can see, there is an undo phase (including the target Edit) followed by a redo phase.

carded after a user does another Edit. The same applies for multi-user. After knowing that a user has done an Edit (either local or remote), all undone Edits by that user are discarded from the Change Log.

When an undo is done locally, the Change Log is scanned backwards until we find an Edit by the user that is not already marked as undone. This is the Edit that we want to undo. To actually perform this multi-user undo, we follow steps similar to the undo/do/redo scheme, except without the “do”. All proceeding Edits are first undone. Then the found Edit is undone and marked accordingly. Lastly, the proceeding Edits are redone. As with the undo/do/redo scheme, Edits that were already undone should be ignored throughout the process. The local timestamp should be then updated to accommodate for this change. Lastly, a message for the undo is created with the local timestamp and user ID stored, and it is sent to the server.

A remote undo does essentially the same steps as in the local case. A remote undo message contains the ID of the performing user. The only difference from the local case is that we find an Edit done by the user whose ID is indicated in the message. Figure 7-2 shows an example of how multi-user undo works. Notice that the timestamp is not used for undoing since undoes only need to be ordered after their target Edit.
Figure 7-3: Multi-user redo. The Edits are color-coded by the user (ID of user also indicated in parentheses). In particular, User 1 has color orange. It has been notified that User 1 is redoing. As we can see, there is an undo phase followed by a redo phase (including the target Edit).

Local redo is done similarly to local undo. The Change Log is scanned backwards until we find an Edit by the user that is marked as undone. The found Edit is the Edit we want to redo. Again, we do something similar to the undo/do/redo scheme to achieve multi-user undo. All proceeding Edits are first undone. Then the found Edit is redone and marked accordingly. Lastly, the proceeding Edits are redone. Again, we ignore undone Edits (except the one we want to redo) throughout the process. The local timestamp is incremented for this change. Lastly, a message for the redo is created with the local timestamp and user ID stored, and it is sent to the server.

As with undo, a remote redo is basically the same as local redo, but applied to a different user. Figure 7-3 shows an example of how multi-user redo works.

Note that unlike the single-user case, we now need to process Edits or undo/redoes while locally in the middle of an Edit. This is because messages from remote users can be received at any time. In such a scenario, the partial Edit is treated just like a complete Edit (i.e. a pseudo-Edit). Note that this pseudo-Edit does not have a timestamp. To handle this, we pretend that this is an “infinite” timestamp so that the pseudo-Edit should always be re-applied at the end of the Change Log. The remote Edit or undo/redo algorithm then follows through as usual.

The project initialization phase basically fetches all changes from the server. The
changes are then applied in order as if a remote Edit or undo/redo were received.

**Design of Operations**

The design of Operations can be thought of as being intimately tied to efficiently supporting the undo/redo feature for real-time collaboration. Especially for StarLogo Nova, the sweeping side-effects of changes make this easy to think. In particular, it suggests that it is reasonable to design Operations to be as fine-grained as possible. Using the `return-type-to-none` example, its Edit can be represented as multiple Operations: one for argument change on the procedure return type, one for block removal of the `abs` block in the parameter of the call block, one for block removal of `pi` block from the `abs` block, and another argument change for the `return-call` block’s selected procedure. When the Edit containing these Operations is sent over to the other users, not only do the users know how to do these Operations, but they also know how to undo them.

There is a problem with this, however. It is possible that another user has added a `return-call` block elsewhere and made it refer to the affected procedure block. Furthermore, this user, inserts additional blocks to the parameter argument of the `return-call` block. Notice that by having fine-grained Operations, we do not have the ability to adapt to changes by another user. What this instead suggests is that Operations should be designed to be as coarse as possible in order to maintain intention and flexibility in the execution. Using the `return-type-to-none` example, its Edit can be represented as a single Operation for changing the argument value. This Operation would effectively run the same code as what happens in the user-input selection change callback. Even in the situation where another user adds a `return-call` block elsewhere, the side-effects are appropriately propagated and the added `return-call` has its selection set to `null`. However, there is no information on how to undo.

In summary, a fine-grained design makes it easy to invert, but loses intention. A coarse design maintains intention, but cannot undo. However, this turns out to be a mistake. As a matter of fact, the real-time collaboration system we have laid out
so far hints at why. Any Operation should be undoable simply by storing the state of the entire project before executing the Operation. This is basically dynamically computing the undo function after receiving an Edit. In fact, dynamically computing the undo function is exactly what single-user undo/redo is anyway. It follows that we can use a fine-grained design of Operations, and have Edits be designed independently of real-time collaboration.

Based on the above rationalization, we have designed our Operations to be as coarse as possible. A listing of all of the Operations that we have in StarLogo Nova are written into a series of tables. The tables give a high-level description for each Operation and how they are encoded. We will elaborate on how merging works afterwards.

Table 7.1 lists the Operations for creating and deleting blocks. Table 7.2 lists the Operations for connecting, disconnecting, and dragging blocks. Table 7.3 lists the Operations related to arguments. This includes adding/removing parameters and adding/removing list items since parameters and list items is comprised of a group of arguments. Table 7.4 lists the Operations for breeds and traits. Table 7.5 lists the Operations for handling widgets. Lastly, Table 7.6 lists the Operations for dealing with series of graph widgets.

**Merge Behavior**

The description of the Operations listed in the above section 7.1.1 would not be complete without explaining how they are merged into workspaces when faced with concurrent Edits.

In general, we would prefer that the merge be intention-preserving. This turns out to be easily satisfiable for many Operations. It typically makes sense that if the Operation does not have the dependencies it needs, then executing the Operation should be a no-op. For example, the MOVE Operation would do nothing if the block it is trying to move was concurrently deleted by another user.

The main exceptions to this solution are the Operations for block connecting and disconnecting. Consider the situation where User 1 tries to connect block B after
<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLONE</td>
<td>Creates a copy of a block</td>
<td>• ID of the block to be cloned</td>
</tr>
<tr>
<td>DESTROY</td>
<td>Destroys the stack of blocks rooted on this block</td>
<td>• ID of the target block</td>
</tr>
<tr>
<td>DELETE</td>
<td>Deletes a block. Also have a parameter for determining whether or not the entire stack should be deleted. If the parameter is true, then this behaves the same as DESTROY. If the parameter is false, then only the block and its children are deleted</td>
<td>• ID of the target block • Whether or not to delete the entire stack</td>
</tr>
<tr>
<td>PASTE</td>
<td>Pastes a stack of blocks</td>
<td>• Block data (stored as a string) to paste</td>
</tr>
</tbody>
</table>

Table 7.1: List of Operations for creating and deleting blocks.

block A, but User 1 deletes block A concurrently and the deletion is prioritized in the total ordering of vector timestamps. Block B is not inserted since block A does not exist, but the insertion should still happen. This is just like a chain of text, where the insertion of a character should be independent of the deletion of the character it is inserting after.

Wantaim has a particular philosophy in that it takes the simplest approach to merging. In other words, even though it would not satisfy intention preservation to make the Operation a no-op, that is what happens anyway. For now, we will handle block Operations such that they do nothing if their dependent context cannot be found. This will be improved later to handle intention preservation when we get to the tombstone approach augmentation in chapter 9.

Despite targeting simplicity, block connection and disconnection Operations are still quite tricky. The INSERT_AFTER Operation needs to take care to be considered a no-op when the block to insert is a terminator block and there exists an after-block for the target block. This situation arises when a user concurrently inserts into the same target block. The PARENT_CONNECT Operation needs to handle a similar scenario in case two users tried to concurrently insert into the same argument of a parent block. The INSERT_BEFORE Operation needs to be careful for when the target block is re-
## Block Connection and Dragging Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>INSERT BEFORE</td>
<td>Inserts a block before another one</td>
<td>• ID of the block to insert</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ID of the block we want to insert before</td>
</tr>
<tr>
<td>INSERT AFTER</td>
<td>Inserts a block after another one</td>
<td>• ID of the block to insert</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ID of the block we want to insert after</td>
</tr>
<tr>
<td>PARENT CONNECT</td>
<td>Adds a block as a child of another block</td>
<td>• ID of the child block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ID of the parent block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Name of the argument to add to</td>
</tr>
<tr>
<td>PAGE CONNECT</td>
<td>Connects a block to a page</td>
<td>• ID of the block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ID of the page</td>
</tr>
<tr>
<td>DISCONNECT</td>
<td>Disconnects a block. This will disconnect a block from its</td>
<td>• ID of the block to disconnect</td>
</tr>
<tr>
<td></td>
<td>parent (page or block) if it has one. Otherwise, if there is</td>
<td></td>
</tr>
<tr>
<td></td>
<td>a before block, it will disconnect from that.</td>
<td></td>
</tr>
<tr>
<td>MOVE</td>
<td>Moves a block</td>
<td>• ID of the block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New location of the block</td>
</tr>
<tr>
<td>START DRAG</td>
<td>Sets up the dragging of a block</td>
<td>• ID of the block to start dragging</td>
</tr>
<tr>
<td>END DRAG</td>
<td>Ends the dragging of a block</td>
<td>• Does not store anything</td>
</tr>
</tbody>
</table>

Table 7.2: List of Operations for connecting, disconnecting, and dragging blocks.
## Argument-related Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_PARAMETER</td>
<td>Adds a parameter to a procedure block</td>
<td>• ID of the procedure block</td>
</tr>
<tr>
<td>REMOVE_PARAMETER</td>
<td>Removes a parameter from a procedure</td>
<td>• ID of the procedure block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• ID of the parameter</td>
</tr>
<tr>
<td>ADD_LIST_ITEM</td>
<td>Adds an item to a list block</td>
<td>• ID of the list block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Index to insert at</td>
</tr>
<tr>
<td>REMOVE_LIST_ITEM</td>
<td>Removes an item from a list block</td>
<td>• ID of the list block</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Index to remove at</td>
</tr>
<tr>
<td>UPDATE_OPTIONS</td>
<td>Updates the options of an argument value. When the argument has no</td>
<td>• ID of the block with the argument</td>
</tr>
<tr>
<td></td>
<td>value, it sets the value to one of the options.</td>
<td>• Argument name</td>
</tr>
<tr>
<td>ARGUMENT_CHANGE</td>
<td>Changes an argument value. This includes various things like renaming</td>
<td>• ID of the block with the argument</td>
</tr>
<tr>
<td></td>
<td>a procedure, renaming a variable, and changing return type.</td>
<td>• Argument name</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Value to change to</td>
</tr>
</tbody>
</table>

Table 7.3: List of Operations related to arguments.
### Breed and Trait Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_BREED</td>
<td>Creates a new breed</td>
<td>• ID of the breed to create</td>
</tr>
<tr>
<td>REMOVE_BREED</td>
<td>Removes a breed</td>
<td>• ID of the breed to remove</td>
</tr>
<tr>
<td>RENAME_BREED</td>
<td>Renames a breed</td>
<td>• ID of the breed to rename</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Name to rename to</td>
</tr>
<tr>
<td>ADD_TRAIT</td>
<td>Adds a trait to a breed</td>
<td>• ID of the breed</td>
</tr>
<tr>
<td>REMOVE_TRAIT</td>
<td>Removes a trait from a breed</td>
<td>• ID of the breed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Name of the trait to remove</td>
</tr>
<tr>
<td>RENAME_TRAIT</td>
<td>Renames a trait of a breed</td>
<td>• ID of the breed</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Old name of the trait</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New name of the trait</td>
</tr>
<tr>
<td>CHANGE_TRAIT_VALUE</td>
<td>Changes the value of a trait of</td>
<td>• ID of the breed</td>
</tr>
<tr>
<td></td>
<td>a breed</td>
<td>• Name of the trait to change value for</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New value of the trait</td>
</tr>
</tbody>
</table>

Table 7.4: List of Operations for breeds and traits.
<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE_WIDGET</td>
<td>Creates a widget</td>
<td>• ID of new widget</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Type of widget to generate new of widget from</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Type of widget</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• If it is a ChartWidget, whether or not it should be shown</td>
</tr>
<tr>
<td>REMOVE_WIDGET</td>
<td>Removes a widget</td>
<td>• ID of the widget to remove</td>
</tr>
<tr>
<td>CLEAR_WIDGETS</td>
<td>Clears all widgets in the workspace</td>
<td>• Does not store anything</td>
</tr>
<tr>
<td>MOVE_WIDGET</td>
<td>Moves a widget</td>
<td>• ID of the widget to move</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• x-coordinate to move to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• y-coordinate to move to</td>
</tr>
<tr>
<td>RENAME_WIDGET</td>
<td>Renames a widget</td>
<td>• ID of the widget to rename</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New name of widget</td>
</tr>
<tr>
<td>SET_WIDGET_PROPERTY</td>
<td>Changes the property of a widget</td>
<td>• ID of the widget to set property of</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Name of the property to change</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Value to change to</td>
</tr>
<tr>
<td>RESIZE_WIDGET</td>
<td>Resizes a widget</td>
<td>• ID of the widget to resize</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Width to set to</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Height to set to</td>
</tr>
</tbody>
</table>

Table 7.5: List of Operations for handling widgets.
### Series Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Effect</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD_SERIES</td>
<td>Adds a series to a graph widget</td>
<td>• ID of the graph widget</td>
</tr>
<tr>
<td>REMOVE_SERIES</td>
<td>Remove a series from a graph widget</td>
<td>• ID of the graph widget</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Name of the series to remove</td>
</tr>
<tr>
<td>RENAME_SERIES</td>
<td>Renames a series from a graph</td>
<td>• ID of the graph widget</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Old name of the series</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• New name of the series</td>
</tr>
<tr>
<td>EDIT_SERIES_COLOR</td>
<td>Changes the color of a series from a graph</td>
<td>• ID of the graph widget</td>
</tr>
<tr>
<td></td>
<td>widget</td>
<td>• Name of the series</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Color to change to</td>
</tr>
</tbody>
</table>

Table 7.6: List of Operations for handling series.

connected onto a block. For example, we could be trying to insert before a block on a page, but another user concurrently drags the target block into the child socket of another block. The block we are inserting must therefore handle the cases for which the target block could, at insertion time, be a child block, an after block, or a page and still perform the connection. In particular, the connection should form a chain where the target block is located. Lastly, it is possible to come up with situations where a connection Operation might attempt to connect onto a stack in a cyclic manner. As an example, consider the scenario where we start with two blocks A and B where B is connected after A. User 1 drags block B onto the page. After receiving User 1’s change, User 2 then drags block A onto block B. User 1 then gets User 2’s change as well. A cyclic stack insertion is attempted if User 1 undoes here. Recall the phases for performing this multi-user undo. In the first phase, User 2 and User 1’s changes are undo, in that order. This restores the project back to the state where block B is connected after block A. Next, final redo-phase, User 2’s change is redone. This will make block A try to connect onto block B, even though block A is already before block B. These cyclic connections need to be detected, and handled (as a no-op, for
example).

There are a few Operations that can easily be defined to a form that satisfies intention preservation better, but have been defined differently for implementation reasons. The listed Operations involving traits and series refer to a trait or series by name instead of ID. This has the consequence that if another user concurrently renames the trait/series, then an Operation like \texttt{CHANGE\_TRAIT\_VALUE} would have to be interpreted as a no-op since it would then operate on a non-existing trait. The appropriate way to handle this is to have traits be referred to by ID, that way Operations on it will work regardless of the name of the trait. The reason why this is not done is because most of StarLogo is already implemented by referring to traits by name. Anyhow, the Operation as is is not entirely undesirable either.

The list item Operations are quite interesting. As defined right now, these Operations would not make much sense during concurrency. This is precisely the insertion ordering problem for concurrent text-editing \cite{5-3}. The way Wantaim would handle this is to change the model representation for list items to be context-based, and make the Operations refer to the context. In other words, each list item would get an ID, and the \texttt{ADD\_LIST\_ITEM} Operation, for example, would store the ID of the list item to insert after as opposed to the index to insert at. The most appropriate way to handle this would be to use Operational Transforms. Since lists are not supported in StarLogo Nova yet, this is not a problem for us yet. These Operations were included merely for completeness and discussion purposes.

Lastly, we still have to keep in mind that we are actually merging Edits on to workspaces, not single Operations (i.e. we are actually merging a list of Operations to the workspace). For situations where an Operation is decided to be a no-op because its context does not exist, we want to rollback all of the other Operations in the Edit that were already applied to the workspace. We made this decision because an Edit is a unit of intention, and a partial application of it would not include intention; it follows that the entire Edit should no-op as well. A special case happens here when the user is in the middle of an Edit and one of those Operations fails. In that case, the entire partial Edit is rolled back \emph{and} it is discarded from the Change Log.
7.1.2 Communication Layer

There are a number of ways to design the communication layer, and they differ based on the properties of the tools employed. We will simply describe one of the ways in which the communication layer can be done.

For any design of a communication layer, we want to maintain a Change Log on the server as well. This Change Log will reflect all changes sent by every user. The only difference is that this Change Log keeps undo/redo messages as an actual stored element as opposed to a flag on an Edit. After all, we do want to give actual undo/redo messages to users.

In other to get changes, users will ping the server with their current vector timestamp via an HTTP GET request. The server then loops through the Change Log and compares the timestamps to find messages that the user was missing. These messages are then sent back to the user. Even if the response fails, the user can fetch again with their timestamp. Since the user does not know when the server will have changes that he has not received, the user pings the server in a loop with some fixed time interval.

To send changes, the user will maintain a queue of messages. Changes that a user makes (Edit or undo/redo) are inserted into the queue. Once there are messages in the queue, the user will send the messages to the server via an HTTP POST request and wait for a response. This is repeated until the server acknowledges that it has applied the changes to the server Change Log. In the mean time, the user can continue to perform changes and populate the queue. Upon server acknowledgement, only changes received by the server are removed from the queue so that changes yet to be sent can still be sent.

On both the server and client-side, out-of-order and duplicate messages need to be handled. Note that these problems only need to be handled on the server. Assuming that the server has resolved these issues, it would not be possible for the client to receive duplicate or out-of-order messages. As described in the related-works section 5.3.1, out-of-order messages can be detected with the vector timestamps, and
duplicate messages can be detected as well. On the server, out-of-order messages should be withheld in a separate buffer and then later put into the Change Log when its preceding messages have arrived. Of course, duplicate messages are simply discarded.

7.1.3 Session Manager

The session manager for our design is fairly naive. It is basically nothing more than a single number kept on the server per project. The value is the number of users to load the project. When a user tries to load a project, the session manager sends along the stored value to the user, and increments the value after sending to account for the user. The user uses this value as the index into vector timestamp.

7.2 Implementation

7.2.1 Fixing IDs

The original implementation of StarLogo incremented a counter as an ID for objects in StarLogo. Notice that all of the Operations refer to objects in StarLogo via an ID, so it would be troublesome if two different objects shared the same ID. With an incremented-counter ID system, users doing concurrent Operations can easily construct two objects with the same ID. Therefore, ID generation has been replaced with UUIDs.

Also, many of the object-creation Operations should be able to create objects with a specific ID. Again, this is required because objects are referenced by their IDs. For example, if the CLONE Operation generated a block with a different ID, then future Operations received from other users would not refer to the desired block. Previously, IDs were automatically generated and set by calling the constructor of an object. This has been changed so that it could optionally take in an ID parameter which would force the created object to have the specified ID if it exists. Cloning and pasting have also been changed so that the data strings representing the copied blocks includes ID,
and the IDs are replicated on paste.

Some objects were also never referenced by ID. Breeds, for example, had an ID field, but all code would reference it by name instead. The Operations were designed to operate on the ID of a breed instead so that concurrent Operations on a breed are not affected by a BREED_RENAME Operation. It would be too much of an endeavor to try to change all of the existing code to start referring to a breed by ID. Instead, the name of the breed was fetched by ID, and then a function was called based on the name.

7.2.2 Operations and Edits

A vast majority of implementation is for the listed Operations in section 7.1.1. The general process is not too difficult. It is known that almost all of the Operations correspond to the chunk of code inside some user-input listener. For example, breed deletion is triggered by the following event listener (re-written slightly):

```javascript
var self = this;
this.showDialog("Confirm Delete",
"Are you sure you want to delete<br><em>\'" + this.targetBreed + "\'<em>\'" + this.targetBreed + "\'<em>\'",
{
 "Yes": function() {
 $(this).dialog("close");
 Breeds.removeBreed(self.targetBreed);
 },
 "No": function(event) {
 $(this).dialog("close");
 }
});
```

The breed removal code is then replaced with its corresponding Operation, and other Operation-handling code. The above code segment gets changed to:

```javascript
var self = this;
this.showDialog("Confirm Delete",
"Are you sure you want to delete<br><em>\'" + this.targetBreed + "\'<em>\'",
{
 "Yes": function() {
 $(this).dialog("close");
 Breeds.removeBreed(self.targetBreed);
 },
 "No": function(event) {
 $(this).dialog("close");
 }
});
```

83
"Are you sure you want to delete\<br\><\em\>" + this.targetBreed + \\
"\<!\em\>?",

{
  "Yes": function() {
    $(this).dialog("close");
    var operation = new RemoveBreedOperation(this.targetBreed);
    ChangeLog.getInstance().addOperation(operation, true /* isEnd */);
    self.doRemoveBreedOperation(operation); // basically calls
    Breeds.removeBreed(operation.targetBreed);
  },
  "No": function(event) {
    $(this).dialog("close");
  }
});

As we can see, all that is done is creating an Operation corresponding to the effect of a user-input, logging the Operation into the Change Log, and then performing the Operation.

Notice that the `addOperation()` method takes in a boolean parameter called `isEnd`. When set to `true`, this parameter implies that it is the last Operation of an Edit. This will go into the Change Log and replace the Operations with just a single representative Edit. Furthermore, it will also create a message and insert into the client-side communication layer message queue. As it turns out, all Edits are comprised of just a single Operation with the exception of block dragging. It is therefore no surprise that the `true` parameter is used here (Edit for removing breed is just the `REMOVE_BREED` Operation).

Although seemingly straightforward, this process is not as smooth as one would hope. There are a number of places where implementing this turns out to be awkward. Block-dragging is implemented via two separate controllers: `BlockDragDropManager` and `ViewManager`. This is partly the reason for having Operations like `START_DRAG` and `END_DRAG` in addition to the other Operations like `DISCONNECT` and `PARENT_CONNECT`, where it seems there is overlap.

Another thing is that the design rarely takes into account UI, but this is something
that needs to be handled with care. There are typically UI changes that should only apply to the local user, or should only be applied the first time the Operation was executed. One such example is when creating a breed, where the current page of the workspace is changed to the workspace. Executing this Operation on remote workspaces should not apply the page change. Executing this Operation during the replay phase as a result of having an earlier Operation being undone (i.e. an Operation being undone by another user) should also not involve changing the page. To handle these situations a flag `sb.Operation.executingRealtime` is set. An if-statement is wrapped around sensitive pieces of code, such the one for changing page in our example, with this flag. When it is not the first time an Operation is done, the flag is set to `true` and a page change occurs. When the flag is set to false, the page change does not occur.

There are also UI problems due to lack of data-binding in the existing implementation. This is a problem for widgets, breeds, and traits. For example, the code for changing the name of a widget does not bother to change displayed name in the widget editor. This is reasonable in a single-user environment since the only way to change the name of the widget is through the widget editor anyway, so the displayed name in the widget editor must have been correct. For collaboration, we need to make sure that applying remote Operations also updates the widget editor if the editor is shown for the target widget. To resolve this, a mapping was created from widget property name to the corresponding DOM element for displaying it. An Operation for changing a the name of a widget, for example, would check for whether or not the widget ID matches with the ID of the target widget of the editor. If there is a match, then the DOM element corresponding to the name property and then update the DOM element to display the new name.

### 7.2.3 Undoing an Edit

As mentioned in the design, we will need to undo Edits and Operations (when receiving remote messages while in the middle of an Edit), but this is not described until the chapter 8. For the purposes of this implementation, we choose to do something
called a *replay undo*. Basically the project is cleared and then replay all of the Edits up to the one we want to undo.

### 7.2.4 Vector Timestamp Comparison

The vector timestamps compared in the related works \[5.3.1\] compare the sum of the counts in the vector timestamp and break ties with the index of the user. We instead compare timestamps by the looking at each of the indices from left to right. As soon as we see that the count of a vector timestamp is less than the count for another timestamp at a particular index, then the timestamp with the smaller count is considered to be the “earlier” timestamp. This comparison is slower (since the sum can be cached), but has the advantage that concurrent changes between two users will always do the changes of one user first. This makes testing and reasoning easier.

### 7.3 Test Results

There are too many test cases to even considering listing all of them. A careful selection of important ones are listed here to demonstrate that the implemented system works (or does not work in some cases). Most of these will not focus on testing undo/redo. Instead, undo/redo testing will be deferred to when we discuss the results of the Edits design in chapter \[8\].

All of the below test cases will involve two users. For all visuals, User 1 will be the left and User 2 will be the right. The project-loading order is done such that User 1 will perform his changes first in the case of concurrency.

### 7.3.1 Block Tests

**Concurrently Inserting After the Same Block**

User 1 and User 2 concurrently insert two blocks after the same socket. After all changes have been propagated to both users, both inserts should have been successful.
Figure 7-4: Concurrently inserting after the same block. The first row shows two concurrent inserts. The second row shows the final outcome of both inserts.

Due to ordering, User 1’s inserted block will be connected after User 2’s inserted block. This example is shown in Figure 7-4.

Concurrently Inserting two Internal Blocks to Same Argument

User 1 and User 2 concurrently insert an internal block as a child to the same argument. Since only one block can belong in a socket, we will see that only one insert succeeds. Since User 1 goes first, his insert will succeed and User 2’s Edit will be reverted. An example of this is shown in Figure 7-5.

Receiving an Edit while Dragging a Block

In the initial state, both users start with a stack of blocks. User 1 completes his Edit of dragging a sub-stack of blocks outside, but User 2 is still in the middle of his Edit
Figure 7-6: Receiving an Edit during a Drag. User 2 drags a stack of blocks. The first row shows the state when both users are concurrently doing something (but User 1 has finished his Edit and User 2 is in the middle of it). The second row shows the state when User 2 has received User 1’s Edit. Note that User 1 will not receive User 2’s Edit until User 2 has finished the drag.

of dragging the entire stack of blocks. When User 2 receives User 1’s Edit, User 2’s dragged stack of blocks will exclude the sub-stack that was dragged out by User 1. Figure 7-6 shows an example of this.

Pasting after Cutting a Stack with a Concurrently Inserted Block

User 1 inserts a block at the end of the stack. User 2 concurrently cuts a sub-stack. After all changes, User 2’s cut should have removed the sub-stack and User 1’s change will not be visible. Note that the cut operation occurs last. Even so, the cut will only store the data string for the block during the time User 2 locally performed the change. This means that pasting from User 2 will only contain the stack that he cut, and not include the block that User 1 inserted.

Cyclic Insertion

We can produce a situation in which a user attempts a cyclic insertion. First we start with a stack of two blocks in the order block A and then block B (B comes after A). User 1 drags block B onto the page. User 2 then drags block A and connects it after
Figure 7-7: Pasting after cutting a stack with a concurrently inserted block. The first row shows the state after User 1 performs his insert. The second row shows the state when User 2 concurrently cuts the state. Lastly, the third row shows what happens when User 2 pastes.

Figure 7-8: Cyclic insertion attempt. The rows successively show the steps to attempt a cyclic insertion.

block B. When User 1 undoes, the undo/redo scheme is enacted. All Edits are first undone, and block B comes after block A. User 2’s Edit is then re-applied, which tries to connect block A after block B (which comes after block A!). This is a cyclic insertion. The cyclic insertion is detected and so, User 2’s insert becomes a no-op. Figure 7-8 shows an example of this cyclic insertion handling.

Inserting After a Concurrently Deleted Block in a Stack

We start with a stack of blocks. User 1 deletes a block (via delete key) while User 2 concurrently inserts after that same deleted block. Due to ordering of concurrent Edits, User 2’s change will happen second. This means that User 2 will attempt to insert after the deleted block when all changes have been received. This results in User 2’s Edit not happening. This is actually not intention preserving. User 2’s Edit should grow the existing stack. Anyhow, this is expected behavior for this design.
Figure 7-9: Inserting after a concurrently deleted block in a stack. The first row shows the concurrent Edits. The second row shows the converged state.

Figure 7-9 shows an example of this.

**Pasting After Text-Focus on a Concurrently Deleted Block**

This is a bug example. ScriptBlocks was implemented so that key-presses are not triggered when the document is in text-focus. A problem therefore occurs when User 1 clicks on an argument’s text-field and then User 2 deletes the block containing the argument. User 1 does not lose text focus from the text-field deletion, yet the user has no indication of this. This means that the user unknowingly loses the ability to paste, cut, or copy after the block deletion.

One might imagine solving this problem by checking for whether or not a deleted block contains a DOM element that was in text-focus, and then un-setting the text-focus if it does. Unfortunately, it is difficult/awkward to traverse through our ScriptBlocks objects to find internal DOM elements. Furthermore, arguments are not the only objects that can capture text-focus – we can text-focus on widgets as well. The lack of structure makes this troublesome.

Another solution is to simply never block the user from doing a key shortcut even when they are in text-focus. However, they might then see text pastes along with block pastes.

Lastly, we have solutions by the users themselves. They can remove the text-focus by clicking on a different text-field and then blurring (e.g. clicking on the workspace). They can also paste via right-clicking or the paste icon. The existence of alternative input methods makes this bug not so dire.
Figure 7-10: Created a call-block and adding a parameter to the same corresponding procedure concurrently. The first row shows the concurrent action. The second row shows the converged result.

7.3.2 Procedure-Related Tests

Creating a Call-Block for a Procedure Block that had a Parameter Concurrently Added

In this example, we show that side-effects occur even in a multi-user setting. User 1 creates a call-block for a procedure. User 2 adds a parameter to the same procedure. Due to ordering, User 2’s add parameter occurs second. Without a coarse design of Operations, the parameter add would not be able to account for adding an argument to the call-block for User 1’s added block. This is properly handled and an argument is added to the call-block in the converged result, as expected. Figure 7-10 shows an example of this.

Adding a Parameter to a Procedure Block that had a Parameter Concurrently Removed

User 1 removes a parameter and User 2 concurrently adds a parameter. User 1’s remove parameter will occur first and then User 2’s add parameter. Figure 7-11 shows an example of this.
Figure 7-11: Concurrently adding and removing a parameter to a procedure. Notice that since the remove parameter occurred first, the add parameter that follows could potentially use the name of the parameter that was removed. That happens in this example.

Figure 7-12: Concurrently renaming two breeds to the same name. The first row shows the concurrent renaming. The second row shows the converged result.

7.3.3 Breed and Trait-Related Tests

Concurrently Renaming Two Breeds to the Same Name

User 1 and User 2 both try to rename two distinct breeds to the same name. Since breeds must be uniquely named, one of these Edits is rejected. Since User 2’s Edit occurs second, his Edit will be rejected, and only User 1 renames. Figure 7-12 shows an example of this.

Changing the Trait Name of a Trait whose Breed was Concurrently Renamed

User 1 renames a breed. User 2 concurrently renames a trait of the breed to something else. User 2’s renaming will occur second, and it will operate with the fact that the breed is named something else. Since our Operations refer to breeds by ID, User 2’s rename will still occur. Figure 7-13 shows this example.
Change the Trait Value of a Trait that was Concurrently Renamed

User 1 renames a trait and User 2 changes the value of the same trait. User 1’s renaming will happen first. This causes User 2’s value change to operate with the fact that the trait is now named something else. Since our Operations refer to traits by name, User 2’s value change no longer applies since the trait with its expected name does not exist. Figure 7-14 shows an example of this. Note that this is not considered, by us at least, to be intention preserving behavior. However, we can solve this simply by referring to traits by ID.

7.3.4 Widget-Related Operations

Deleting a Widget with Widget Editor Open

This is not a concurrency example, rather it shows an example of how UI is handled in a multi-user environment to avoid illegal changes by a user. User 1 opens up the widget editor of a widget. User 2 deletes this widget. When User 2’s Edit is received by User 1, the open widget editor will be destroyed. This prevents User 1 from doing changes on a non-existent widget. Figure 7-15 shows an example of this.
Figure 7-14: Concurrently renaming a trait and changing its value. The first row shows the concurrent action. The second row shows the converged result.

Figure 7-15: Deleting a widget with its widget editor open. The first row shows the state when User 2 has deleted the widget, but User 1 has yet to receive the delete. After getting the delete, both the widget and the editor are removed. This is shown in the second row.
Chapter 8

Efficient Undo/Redo

8.1 Analysis of Past Work

An Operation corresponds to the execution of some code. To figure out how to undo this Operation, it follows that we just need to undo the code that it did. Given that we own the source code, this should not be an impossibility. The trick is to insert hooks (directly into the source code) to capture relevant changes that we know how to undo, and then undo those constituent changes.

As mentioned in the related work, Google and Suzuki both tackled this problem. They both noticed that the constituents of an Operation can be captured by capturing events. For example, a \texttt{sb.BlockEvent.CHILD\_CONNECTED} event implies that the \texttt{sb.Block.connectChildBlock()} method was called. The constituents can then be inverted in reverse order to undo the effects of an Operation. The major blocker with Google and Suzuki’s approach is that it requires significant bookkeeping on the developer’s part, which is less viable for large codebases like StarLogo Nova.

Especially for StarLogo Nova, the complexity of the current StarLogo Nova implementation makes the above approach very difficult to reason about for developers. Even if we know that \texttt{sb.Block.connectChildBlock()} has been called, it is unclear what the corresponding inverse is. From the naming alone, it might be reasonable to assume that the corresponding inverse is \texttt{sb.Block.removeBlock()}. Unfortunately, this is not necessarily the case. The problem stems from the fact that...
`sb.Block.connectChildBlock()` does more than its naming implies. It could potentially call `sb.Block.removeBlock()` if there is an existing child block. Since the optimistic route of assuming `sb.Block.removeBlock()` is the inverse does not work, the developer is then forced to reason about what the inverse is from reading the implementation. It also ends up calling multiple functions like `sb.Argument.setValue()` and `sb.Block.setParent()`. A developer would need to reason about which of these are not already captured to avoid double-inverting.

There are other worries that come with this approach. One such worry is that there might be cases where the event is fired in an awkward position. Consider the hypothetical example code shown in [Figure 8-1](#). In this example, `sb.Block.SOME_EVENT` is captured to indicate that the function has been called. When applying the inverse, it should undo the `this.fieldToReset` field back to its original value. Unfortunately, there is another function call between the change and representative event, which also fires an event. This event would indicate another change. Based on the capturing order, we see that `this.fieldToReset` will actually be reverted first (because it was captured last) even though it was changed first. This violates reverse ordering, which could be a problem if undoing the change by `this.changeThatFiresAnEvent()` actually requires that `this.fieldToReset` equal 0. It turns out that ScriptBlocks does not seem to have this problem; either event is fired after change or the ordering does not matter. Still, this is something that a developer would have to be aware of and keep cautious.

Another worry is that event capturing does not actually capture all constituents of a change. None of the code in WebLogo fires an event because, unlike ScriptBlocks, it does not need to notify anything of a change. The `ProcedureManager` has a number of changes that happen to it when a parameter is removed, a procedure block is deleted, a call block is added and so on. This internal state should also be restored, but there are no events captured to indicate that we should undo some part of it. The fix for this can be quite simple: we simply add events to the `ProcedureManager` (and all other non-ScriptBlocks objects) and make sure they are captured. The downside here is that, in addition to firing the event, we need to create the event class and add
Figure 8-1: Example where a representative event (sb.Block.SOME_EVENT) of a function is fired/captured in an awkward place. The representative event implies that this.fieldToReset should be reset, but it is fired after another change that fires events. Since sb.Block.SOME_EVENT is captured last here, this.fieldToReset is undone first, which is not actually applying inverses in reverse order.

another switch case in the event capturer. Every event requires modification in three separate files, which could end up being tedious.

Lastly, “traps” can be quite tempting. Consider sb.ScriptBlocks.deleteBlock(), for example, which is a function that is called when a user cuts or deletes a block. It recursively deletes the stack under the target block, and ends up calling various block disconnect and removal functions. Instead of capturing events for those disconnects and removals (ignored by setting a flag), we can instead create and fire an event for representing that sb.ScriptBlocks.deleteBlock() was called. The inversion corresponding to this event would be to re-construct a stored copy of the deleted block. The problem with this is that it requires knowing the side-effects from the disconnects and removals do not matter, and that restoring the block only is expected behavior. This requires deep understanding of the code. Having this option means that the developer needs to figure out where to flag events to not be captured. Furthermore, undo would break if the block deletion is then specified to include another side-effect. For example, it is reasonable to expect that deletion of a procedure block would cause all call-blocks to de-select it (for some reason, this is not current behavior). Restoring the procedure block without restoring the argument values for the affected call-blocks would be incorrect.

Due to the reasons above, Suzuki had trouble in implementing Edits. To be fair,
many of the worries here are particular to the implementation of ScriptBlocks, and can be handled with careful bookkeeping. In fact, Google takes this approach for Blockly and has succeeded. They even “fall” into the “trap” referred to above when implementing the undo of the deletion of a block. Although procedure block deletion has side-effects in Blockly, they manage to gracefully handle the undo to re-create the procedure block as well as undo the side-effects by restoring arguments to call blocks. They do, however, have a different (and generally simpler) code base, so it is not clear cut to say that this approach should be applied to StarLogo Nova. But this does show that the approach has its merits, and there are potential takeaways from it.

8.2 Design

To remove the concerns of the above approach, we capture at a level in which we know for sure what the inverse is. A program execution can be expressed as a tree of function calls. For each function call, we can further express it as a list of statements. There are three types of statements: function call statements, getter statements, and setter statements. Basically, setter statements are statements in which there is a direct change. Getter statements make no changes. Function call statements can indirectly modify by ultimately calling a setter statement. With this distinction, capturing a setter statement would imply unquestionably that we want the inverse of the setter statement.

This is essentially the entire idea of the design. As we will find out, implementation is a different beast and requires deviating from this design. However, keeping it in mind helped make reasoning about the code much easier.
8.3 Implementation

8.3.1 UndoMethodBuilder

We need some mechanism for capturing setter statements and storing their inverses. To do this, we use an object called an `UndoMethodBuilder`, which, as its name suggests, is used to build an undo method. Internally, the `UndoMethodBuilder` maintains a list of functions. The `UndoMethodBuilder` provides an interface for modifying this list as well as building an undo method from this list. It has a `reset()` method for clearing out the list of functions. This is for starting to build a new undo method. It also has a `append()` method for adding a function to the list of functions. The added function should be used for inverting a change. This effectively means that the `append()` method is for building onto the undo method. Lastly, we have the `getMethod()` method which constructs the undo method from the stored methods. The constructed method basically iterates through the list of functions built in reverse order and applies them.

8.3.2 Applying the UndoMethodBuilder

Most Operations follow the same pattern for applying the `UndoMethodBuilder`. Before executing the code of an Operation, the `UndoMethodBuilder` is reset. In the executed code, there will be a number of hooks (`UndoMethodBuilder.append()`) to build the undo method. When the execution is done, we fetch the undo method and set it to be the Operation’s undo.

Consider the example code for removing a breed [7.2.2](#). To incorporate the `UndoMethodBuilder` to it, the code would look like:

```javascript
var self = this;
this.showDialog("Confirm Delete",
"Are you sure you want to delete<br><em>" + this.targetBreed + "
\'</em>?",
{
"Yes": function() {
```
With the appropriate hooks inserted in the places that are called by the `Breed.removeBreed()` method, the `operation.undo` method can be called to undo the Operation. A very simple and hypothetical example of the usage of a hook would be:

```javascript
var oldName = this.name;
this.name = newName;
var self = this;
UndoMethodBuilder.getInstance().append(function() {
  self.name = oldName;
});
```

These hooks are inserted all throughout the code of StarLogo Nova. This is required to ensure that every Operation can generate its own undo method.

### 8.3.3 Library Function Calls

At some point, there will be a function call for which we do not have the source code for, and therefore cannot insert a hook to capture the setter statements; this is especially true for library functions. For example, Google Closure has a function `goog.events.EventTarget.setParentEventTarget()` for setting the parent event target. Since we cannot dig even deeper, we would need to treat these as setter statements as well. The only problem is if they do not have an inverse function. It turns out that the ones we encounter either have an inverse, or can just be ignored.
8.3.4 Refactoring to Foundation Functions

There are orthogonal problems for which we can tackle to resolve a number of issues. The first of these is that StarLogo Nova’s methods do more than they should be. This has been mentioned already regarding how `sb.Block.connectChildBlock()` does more than its naming suggests. The second problem is that my design of Edits does not suggest a simple way to handle view updates. As a matter of fact, the design was designed under the assumption of data-binding so that views are updated whenever the model is updated.

To resolve both of these problems, we refactor many sections of the code into foundation functions. Basically, a refactoring of `sb.Block.connectChildBlock()` to a foundation function would strip out the section of code inside that only connects a child block. This excludes extraneous functionality like calling `sb.Block.removeBlock()` or firing an event to trigger side-effects. This new function is suffixed with “Foundation” in its name to distinguish from the original function.

Note, however, that these functions need to update the view. The current way views are updated is to fire an event that gets captured by the `ViewManager`. We therefore have this problem where we want to fire an event to update the view, but also not fire an event to avoid foundation functions not actually being foundational. The solution to this is to separate the events fired into two so that one type of event only updates the view. The other is used only for triggering side-effects. The foundation functions therefore fire this view-update event at the end to update the view.

Once we have foundation functions, the foundation function of `sb.Block.connectChildBlock()` is actually the inverse of `sb.Block.removeBlock()` and vice-versa. Therefore, instead of capturing just setter statements, we can now capture these foundation functions. Within each foundation function implementation, we would add a hook with the `UndoMethodBuilder`, where the undo method it adds on just calls the inverse foundation function.
It turns out that almost all view updates happen in these foundation functions. It follows that we do not need to capture statements for re-rendering the view. This is already handled by us capturing the foundation functions.

### 8.3.5 Quirks of `sb.Argument.addArgument()`

It turns out that `sb.Block.addArgument()` has been implemented such that it never fires an event (or rather the event is captured at a level that prevents it from being propagated to listeners that actually do something with it). Part of the reason for this may be that its implementation for modifying the block label is incorrect. The way this method has always updated its view is by following up with a call to `sb.Block.setLabel()` on a correct label. Therefore, the foundation function to `sb.Block.removeArgument()` actually calls the foundation function for `sb.Block.addArgument()` and then calls `sb.Block.setLabel()` to update the view.

Capturing Multiple Statements at Once

Capturing foundation functions is just one of the many instances in which we capture multiple statements at once. There are some typical cases where it makes sense to capture multiple statements at once since they are independent setters. For example, the code below can have every setter belong in a single hook since they modify different fields.

```javascript
this.x = newX;
this.y = newY;
this.angle = newAngle;
```

There are, however, less usual situations in which multiple statements are captured at once. For example, the inverse of `sb.Block.dispose()` is actually a huge combination of code involving the `sb.Block` constructor. The constructor triggers an event which resets the listeners associated with a block. It therefore follows that we do not need to capture statements for de-registering the listeners (so that they are re-created on instantiate). It is possible to instead make the constructor not trigger this event when called in an undo environment, and then capture the statements for
de-registering the listeners. In fact, this is more aligned with this design in that we want to be as minimal as possible so that understanding the system is easier. However, this actually makes the implementation quite a bit easier, so this route was taken.

8.3.6 Masquerading

A deletion is something that typically cannot be undone. The undo of a block deletion is therefore actually a create with the same properties as the original block. From our usage of the UndoMethodBuilder to add a hook, it can be seen how we might potentially reference the wrong objects. For example, consider the following hypothetical code piece:

```javascript
sb.Block.prototype.rename = function(newName) {
  var oldName = self.name;
  self.name = newName;
  var self = this;
  UndoMethodBuilder.getInstance().append(function() {
    self.name = oldName;
  });
}
```

Consider the situation in which the block is renamed from `name1` to `name2`, and then deleted. After undoing once, a copy of the deleted block is created. Undoing once more is now to lead to unexpected behavior. The `self` variable in the hook actually refers to the original deleted block, so undoing again does nothing.

To solve this problem, all references to objects should be done via the ObjectManager. The hook above should therefore be transformed to the below code to work:

```javascript
sb.Block.prototype.rename = function(newName) {
  var oldName = self.name;
  self.name = newName;
  var thisID = this.getID();
  UndoMethodBuilder.getInstance().append(function() {
    var thisCopy = ObjectManager.getInstance().getObject(thisID);
  });
}
```
Troubles with Inverting In Reverse Order

It turns out there are changes that cannot be inverted in reverse order. For example, our execution might remove options for a list argument and then set its value to null. The inverses in reverse order would mean that the value is set back to original and then the options are added back. However, this does not appropriately re-update the view because setting to a value that is not an option does not update selection views. To a user, this would just seem like undo did not restore the value of an argument. To handle this situation, I changed the `sb.Argument.setValue()` method to add the option for a value it is trying to set if it did not exist.

A more nasty case happens for the `sb.ProcedureManager.onDeleteProc()` method. When a procedure block is deleted, its listeners are removed as well via the `onDeleteProc()` method. The block is first deleted and then the listeners are removed. When undoing, the listeners are added back again, and then the block is added back. But we cannot add listeners back since the block it should be referring to is created later. The simplest way to fix this is to simply make the re-adding of listeners always happen at the end. For this purpose, I added a `appendEnd()` to `UndoMethodBuilder`, which adds to a list of functions that get called at the end.

8.4 Single-User Test Results

These are tests that involve only a single-user. The figures shown in this section will only show the before and after of the test case we have attempted. It would be redundant to show the result after an undo since it is exactly what the before state was anyway. The purpose of this listing is therefore only to show the test cases considered (and the reader can only trust that the results stated reflect the behavior of the implementation).
8.4.1 Procedure-Related Tests

Undoing Return-Type Change of Procedure Block

See Figure 8-2 for example. The user starts with a procedure block, a return-call block, and some arguments inserted into the call block. The user then changes the return type of the procedure block to nothing. This makes the return-call block no longer refer to a valid procedure. Therefore it is completely emptied out. The undo of this argument value change restores everything to the previous state.

Undoing Deletion of a Parameter in a Procedure Block

See Figure 8-3 for example. The user starts with a procedure block, a return-call block, and some arguments inserted into the call block. The user then deletes the parameter of the procedure block. This removes the corresponding argument from all call blocks. Undo restores all the arguments as well as the blocks that existed before.
8.4.2 Breed and Trait-Related Tests

Undoing Deletion of a Breed

See Figure 8-4 for example. When undoing the deletion of a breed, all of the blocks on the page of the breed should be restored. Furthermore, the traits need to be restored as well. This is appropriately handled.

Undoing Renaming of a Trait

See Figure 8-5 for example. When undoing the renaming of a trait, arguments in blocks referencing the trait need to be renamed back. Furthermore, trait UI show also reflect this renaming back.
8.4.3 Widget-Related Tests

Undoing Renaming of a Series

See Figure 8-6 for example. When undoing the rename of a series, argument in blocks referencing the series need to be renamed back. The widget editor should reflect the renaming. Also, the widget itself need to update its view to reflect the renaming.

8.5 Multi-user Test Results

This section will show some more interesting tests on situations that can arise from having multiple users and undoing.

There will be two users in each of the tests. For an figures, User 1 will be shown on the left and User 2 will be shown on the right. The ordering of concurrent changes is such that User 1’s changes will always be performed first.
Figure 8-7: Undo creation of block that has another block inserted after it. User 1 inserts the green block. User 2 inserts the red block after User 1’s green block. The first row shows the outcome after these two inserts. The second row shows the outcome after User 1 undoes. Note that User 2 has yet to receive User 1’s undo message.

8.5.1 Block-Related Tests

Undoing the Creation of a Block that has Another Block Inserted After It

In this test case, User 1 and User 2 each insert a block to an existing stack of blocks. First User 1 inserts, and then User 2 inserts after the block that User 1 inserted. Lastly, User 1 undoes. The result is that neither insertion happens because User 2’s insert no longer has the context to insert. This is not intention preserving behavior. See Figure 8-7 for an example of this.
Chapter 9

Intention Preservation with Tombstones

9.1 Design

The Wantaim system handles intention preservation for all (as far as I can tell) Operations except for block connection and disconnection Operations. For example, it cannot insert a block after a concurrently deleted block. Figure 9-1 shows how Wantaim behaves in this situation and what should actually happen. The expected behavior is to mimic text, i.e. inserting a block to a stack is like adding a character to a string of text. Even though the green block tried to connect to a non-existing block, it should still have been connected to the stack. Note that the type of delete mentioned here is a delete key delete (i.e. DELETE Operation with false parameter). If the user’s intention was to delete an entire stack (i.e DELETE Operation with true parameter), then the entire stack should be deleted and concurrent inserts into the stack are not the most sensible choice.

The root cause of failure-to-insert problem here is that deletes remove context that is being operated on by other users. To resolve this, we employ a technique from WOOT which is referred to by other papers as tombstones. The key idea here is that objects are never truly deleted. Instead they are tombstoned (WOOT would call this “marked invisible”) so that they are no longer visible on the editor, but their relative
Figure 9-1: The first row shows the setup. The second row shows the actions taken by two users. The third row shows the outcome from using Wantaim. Lastly, the final row shows the expected results.

positions against other objects is preserved.

This improved real-time collaboration system adapts this tombstone technique to StarLogo Nova’s project structure and complex set of Operations. This turns out to be a very interesting extension from WOOT. The project structure is essentially a huge branching of chains. Furthermore, these chains can be disconnected from branches and popped into other chains. Also, StarLogo Nova supports two types of block deletion. One deletes the stack and another deletes just a single block (including its children). Inserting after a singly-deleted block should mean the insertion should be on the stack. Inserting after a block that was involved in stack-deletion should result in a no-op like in Wantaim. It follows that we need two separate types of tombstones. A block is marked as invisible (meaning a user can still insert to it) or gone (a user cannot insert to it). This is in contrast to WOOT which only has a single chain, supports only inserting and deleting characters, and only considers the invisible flag.
9.2 Implementation

9.2.1 BlockOrderingTree

It is impractical for us to directly mark blocks as invisible or gone in our program structure. For one, our program needs to be traversed by the compiler to compile the program. With blocks that are marked invisible and gone, the compiler would have to be aware of this extra piece of specification. From a modularity perspective, this does not sound like a good idea. Another reason why this is impractical for us is that the `sb.Block` class could potentially become unmanageably complex. Blocks would need to maintain links to “non-existent” blocks. However, we also want to maintain a link with the next alive block. For every block, we would thus need to maintain two separate links and ensure they are consistent. Given that the `sb.Block` class is already over 1000 lines of code, adding on this additional support for fancy block manipulations seems unadvised.

The solution to this is to build a virtual tree that reflects the program structure called a `BlockOrderingTree`. This is essentially a stripped-down version of the program structure that only maintains relationships and keeps track of deleted blocks by marking them as invisible/gone instead of actually removing them. This class then provides a number of helpers to update the virtual tree and fetch actual relationships between blocks, in accordance with the intention preservation cases laid out earlier.

The virtual tree supports the following interface.

1. Modification methods. As we can see these correspond with the Operations that we have.

   - `insertAfter()`
     
     - **Parameters:** block ID and target block ID.
     
     - **Specification:** Inserts the block ID after the targeted block ID.

   - `insertBefore()`
     
     - **Parameters:** block ID and target block ID.
- **Specification**: Inserts the block ID before the targeted block ID.

- **parentConnect()**
  - **Parameters**: block ID, target block ID, and argument name.
  - **Specification**: Inserts the block ID as a child to the target block ID at the provided argument.

- **pageConnect()**
  - **Parameters**: block ID, page ID, and position on page.
  - **Specification**: Inserts the block as a child to a page at the given position.

- **disconnect()**
  - **Parameters**: block ID.
  - **Specification**: Disconnects the block from where it is connected to. Note that everything rooted under the given block is still connected to it; the block is only disconnected from the previous block or parent.

- **delete()**
  - **Parameters**: block ID.
  - **Specification**: Basically applies DELETE Operation with false parameter on the specified block. This marks only the provided block and its children as invisible.

- **cut()**
  - **Parameters**: block ID.
  - **Specification**: Basically applies DELETE Operation with true parameter on the specified block. This marks only the provided block and its children as gone.

2. Getter methods

- **getActualInsertAfter()**
  - **Parameters**: target block ID.
- **Specification**: Gets the actual object to insert after. When the INSERT_AFTER Operation is called on a block and target block, it is possible that the target block is invisible. This method traverses up the stack to find an appropriate location to connect to. Note that we could want to connect to a page, parent block, or after a block. If we want to connect to a page, then a position is returned for where to connect. If we want to connect to a parent block, then an argument is provided. If we want to connect after a block, then just the ID of the block is returned. If there is no reasonable object to insert after, then `null` is returned.

- **getActualInsertBefore()**
  
  - **Parameters**: target block ID.
  
  - **Specification**: Gets the actual object to insert before. When the INSERT_BEFORE Operation is called on a block and target block, it is possible that the target block is invisible. This method first traverses down the stack to look for an object to insert before. If one does not exist, it then traverses up the stack to find an object to insert after. If nothing is found, then `null` is returned. When an object is found from traversing up the stack, we know this is a block so it suffices to return the block ID. When an object is found from traversing down, we return the same information as returned by `getActualInsertAfter()`.

The implementation of the BlockOrderingTree involves numerous cases and careful pointer manipulation so that all of the relationships are correct. However, the basic idea for the implementation is not too difficult. All of the modification methods are surprisingly simple in that it works exactly the same way as in the real program structure. Again, the only difference is that the blocks in the tree are marked with flags indicating whether it is invisible and/or gone.

The getter methods are actually the harder part of the implementation. These traverse the tree and ignore invisible nodes to find non-invisible, non-gone nodes for
insertion. However, the primary difficulty here is also handling the numerous edge cases for the type of object we find.

9.2.2 Applying the BlockOrderingTree

We first consider what happens when we receive a block connection Operation. Notice that we do not have a method in the BlockOrderingTree for handling inserting to a parent or to a tree. This is because if they do not exist, then there is no reasonable location to be inserting them. In that case, the Edit with the page or parent connection Operation becomes a no-op. When the connection is for an INSERT_AFTER or INSERT_BEFORE Operation, then we can just call the corresponding method from our BlockOrderingTree to get the correct object to insert to. This is messy since there are many cases involved, but it is doable. After this insertion, we call the corresponding modification method in the BlockOrderingTree to synchronize it with the state of the project.

Note that previously, the context was checked using just the target block stored in the Operation. Here whether or not the context exists is determined by the BlockOrderingTree. When it does not exist, the Edit is a no-op as usual.

A disconnect Operation should always be doable as long as we have the block. Therefore we just disconnect the block as usual, and then synchronize it in the BlockOrderingTree by calling its disconnect() method.

When a DELETE or DESTROY Operation is called, the blocks are deleted as usual. Then we update the BlockOrderingTree so that the nodes in the tree are marked invisible or gone appropriately. This is done by recursively calling the delete() or cut() method on the affected blocks. delete() is used when we have DELETE Operation with false parameter. cut() is used when we have a DELETE Operation with true parameter.
9.2.3 Undo/Redo

Undo is a little trickier than usual. For the most part it works the same, except that there is no such thing as deleting blocks. The unusual part here is that the first connection is what counts as creating a block, so the first connection is not actually undone. All subsequent connections to the tree can be undone. This makes it so that block nodes in the BlockOrderingTree need to keep track of a connection counter. Undoing when the block’s connection counter is 1 (i.e. only had first connection) will do nothing other than make the block invisible.

9.3 Test Results

Many of the real-time collaboration test cases were already done in an earlier chapter describing the real-time collaboration system based on Wantaim. The test cases here will focus on demonstrating the ones that cannot be handled by Wantaim.

There will be two users in each of the tests. For an figures, User 1 will be shown on the left and User 2 will be shown on the right. The ordering of concurrent changes is such that User 1’s changes will always be performed first.

9.3.1 Inserting after a Concurrently Deleted Block

We start off with an initial stack of blocks. User 2 inserts after a block while User 1 concurrently deletes (with the delete key). Due to ordering of changes, User 2’s insert will happen after User 1’s delete. Despite this, User 2 is still able to complete his insert. This is in contrast to the previous design using Wantaim that would ignore the insert. Figure 9-2 shows an example of this.

9.3.2 Undoing Creation of a Block that is Inserted After It

We start off with an initial stack of blocks. User 1 inserts a block, then User 2 inserts a block after the one User 1 inserted. Lastly, User 1 undoes. Even though the context
9.3.3 Inserting into a Concurrently Deleted Stack

We start off with an initial stack of blocks. User 2 inserts a block, while User 1 concurrently deletes a stack (with Ctrl+X) that the block is inserting into. For stack deletes, the intention is to delete the entire stack. Therefore, User 2’s insert should not have any context to insert into. As we can see in the example shown in Figure 9-4, this behavior is achieved.
Figure 9-4: Inserting into a concurrently deleted stack. User 2 inserts a red block after the green block, but User 1 deletes the stack rooted on the green block. This concurrent action is shown in the first row. The second row shows the final converged outcome.

Figure 9-5: Concurrent delete and parent connect. The first row shows the concurrent action of User 1 and User 2. The second row shows the final converged outcome.

9.3.4 Connecting to a Concurrently Deleted Parent Block

We start off with an initial stack of blocks, and, in particular, one of these blocks is a parent block. User 2 inserts a command block into the parent block, while User 1 concurrently deletes this parent block (with `delete` key). The final outcome should be that the command block User 2 inserted does not connect to the stack where the parent block was. As we can see in the example shown in Figure 9-5, this behavior is achieved.
Chapter 10

Future Work

The work presented in this thesis implements an efficient Edits and gives a proof of concept for real-time collaboration in a block-based programming environments. There are still many considerations and a lot of work left to do to make real-time collaboration a feature in StarLogo Nova.

The first of these is setting up for incorporating real-time collaboration to StarLogo Nova. A working real-time collaboration system that can demonstrate the ability to appropriately merge changes from multiple users in real-time is far from sufficient to have it be included as a feature. Authorization and privacy are important. A user should be able to choose who he wants to allow to edit or read a project. In addition, backwards compatibility is necessary if we want users to be able to collaborate on older projects. The IDing of objects has been completely changed with real-time collaboration, so it might not be compatible. Also, it would be useful to have the page be able to show users currently working on the project at the same time. The designed system does not even have the ability to reason about disconnecting users, so more thought needs to go there.

The real-time collaboration system is also lacking in terms of performance. There are several problems that stand out. First, the vector timestamps grow with each time the project is loaded. A more clever approach is needed to either limit the growth or prune out unused indices. Furthermore, the current implementation of the communication layer uses a uni-directional communicational channel from client to
server. To emulate communication from server to client, the client does a looping ping to the server. More appropriate tools such as SocketIO [8] which enable bi-directional communication should be used instead. Lastly, receiving concurrent changes and far-back undo/redo can be expensive. Edits need to be rolled back and then rolled forward again in these situations. Considering that view updates are unnecessarily performing during this process as well, this is more troubling that we would like. One potential improvement is to figure out a way to determine whether or not an Edit is dependent on another. Independent Edits would not need to be undone and redone in that case.

Lastly, there are UI improvements that we can have for Edits and real-time collaboration. For both Edits and real-time collaboration, it could be useful to have undo/redoswitch pages, scroll to the right position, or open up the trait UI tab for where the change happened. This helps the user keep track of his undo/redo. For real-time collaboration, it could be useful to indicate the page or block that a user is working on.

The future work listed here is only a small subset of possibilities.
Chapter 11

Conclusion

StarLogo Nova is an educational tool for programming and visualizing 3D systems and games. It supports visual learning by letting students see in a graphical user interface the systems that they implement. It also teaches computational thinking by letting users program in a high-level block-based programming environment that eliminates syntax concerns.

My thesis works on a project that tackles two features of the block-based programming environment editor. They are undo/redo and real-time collaboration. Undo/redo is especially important for user experience. Most editors these days support undo/redo – lacking it almost seems unusual. Real-time collaboration is valuable both from a productivity perspective and educational perspective. Collaboration is an element that helps students build computational thinking by learning to build upon each other’s work and thinking about how to abstract work into independently-implementable components.

As a result of my work, I have designed and implemented a proof-of-concept real-time collaboration system with multi-user undo/redo support. Furthermore, it has, as far as I am aware, good guarantees for intention preservation. The results for this system are collectively shown in the results section of each intermediate system designed and implemented to reach this final system.

The implemented system can be easily modified to support single-user undo/redo for StarLogo. It also serves as a starting point for adding a complete real-time col-
laboration feature on StarLogo and other block-based programming environments. Moreover, I believe the work presented in this thesis will be valuable for other applications as well. The Edits design I have described can be a useful reference for other editor applications that require undoing complex but undoable operations.
Bibliography


