COLLABORATIVE EDUCATIONAL ENVIRONMENT DESIGN FOR ACCESSIBLE DISTRIBUTED COMPUTING

By

Brian Broll

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

DOCTOR OF PHILOSOPHY

in

Computer Science

May 11, 2018

Nashville, Tennessee

Approved:

Akos Ledeczi, Ph.D.

Gautam Biswas, Ph.D.

Jules White, Ph.D.

Aniruddha Gokhale, Ph.D.

Corey Brady, Ph.D

Acknowledgements

First, I would like to thank my advisor, Professor Ákos Lédeczi. I am grateful that he took a chance and accepted a student with a mathematics major and only a few classes in Computer Science. Ákos has often provided guidance, constructive criticism, and unadorned feedback on my writing which has improved the quality of my research. I would also like to thank my Ph.D. Committee, Professors Biswas, White, Gokhale, and Brady. Their valuable feedback and questions regarding my research has been very helpful both for the dissertation and potential future work.

I would also like to thank my frequent co-authors, Péter Völgyesi, Miklós Maróti, and János Sallai. Each of them have been willing to provide valuable feedback and insight when I would wander into their office while working on some research problem. As these questions were sometimes asked in broken Hungarian, I am also quite thankful for their patience.

Finally, I am especially thankful for my wife, Cassie. She has been very supportive throughout the entire graduate program. This has included busy travel schedules, late nights (and early mornings) working on research and class work, and learning much more about NetsBlox and educational environments than she probably ever thought she would.

This work was made possible through Vanderbilt University's Trans-institutional Programs (TIPs). This material is also based in part upon work supported by the National Science Foundation under grants CNS-1644848 and DRL-1640199. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NSF.

Table of Contents

Appendices

List of Figures

Chapter I

Introduction

1.1 Computer Science Education

Software has become a ubiquitous part of everyday life and its presence only continues to grow. Smartphones, wearables, and the Internet of Things have been increasing the prevalence of electronic devices, providing new platforms for software innovation and reinforcing the importance of software development. Occupations in information technology have drastically increased over the past 50 years. In 1970, there were approximately 450,000 individuals working in information technology in the United States; in 2014, this number reached 4.6 million [13]. This growth has also been projected to continue through 2022, increasing the number of jobs by 18% [108].

This rise in technology employment and use in everyday life has emphasized the importance of computer science and STEM education [68]. There have been a number of efforts worldwide to introduce children to computer programming such as Computing At School and CSForAll. Additionally, there are many organizations supporting this same goal including Khan Academy, Code.org, Girls Who Code, and the Raspberry PI Foundation. Visual programming languages have played a very prominent role in this effort and have been used to teach programming [83, 139, 114] as well as science and computational modeling [135, 124].

1.2 Educational Visual Programming Languages

The educational literature on learning computing is filled with observations about challenges faced by beginner programmers. When first learning to program, students must learn both the syntax and semantics of the given language. As languages consist of a limited number of elements which can be composed in many ways, students must learn the semantic results of combining the given language elements in a variety of ways. Furthermore, the students also need to learn how they can compose the given language elements in a way in which the semantic result performs the desired function. As students are learning to solve these semantic challenges, they often get confused with syntactic intricacies of the language [125, 102]. Reducing the syntactic complexities of a language allows the students to focus on the semantic challenges [54].

Many visual programming languages, like Scratch, simplify the syntactic complexity of the language by providing blocks representing the language elements which can be composed

using a simple drag-and-drop interface. The shape determines how the blocks can be composed; incompatible blocks cannot be connected together resulting in the inability for users to create syntactically invalid code. As a beginner programming language, it allows novices to simply focus on the semantic challenges of learning the language rather than struggling with syntactic intricacies.

1.3 Networking and Distributed Computing

Existing educational visual programming tools focus only on the computer and disregard the network, an equally important concept. Networking is a key component to many commonly used software applications. Examples of networking are everywhere and include the web, Amazon, Google, Twitter, Facebook, YouTube, autonomous vehicles, mobile phones, and online gaming. Networking enables the development of distributed applications where multiple applications communicate over the network to coordinate and perform some desired function. Distributed computing is another important concept and is present in all the previously mentioned networking examples.

The ubiquity of networking, especially among common daily activities, makes it a necessary part of computer literacy. Many common applications not only use networking but require it to function. Without introducing networking concepts to young learners, the key elements of these types of applications cannot be effectively understood. If they cannot understand the key elements of this fundamental technology, computer literacy will be a challenge.

Introducing networking concepts to young learners provides a pedagogical opportunity. The pervasiveness of networking, especially in many popular social applications, makes programming these types of network enabled applications particularly relevant to students and provides excellent motivation. Incorporating networking capabilities into early educational programming tools could enable users to access network resources and build their own distributed applications. This can enable users to incorporate relevant, real-world data making programming more relevant as well as engaging to a variety of different student interests. Building distributed applications provides an opportunity to make programming a more social activity as students can develop distributed applications together.

Networking and distributed computing are important computer science concepts and should be introduced in the K12 curriculum. Concepts such as asynchronous and synchronous communication, reliable and unreliable protocols, and the need for concurrency in operating systems are advocated by the ACM/IEEE computer science curriculum starting at the college level. The premise of this work is that through a carefully designed interface

and the use of natural, intuitive abstractions, distributed concepts could be introduced as part of the high school computer science curriculum.

1.4 Problem Statement

The primary focus of our work is to make distributed computing accessible to high school students with little programming experience. By building on the success of visual programming languages for making programming accessible to novice programmers, we believe that developing abstractions within the blocks-based programming paradigm can make distributed computing accessible even to users with little programming experience. Enabling students to develop distributed applications would provide concrete examples and experiences from which they can learn distributed computing concepts. However, providing capabilities for developing distributed applications also adds complexity to student projects. As the complexity of user applications grows, it is important to provide additional supportive capabilities to enable the users to manage increasingly complex applications. We have identified a number of challenges in making distributed computing accessible to high school students. The first two problems pertain to enabling novices to develop distributed applications; the remaining problems are related to the management of complex applications.

- **Distributed Programming Abstractions.** Designing the appropriate abstractions supporting the development of distributed applications is important for enabling novices to work with them effectively. The abstractions must hide unnecessary complexities and have a low threshold to developing basic distributed applications. At the same time, they must also have a high ceiling to support the creation of more sophisticated distributed applications demonstrating concepts like messaging patterns or data processing paradigms. Access to the network, a fundamental component of distributed applications, provides an opportunity for accessing internet resources as well.

- **Remote Block Execution.** Developing distributed applications introduces additional design considerations such as network latency and data locality. That is, understanding the strengths and limitations of specific execution environments can be used to inform design decisions when developing a distributed application. To expose students to these concepts, we propose allowing them to execute custom block functions on remote computing resources outside of the blocks-based programming environment.

  The execution of custom block functions outside of the blocks-based programming environment is a non-trivial task. Different execution environments may have different capabilities from the original programming environment. Consequently, the behavior

of the individual blocks may require modification to perform as expected when executing in the new environment. Although the behavior of the individual blocks may change, it is important that the semantics of the code remains unchanged and behaves as expected. This includes conforming the concurrency model used by the original environment as well as supporting closures and functional capabilities of the original language. Additionally, security concerns resulting from executing arbitrary code on shared, distributed resources are also an important design consideration.

- **Collaborative Editing.** Collaboration is an essential part of solving challenging problems and developing complex applications. Collaborative editing in lively, blocks-based programming environments introduces a number of unique challenges. "Lively" programming environments do not provide distinct development and execution stages; the program is always responsive and automatically updates according to user modifications. Although this promotes student exploration and feedback, it introduces challenges when considering collaboration in these types of environments. Specifically, what should be synchronized in these environments? Subsequently, exactly which actions or behaviors should be synchronized and how should we support this synchronization?

- **Novice-Friendly Version Control.** Version control is commonly used in large software projects and provides a powerful way to recover from mistakes as well as gather insight into the history of a given project. Perhaps as a result of the powerful capabilities of version control, they are often very complex and not accessible to novice programmers. Most version control tools operate on text files and cannot provide meaningful information about changes made in other types of files.

  Regardless, version control capabilities could be very effective in assisting novice programmers in managing more complex applications. This introduces unique challenges. The version control capabilities should be intuitive and understandable to students with little or no direct instruction. Project modifications should also be represented meaningfully to the students. Standard approaches to displaying differences in version control tools are designed for text data and are inadequate in the given setting.

1.5 Contributions of this Dissertation

In this work, We present abstractions designed to make distributed computing accessible to novice programmers. These abstractions have a low threshold while simultaneously supporting the development of even sophisticated distributed applications. Additionally,

We demonstrate the capabilities of these abstractions through examples that contain both simple and complex distributed applications to illustrate the low threshold and advanced capabilities of the provided abstractions.

We also present a technique for executing blocks in alternative execution environments with different capabilities. This includes designing a cross-compiler from the source block language supporting the configuration of the underlying primitive block implementations. The compiler design also addresses adherence to the original concurrency model and safety concerns when compiling arbitrary code for execution on potentially shared resources. Furthermore, we present supportive capabilities including collaborative editing in a lively, blocks-based environment and novice-friendly version control support.

Finally, we have designed a sophisticated environment supporting these abstractions (including the collaboration and version control capabilities) called NetsBlox. We provide three case studies using NetsBlox to evaluate the following hypotheses about the provided contributions:

- The networking abstractions enable novices to develop distributed applications.
- Building distributed applications with these abstractions enables users to develop a better understanding of important distributed computing concepts.
- Providing access to additional resources and making programming more social improve student interest and engagement.

1.6 Organization

This dissertation is structured as follows. Chapter 2 presents the distributed programming abstractions. This includes an overview of the related work and a presentation of the proposed approach in Section 2.1 and Section 2.2, respectively. Section 2.3 presents concepts and interfaces for debugging distributed applications. Examples demonstrating the simplicity and expressiveness of the abstractions are outlined in Section 2.4.

Chapter 3 discusses the approach for execution blocks in alternative environments. Section 3.1 presents related work for generating code from block-based programs as well as hybrid visual and textual programming environments. The design of the block compiler is specified in Section 3.2. In Section 3.3, the use of the compiler to execute blocks in a server environment is described.

Collaborative editing capabilities are discussed in Chapter 4 including an overview of related work. Conceptual and technical challenges are examined in Section 4.1 and Section 4.2, respectively. Our approach is then presented in Section 4.3.

Similarly, novice-friendly version control is discussed in Chapter 5. Background and related work is provided in Section 5.1 including both teaching version control in the classroom and approaches to simplify version control. Finally, we present my approach in Section 5.2.

Chapter 6 investigates empirical support for the presented abstractions and designs. Related work evaluating educational programming environments is shown in Section 6.1. NetsBlox, a prototype environment of the presented work is described in Section 6.2 and three case studies using NetsBlox to evaluate the presented concepts and abstractions are discussed in Section 6.3. Finally, we conclude the dissertation with a discussion of the advantages, limitations and areas for future work in Chapter 7.

Chapter II

Distributed Programming Abstractions

2.1 Background and Related Work

2.1.1 Educational Programming Languages

Logo

One of the earliest educational programming environments is LOGO. This is a programming language designed to teach students computer programming by enabling them to control a "turtle" which moves and draws to create graphical effects. By simplifying the language syntax, LOGO made programming more accessible to novices and made it an effective tool in the classroom [93, 63, 99, 24]. Although LOGO generates a graphical output, it is a powerful list-processing language which supports a number of advanced features including subprocedures and recursion [82].

LOGO proved an effective resource in the classroom. As a result, it provided inspiration for many subsequent educational programming languages. These new languages build upon LOGO to make it applicable for various other applications including modeling complex systems [107, 135, 97, 130], learning object-oriented programming [36, 91], and even audio programming [53]. LOGO also provided the inspiration for a number of other languages and environments which may have a less obvious resemblance but still extend the principles and vision of LOGO [59, 78, 87, 75].

NetLogo and StarLogo

Two notable LOGO derivatives are NetLogo [130] and StarLogo [107]. These environments are both targeted toward modeling complex systems and require the user to program in a "decentralized manner." Users program the behavior of individual agents and then explore the outcomes of the entire system. This enables users to view complex systems as the result of many individual, independent behaviors rather than a simple centralized mindset.

One example of such a simulation can be found in mold aggregation behavior (provided in [107]) and is shown in Figure 1. In this example, mold cells are represented by white pixels on a black background. The user programs the behavior of the mold cells to emit a chemical pheromone and follow the gradient of the surrounding pheromone. The background "patches" can be programmed to allow the current pheromone on the given patch to slowly evaporate. When the behaviors of the mold cells and the background are combined, the

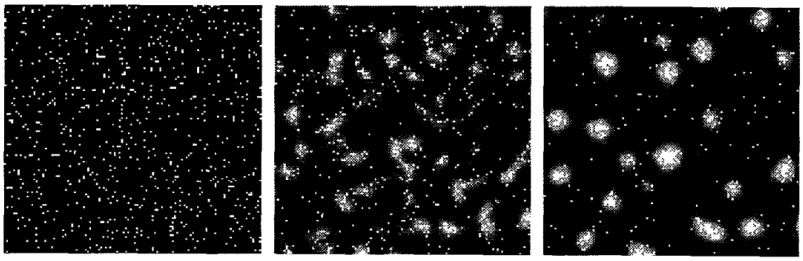


Figure 1: Modeling Mold Aggregation in StarLogo [107]

mold cells will begin to form clusters shortly after running the simulation. Although this is a simple example, it captures the essence of the emergent behavior in decentralized systems and their ability to be modeled in environments such as StarLogo and NetLogo.

2.1.2 Educational Visual Programming Languages

LogoBlocks

LogoBlocks was the first blocks-based programming language. It was designed for programming on the Programmable Brick [14]. Building on top of Brick Logo, LogoBlocks merged the visual programming techniques of the time with the syntax and functionality of Brick Logo [82]. Although LogoBlocks was limited in its functionality (it did not have support for functions or branching [14]), its user interface introduced some concepts and visual cues still used in popular blocks-based languages today. These include the rounded rectangles used for statements and input slots for input arguments. The sample program is shown in Figure 2 [14].

Scratch

The most well-known blocks-based programming environment is Scratch [78]. Scratch enables users to build interactive applications ranging from greeting cards to simulations. Drawing inspiration from Etoys, StarLOGO, and LOGO, Scratch provides a simple responsive environment to make programming accessible to novices [59, 107]. It also runs in the web browser and requires no additional software to be installed on the computer, contributing to its ease of adoption. Unlike many other educational programming environments, such as Alice, Greenfoot, and Snap*!* [27, 64, 87], it targets younger users (the most common starting age for a "Scratcher" is 12 [116]).

Scratch provides a carefully designed interface and simple abstractions to enable novices to begin programming. Users are given a "stage" where the output of the program can be viewed along with multiple "sprites" or entities interacting on the stage. Both the stage and

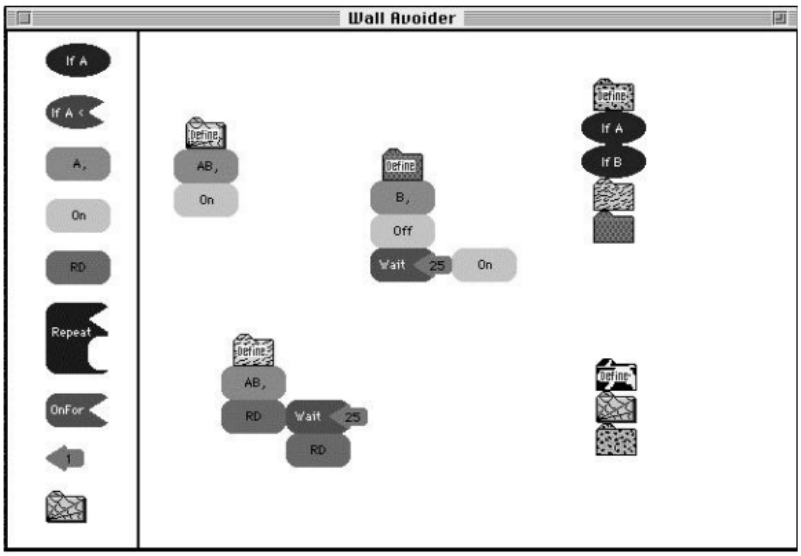


Figure 2: Sample Program in LogoBlocks

the sprites are programmable and can also easily incorporate their own media by adding either "costumes" (images) or "sounds" to the given entity. This environment is shown in Figure 3.

Lego-like blocks are connected to create programs and the shape of the blocks provides the user information about how it can be used. The color and text provide insight into what the block might do. For example, statements (or "command blocks") are represented using rounded rectangles with indents on the top and bottom which imply that they can be connected in sequence. Blocks that only have a connection indent on the bottom listen for events and are used to start scripts. Expressions (or "reporters") are rounded and have no top and bottom indents and cannot be connected in sequence but rather can be placed inside of other blocks that contain a similarly shaped input slot.

The script in Figure 4 is executed when the green flag is clicked (as the top block states). It results in the corresponding sprite to bounce around the screen and say "You got me!" if the mouse touches it. In this script we can see that the blocks are designed to be simple and user-friendly. Block shapes restrict them from being used in invalid contexts; "forever" and "if" blocks can only be connected after another statement and could not be used in an input slot of another block. The "touching" block's diamond shape indicates that it returns a boolean value. The empty "if" block's input field is also a diamond shape indicating that it expects a boolean input as opposed to other input slots like in the "move" or the "say" blocks. By carefully designing the shapes and colors of each of the blocks, Scratch is

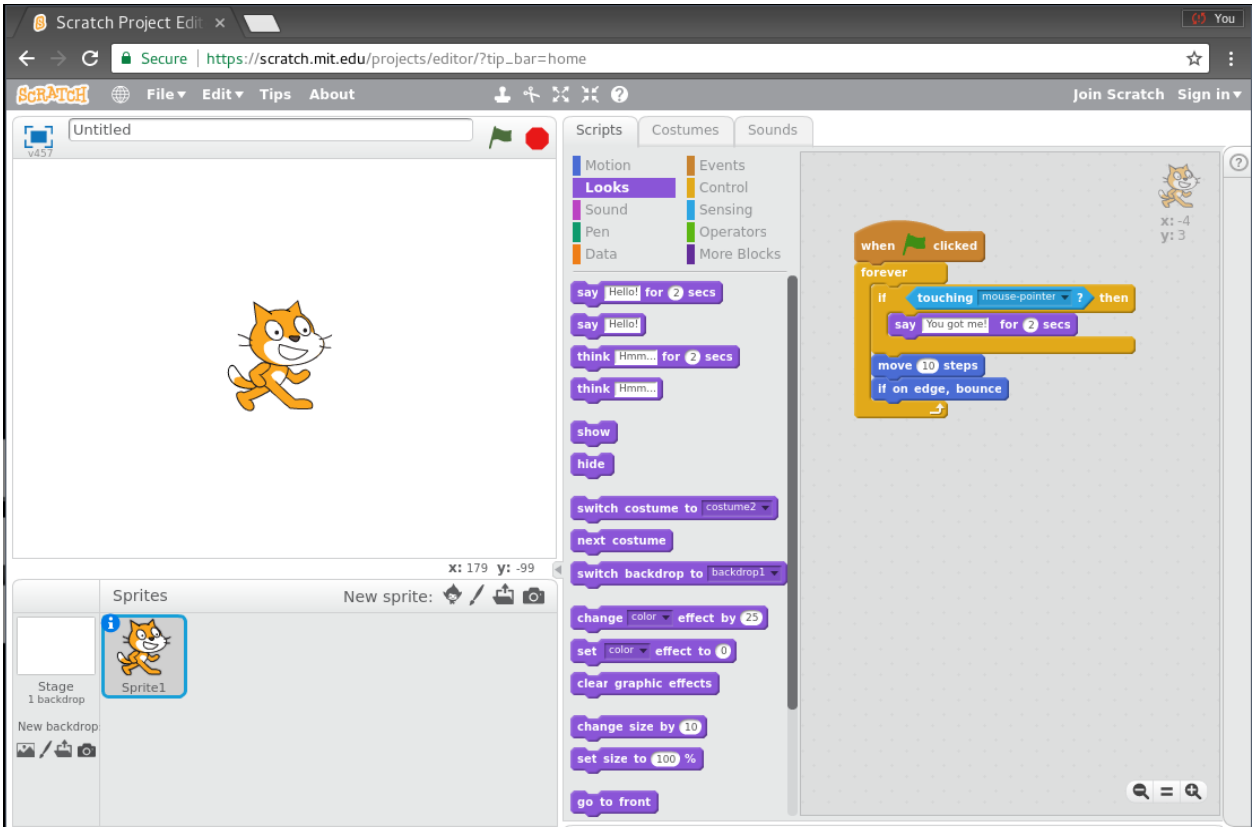


Figure 3: The Scratch Environment

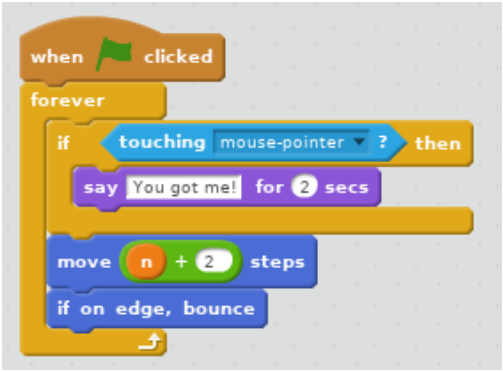


Figure 4: A Simple Script in Scratch

able to prevent the user from creating syntactically erroneous scripts and visualize implicit constraints between blocks to promote creating meaningful scripts and programs.

The Scratch team provides support for experimental extensions to Scratch on their ScratchX website. These extensions allow users to add extra functionality to Scratch including access to temperature data, Twitter, Spotify, and arduino support [32]. However,

these extensions are not managed or endorsed by the Scratch team. Projects created on the ScratchX website are not able to be shared on the Scratch website.

Snap!

Snap*!* is a conceptual extension of Scratch developed at UC Berkeley. It was designed to raise the ceiling of Scratch by providing more advanced functionality including first class functions and lists [87]. Drawing heavily from Scratch, Snap*!* provides a seamless transition for Scratchers. Snap*!* and Scratch both provide open-ended, exploratory spaces in which the users can develop their programs.

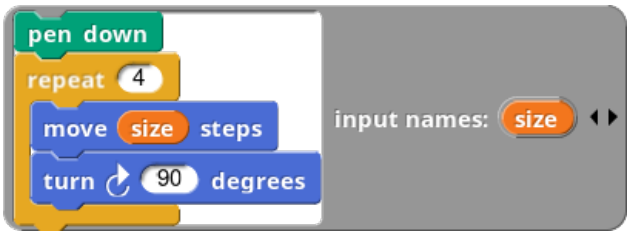


Figure 5: An Anonymous Function for Drawing a Square in Snap!

Figure 5 provides an example of an anonymous function in Snap!. As Snap! also supports first class lists, these functions can be placed in lists and dynamically selected and run at runtime. This high degree of flexibility makes Snap! a powerful tool for teaching advanced programming concepts and has lead to its success as part of the Beauty and Joy of Computing [12] curriculum. An example using both first class lists and first class functions to introduce recursion is provided as a sample project in Snap! called "vee" shown in Figure 6.

In this program, a sprite has a list variable called "shapes" which is filled with functions for drawing squares, hexagons, and stars as well as a function called "vee." "Vee" selects two functions out of the list to call randomly. When the application starts, the "vee" function executes and results in the sprite drawing two shapes from the ones in the list. This is a simple example but can become even more interesting as pressing the "up arrow" will result in adding the "vee" function itself to the list. This results in the program becoming non-deterministic as the function now may recurse and select two more shapes to draw from the list. A screenshot of this application (including the implementation of the "vee" function) is given in Figure 6.

Along with raising the ceiling of Scratch, Snap*!* is implemented in JavaScript (as opposed to ActionScript as in Scratch v2.0) and made its source code openly available under the AGPL license. These subtle technical decisions allowed it to become the starting point for a number of extensions which pushed the boundaries of block-based programming and applied it to a number of different domains.

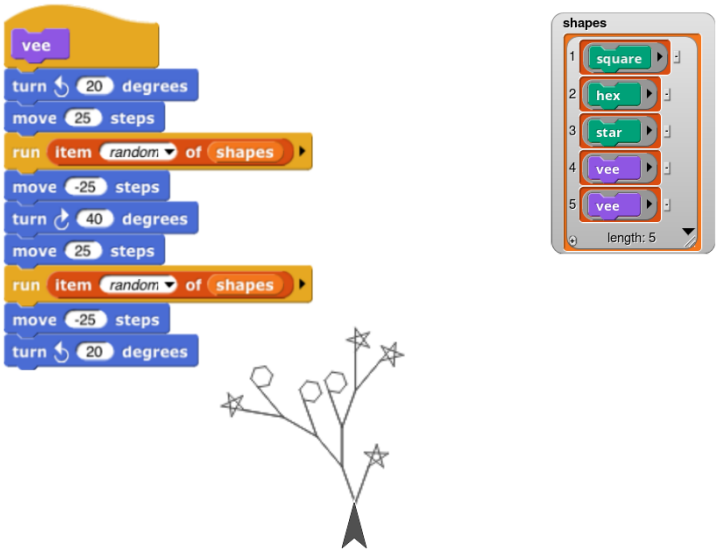


Figure 6: An Introduction to Recursion using Anonymous Functions and Lists

One such extension is BeetleBlocks which replaces the iconic LOGO turtle with a beetle which can not only "walk" around a two-dimensional plane but also fly in a three-dimensional environment. In this environment, users can construct three-dimensional objects and explore three-dimensional geometry [66] (which can later be exported and printed on a 3D printer).

Figure 7 shows an example project in BeetleBlocks constructing a three-dimensional staircase with the code displayed on the left and the output of the execution on the right. In this example, the program is creating two sides of a step in the nested loop where the depth of the stair is -50 and the length is given by the variable a. The inner loop is executed twice which results in the construction of all 4 sides of the three-dimensional stair. The code following the nested loop decreases the size of a (shortening the width of the subsequent stairs), draws the vertical section between stairs, and updates the hue. This outer loop is performed 10 times resulting in the creation of a staircase with 10 stairs of different colors.

The example in Figure 7 demonstrates the complexity of thinking in 3 dimensions and how relatively simple code can result in complex and intriguing structures. BeetleBlocks also supports exporting projects to a 3D printer which enables code to become even more tangible and relevant for users as their projects are no longer simply an abstract entity on the screen. There are also many other Snap*!* derivatives targeting a number of other domains including robotics, APIs, database queries, graph algorithms, and parallel computing [120, 4, 47, 40, 29].

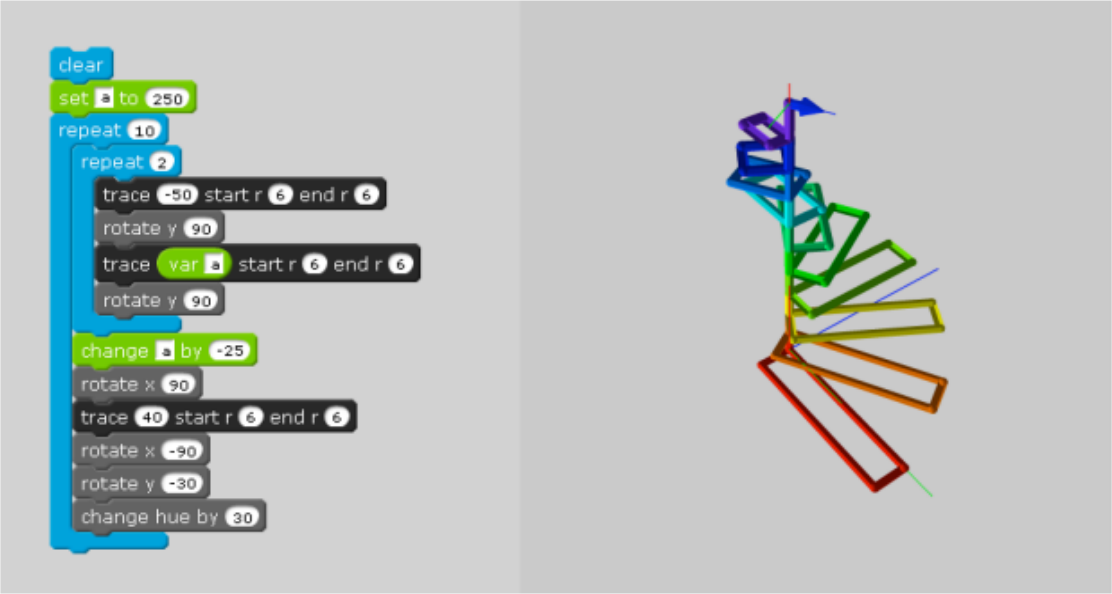


Figure 7: Creating a 3D Staircase in BeetleBlocks [112]

Three-Dimensional Environments

Although BeetleBlocks provides a very powerful platform for creating programs with a three-dimensional output, there have been a number of other projects providing different approaches [27, 60, 31, 135, 75, 95]. Some noteworthy alternatives include Alice [27], and Kodu [75].

Alice3 is an educational programming environment which provides users with a three-dimensional world in which they can see their projects come to life. Although early versions (Alice and Alice2), provided a textual programming interface [27], Alice3 provides users with a visual blocks-based language [60, 31]. This environment provides three interfaces: a scene editor, a code editor, and a runtime display.

Figure 8 shows an example project in Alice3 [31]. In this example, the user has created a scene with a number of different entities shown in the interface on the top left. These entities include obvious ones, such as people and objects, to the less apparent ones such as the ground and the camera. The code for the scene is shown in the code editor given in the top right panel. In this scene, the code defines the behavior of the scene when it runs; in this case, Joe will run "callEveryoneOver" and Susan will run the method "lookAroundParanoically" simultaneously. The output of this scene is currently visible in the runtime display shown at the bottom of the window. This display shows the "Joe" avatar is raising his hands and looking around ("calling everyone over") and the "Susan" avatar looking around "paranoically."

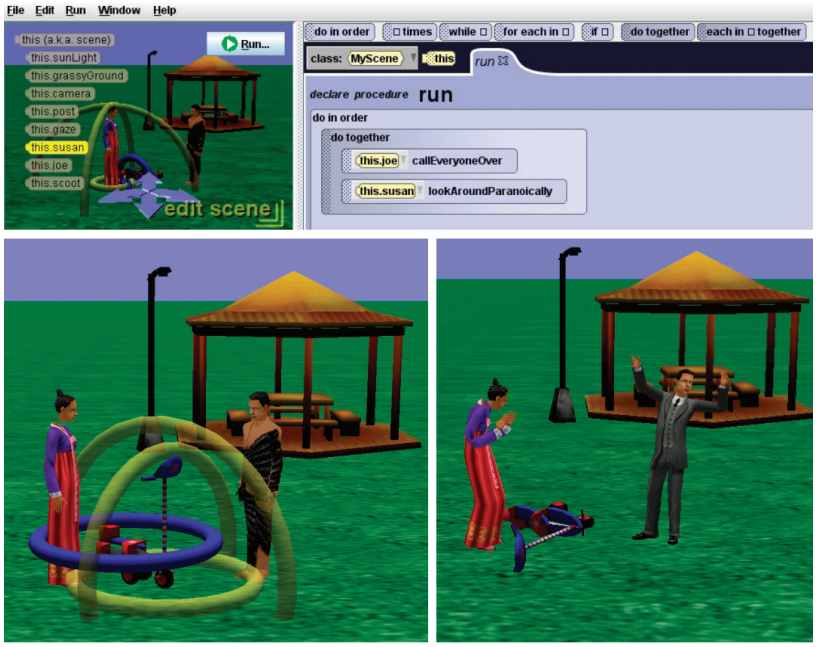


Figure 8: An Example Project in Alice3

Alice3 provides a number of other valuable pedagogical improvements upon its predecessors. These include providing both block and text interfaces in the code editor enabling users to switch between them during development as well as facilitate the "graduation" from blocks-based to text-based programming. They also include more motion methods to simplify the development of three-dimensional actions and "lower the barrier to entry" for creating three-dimensional projects. Finally, they facilitate the transition to textual languages by not only letting users write textual code in their code editor, but also supporting opening their project as a Java project in the NetBeans IDE [31].

Another interesting programming environment for developing three-dimensional applications is Kodu from Microsoft Research [75]. Kodu runs on the Xbox 360 and is targeting young children. Whereas Alice provided a number of tools for bridging the gap between block based and textual environments, Kodu provides a much more simplified language for developing the applications.

The language contains five basic elements: sensors, filters, selectors, actuators, and modifiers. A "sensor" specifies a virtual sensor used to detect elements in the game. The elements detected by the sensor are then reduced with a "filter." The "selector" selects an element from the filtered set, such as *nearest*. An "actuator" makes the given actor do some action toward the selected element or itself and the "modifier" is an optional parameter for the "actuator."

Figure 9: Programming in Kodu

Figure 9 shows an example script created in Kodu [75]. The top script defines a behavior for the given entity in which the sprite will move toward a red apple if it sees one. In the "when" block, three blocks are used to specify that the action will occur when a red apple is seen: "see," "red," and "apple." The "see" block is a sensor block which detects elements in the environment. These elements are then filtered by the "red" and "apple" filters.

After the "when" block detects that the robot can see red apples, the "do" block defines the behavior of the actor in this event. In this case, the "do" block contains one actuator, "move," and one modifier, "toward." Together these scripts define a behavior for the actor in which it will move toward a red apple when it sees one. One noteworthy visual cue is the subtle size difference between the first square in each section and the subsequent squares which modify the behavior of the leading square. This helps reinforce the prominence of the leading squares and the sort of complementary, secondary nature of the latter squares.

State management is particularly interesting in Kodu. Unlike most other languages, Kodu does not provide typical support for variables such as strings or integers. Rather, Kodu programs rely on storing the information about the state of the application in the environment. One example of such a method is given by encoding the information into the color of an object. According to MacLaurin, they have seen projects use the character color as a lightweight form of synchronization between characters [75]. Therefore, characters change their color to notify each other about their own internal state.

There is one method in which users are able to store information: the concept of the "score." Kodu provides characters with integer valued scores which can be modified and used to maintain program state. The Kodu team has found that users have been able to utilize the character scores for many other tasks such as resource counts, iteration counts and semaphores [75].

Another unconventional platform for visual programming environments for novices can be found in Minecraft, an online open-world computer game composed of bricks with which users can build their own worlds [86]. Although Minecraft itself is not designed specifically

for programming, it supports modifications which can enable it to be extended into a programming platform. When this is combined with the popularity and simplicity of the game itself, Minecraft can be used as an effective vehicle for engaging students and introducing programming.

"CodeBlocks" is a block-based programming environment enabling users to write programs within the Minecraft environment using custom three-dimensional blocks and placing them in sequence to define functions [144]. After creating functions, the user is able to create robots which can then be controlled with the user's defined functions. This enables users to create and program bots entirely within the Minecraft environment.

CodeBlocks provides support for commands, branching, and functions (without return values). Some blocks can be parameterized by placing a sign above them. Like Kodu, the robots exist in an interactive world which can contain the state of the program. For example, in [144], users were asked to implement a variation of bubble sort in CodeBlocks in which the robot was sorting three-dimensional blocks in the virtual world.

Figure 10: Programming in CodeBlocks [144]

Figure 10 provides a simple example in which the robot is programmed to simply move forward, turn right and then pick up the block in front of it [144]. The beginning of the program is given by the "program example" sign and is executed by running the blocks following the sign. An example of branching and block parameterization can be found in Figure 11.

In this example, the function contains a sensing block which checks for "dirt" as given by the sign on the top of the block. As the sense block may or may not detect dirt, the code branches. Visual feedback for these paths is given around the sensing block. The green tiles (in line with the sensing block) represent the blocks to be executed if dirt is sensed by the robot; if not, the code will continue along the path of the red tiles.

Mobile Development

A number of educational visual programming languages have been created for mobile development including App Inventor, Sketchware, and PocketCode [103, 123, 122]. App

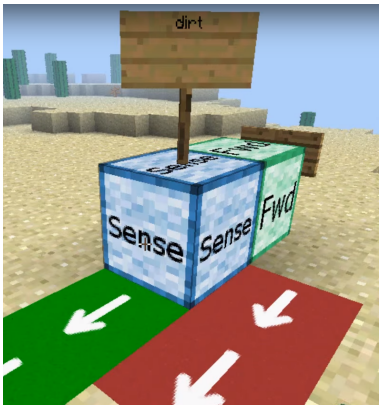


Figure 11: Branching in CodeBlocks

Inventor provides users with a "Designer" and a "Blocks Editor." The Designer enables users to design the components of the app including sensors and visual elements. After designing these components, the Blocks Editor can be used to program the behavior of the application such as responding to shaking the device. An example app created in MIT App Inventor is given in [103] and is shown Figure 12 and Figure 13.

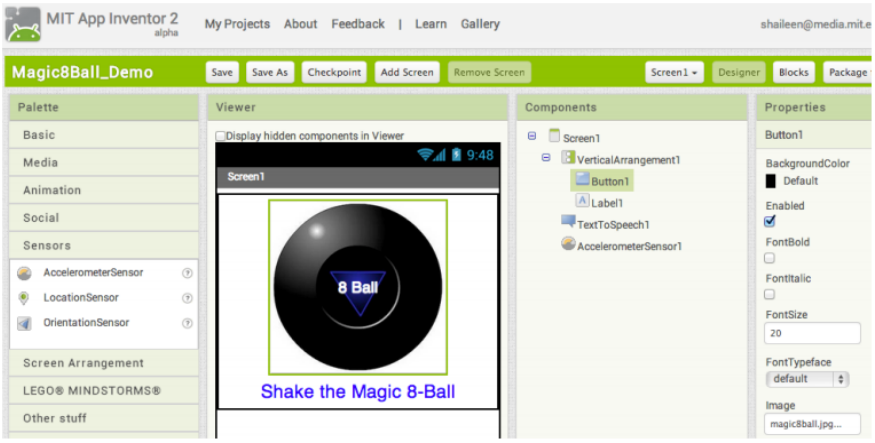


Figure 12: Designing Components of a Mobile App using the Designer [103]

Figure 13: Programming App Behavior using the Blocks Editor [103]

In this example, the user is creating a simple "Magic 8 Ball" app which provides an answer when the mobile device is shaken. The Designer interface is shown in Figure 12 and shows the components of the given app. We can see that there are five different components in the app: Button1, Label1, TextToSpeech1, and an accelerometer sensor. The first two elements are clearly visible on the "Viewer" panel in the center as they make up the Magic 8 Ball image and the text beneath it.

In Figure 13, we can see the behavior of the app defined in the Blocks Editor. These blocks use the accelerometer and text-to-speech components from the Designer and specify that when the accelerometer detects shaking, the text-to-speech component will say a random response from the list of responses given.

Unlike App Inventor, Sketchware and PocketCode have been designed for developing the mobile app on a mobile device. Similar to App Inventor, Sketchware decomposes the application into two categories: "View" and "Logic". The View section provides the user with an editor for creating the visual elements of the screen and the Logic section contains the code for event handlers and other programmatic components using a block-based programming language. PocketCode provides an environment in which users can create sprites and programmatically add behavior (like in many of these environments) and allows them to draw or use images to visually represent these sprites on the screen. In this way, PocketCode provides an experience more closely related to the approaches used in environments like Scratch [78].

2.1.3 Networking

There are a number of platforms used for teaching distributed algorithms including both generic simulation environments and custom tools designed for education. One common platform is Omnet++ [133, 132] but there are also a number of others [11, 136, 17, 128, 55]. Although simulation environments are very powerful and feature-rich tools, their complexity can sometimes provide a barrier to understanding and education. Consequently, there have been a number of other environments designed specifically for teaching distributed algorithms including PADS, SPLAY, and a Java Toolkit for Teaching Distributed Algorithms [8, 109, 115].

Omnet++ is a discrete event simulator often used for teaching networking [133, 132]. In Omnet++, users can program the behavior of nodes in the network including code for initialization (simulation start), finalization (simulation end), and an event-processing function executed when messages are received. Behavior is programmed using C++ and the network topology is defined in a declarative textual language called NED. As Omnet++ is a powerful simulation library, it can also simulate a variety of network topologies and programmable

nodes to enable users to implement and simulate distributed algorithms on their own computer. An example of the Omnet++ development environment is provided in Figure 14 and shows the network topology of the given example.

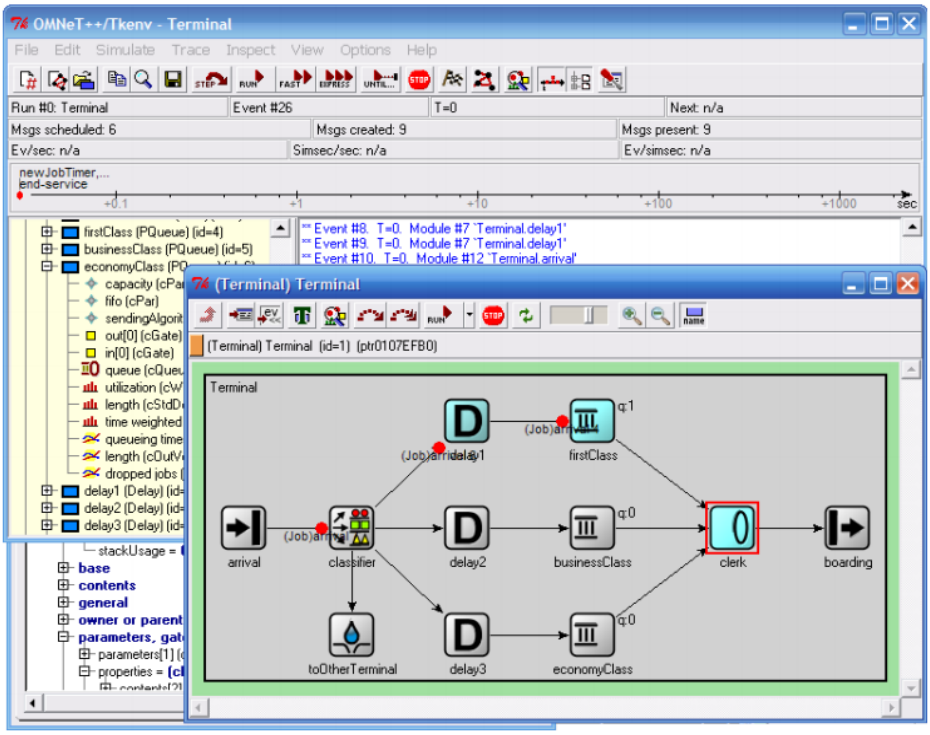


Figure 14: The Omnet++ Development Environment [133]

Omnet++ provides a very powerful environment for network simulation. Hence, educational use of Omnet++ would need to be targeting more advanced networking curriculum for users with programming experience rather than novice developers.

Another educational environment for teaching networking is the Playground of Algorithms for Distributed Systems (PADS) [8]. PADS provides a visual environment to create a distributed algorithm, define a topology, select a target deployment environment, and execute the given distributed application. Deployment environments in PADS includes simulation environments such as Omnet++. An example using PADS to define a topology for a BitTorrent network is shown in Figure 15.

In this example, the user has defined a network topology for a BitTorrent application with five peers, five routers, and a tracker. The network connections between the nodes are designated using visual connections in the PADS editor. The lower left contains a palette of valid nodes which can be added to the active network. At the time of this writing, PADS supports user-defined behavior by providing templates which can then be modified to add the desired behavior to the given node.

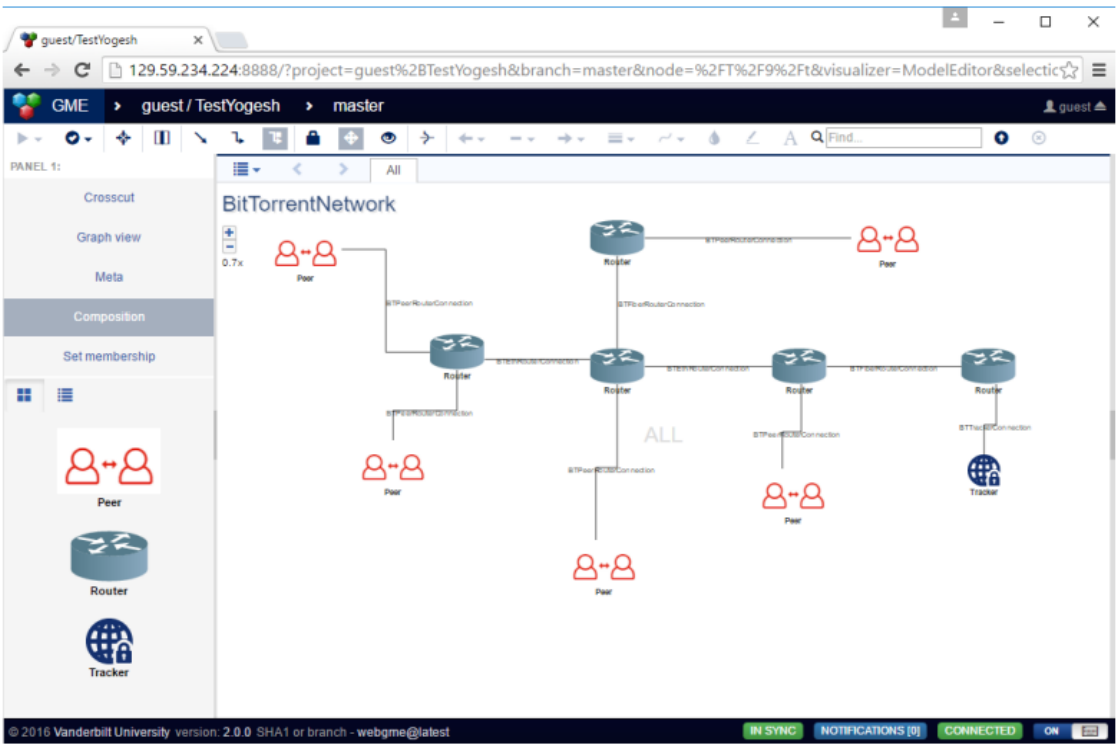


Figure 15: Defining a BitTorrent Network in PADS [8]

PADS is designed to target undergraduate students learning about distributed algorithms. As these students are expected to have some programming experience, they are currently able to implement the node behavior in a textual programming language then use the visual editor to define the network topology. After defining the desired network topology, this environment allows them to easily configure the network and deploy it across a number of different environments. In the classroom, PADS was found to be simpler for students when learning about distributed algorithms [8]. When developing behavior for the nodes in the network, PADS still targets a more advanced audience than most of the other education programming environments and assumes that users have experience in a textual programming language.

SPLAY is another environment for teaching distributed algorithms and systems [109]. In this environment, users are able to define distributed algorithms in Lua, test the algorithm locally, deploy the algorithm in a real world setting, then collect the logs and results of the execution. SPLAY also supports more advanced features natively such as RPCs and provides a sandboxed testbed for the deployment environments. An overview is provided in Figure 16.

In [115], a Java toolkit is presented for teaching distributed algorithms. This toolkit is a Java Applet which runs in the browser and enables users to create network topologies and

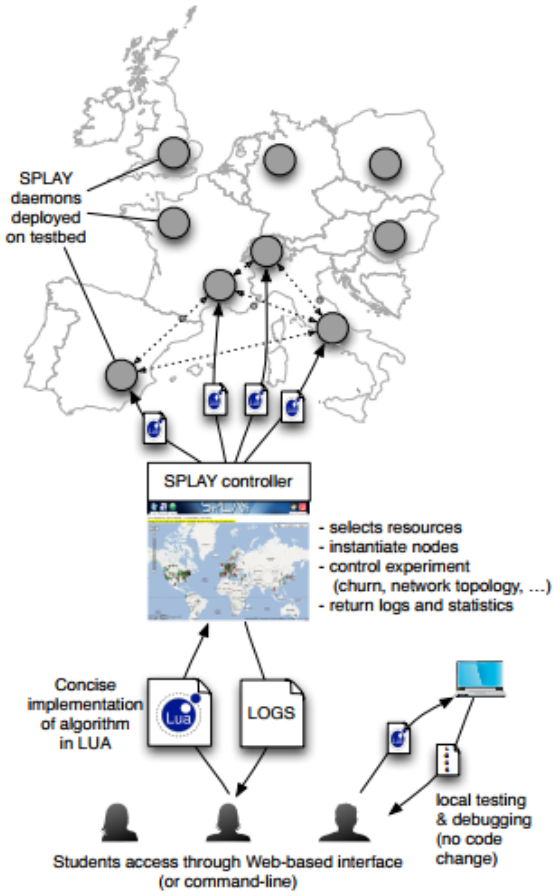


Figure 16: Overview of SPLAY [109]

define behaviors for the given nodes in Java and then execute the distributed system within the browser. After execution, the browser is able to provide a simple visualization of the network topology including color coded nodes and channels representing the state of the given entities. This includes indicating if the node is ready for execution (green), waiting for a message (red) or terminated (blue). Channel states are color coded to show if they are empty (gray), holding one or more messages (green) or empty with a waiting node (red). Also, a channel has a bulge at its connection point with the destination node providing a subtle visual cue about its direction.

Figure 17 shows a visualization of a simple three node network [115]. In this network, node 0 and node 1 are both running or ready for execution and node 2 is waiting for a message. The green channel from node 2 to node 0 signifies that the channel contains at least one message. In the top right, the visualization provides some basic debugging tools to control the execution of the given distributed algorithm and the bottom of the window contains status information during the execution of the system as well.

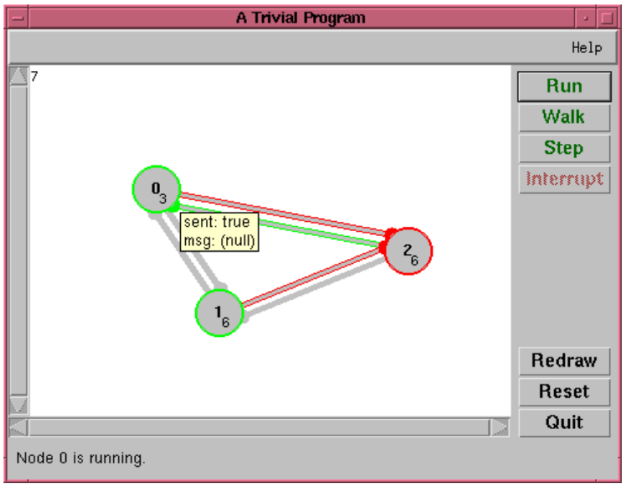


Figure 17: Visualization of the Java Toolkit [115]

Overall, this environment provides a very simple, yet informative, visualization of the network and is much simpler to understand during the execution than the more sophisticated tools mentioned earlier. Like some of the previous educational tools for learning distributed algorithms, this environment targets more advanced users and was used in graduate courses on distributed systems. As the target audience is older, users are still expected to implement the behavior of the nodes in the system in a textual programming language (in this case, Java).

2.1.4 Networking and Visual Programming

There has been limited work in providing networking capabilities to current educational visual programming environments. This work includes providing support for making requests from a visual programming environment to a REST endpoint [87, 52, 51], providing limited shared data storage in the cloud for use from visual programming environments[78], and limited messaging capabilities between clients [32].

Web APIs

Snap*!* provides support for interacting with external REST APIs through the use of an *http* block which provides basic support for accessing the many online resources. This is certainly a powerful capability; however, it has some limitations for adoption among novice programmers. Using this block, users are required to manually construct the URL with the expected query string parameters (including any required API keys) as well as manually

parse the response. Although this may not be too challenging for a more experienced user, this can be a serious hurdle for a novice programmer.

Figure 18: Requesting a weather forecast using the OpenWeatherMap API

In Figure 18, the *http* block is requesting the weather forecast using the OpenWeatherMap API [94] and storing the response in the "response" variable. After making this HTTP request, the "response" variable will be set to the raw JSON response of the OpenWeatherMap API. Next, the user will need to parse this nested JSON structure into something that is parseable by Snap![1], retrieve the relevant information from the response and then incorporate it into the application.

The transparency of this process provides a good learning opportunity for Snap! users; it will introduce marshalling and unmarshalling data, JavaScript Object Notation, and the basics of REST communication. Also, this API requires the user to provide a secret "app ID" to prevent abuse of the API which can motivate discussions about denial of service attacks and safeguards to prevent API abuse. Unfortunately, as the app ID required for this API is supposed to be secret, this project cannot be shared with other users without prompting the end user for an app ID on application start. Leaving the app ID in the source could result in other users stealing this secret ID and would also be teaching the students bad practices by ignoring privacy and security in a networking application.

Overall, the *http* block is very powerful and useful for introducing some topics of networking. This block enables experienced Snap! users to access third party APIs and incorporate them into their own applications. However, there is a relatively high barrier to entry when using this block with non-trivial APIs and also presents some challenges if users want to share their own projects if a secret ID is required for the given API.

Shared Variables

One of Scratch's many features includes support for storing variables in the cloud, i.e., "Cloud Data." Cloud variables can be shared among all copies of the project that are open at a given time. This enables students to build more interactive games as the games can store persistent information among all shared running versions. It has been shown to be quite powerful as it has been used to provide simple features like high score lists as well as more advanced functionality including multiplayer pong [104, 1].

[1]This could be provided by the teacher as a custom block so the students would not need to implement the JSON parsing

In these types of applications, all communication happens by polling the shared variables and responding as soon as they see a change. An example of this can be found in the given implementation of multiplayer Pong and is shown in Figure 19.

Figure 19: Multiplayer Movement with Cloud Data

This application first detects if it is the first or the second player. Then it starts the appropriate movement script which determines the shared variable it will update: the variable for the first or second player's x position. Both paddles then update their x position by polling their respective variables and updating their position. Figure 19 shows how the first player's paddle updates its position by rapidly polling the shared variable for the first player's x position and then updating its own position accordingly. This example demonstrates the power of shared variables in Scratch and how they have been used to develop non-trivial multiplayer applications.

Although cloud variables can enable users to build multiplayer applications, they are not a replacement for other valuable networking concepts, such as messaging. With only shared data, users are unable to develop any application using messaging and must emulate this by polling shared data. Polling is only one of many methods of communication and coordination. Combined with the limitations of Scratch's cloud data, cloud variables are insufficient for enabling users to develop sophisticated networking applications and teach computer networking concepts.

Network Messages

Previous versions of Scratch and Snap*!* supported creating a "mesh" using the IP address of another Scratch client. In this mesh, clients were able to share variables as well as share "broadcast" events. This enabled the clients to not only share data but also to trigger actions on another network machine. However, using the mesh to send and receive data could be somewhat complex and not the most accessible for novice programmers as it required the

user to manually serialize and deserialize the list. Figure 20 contains an example of how to send a list using the mesh provided on the Scratch Wiki.

Figure 20: Sending a List using the Mesh

In this example, the program is serializing the list by joining all the elements with a delimiter. When it is complete, it broadcasts "list is ready" which is then received by all programs in the mesh. The recipient then has an event handler for the "list is ready" event and accesses the serialized list using the "sensor value" block which provides access to the other programs' variables in the mesh. Using this serialized list, the program then manually deserializes it as shown in Figure 21 and can now use the list that was sent from the first client.

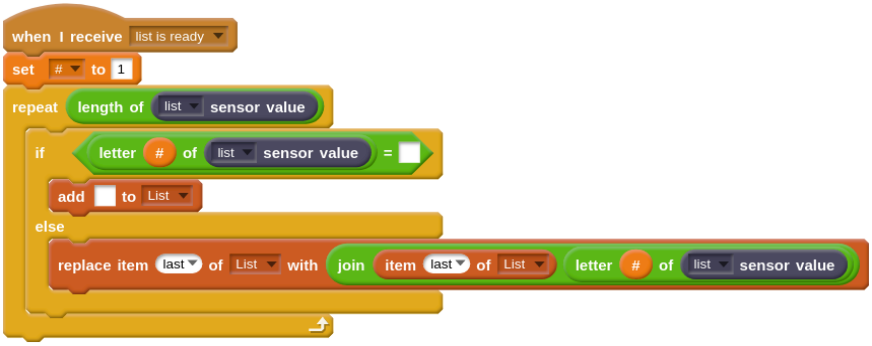


Figure 21: Receiving a List using the Mesh

The ability to broadcast events to other clients and share data between programs is very powerful. This facilitates developing applications which do not depend solely upon polling shared data but can actually be responsive and event-driven. Messages containing a data payload can be emulated by sending an event and then querying a shared variable. This enables users to build much more sophisticated applications than shared data alone.

This method has some weaknesses. The inability to send peer-to-peer messages makes some applications more complicated. Peer-to-peer messaging would need to be emulated by using an additional shared variable for storing the recipient's ID. When all the peers receive

the broadcast, they can then check this shared variable to see if they are the intended recipient. This introduces additional complexity as well as provides a bottleneck on the user's application by requiring only a single message to be sent at a time. Requiring manual synchronization by the user makes the use of lists and networking complex and presents a barrier for novice users.

Although the mesh functionality is not supported in the current versions of Scratch and Snap!, some similar functionality is provided in ScratchX, a version of Scratch supporting third party extensions [32]. ScratchX enables users to develop third party extensions to Scratch and then share them with other users but these projects cannot be run or shared on the main Scratch website. Scratch mesh-like functionality is available using the "Firebase Mesh" extension. However, unlike mesh functionality available in previous versions of Scratch and Snap!, this extension provides custom blocks for broadcasting to the mesh and receiving broadcasts sent over the mesh. Firebase Mesh provides only the broadcast functionality of the previous mesh functionality of Scratch.

Extensions have also been created for accessing some real world data including weather data and tracking information for the international space station [52, 51]. This data is accessible to students using custom blocks providing access to the given data sources. The following figures provide specific examples of extensions accessing real world data.

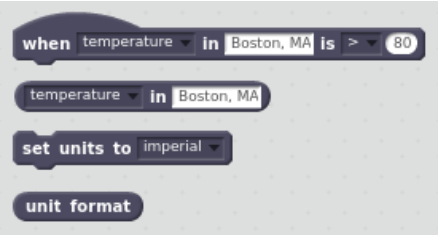


Figure 22: Accessing Weather Data with the Scratch Extension

Figure 22 shows the blocks provided by the weather extension for Scratch. These blocks allow a user to get weather information using the OpenWeatherMap API [94] for a given area as well as set the unit format and trigger events on certain weather conditions. The dropdown for the "temperature in" block allows the user to request other types of weather information for a region including wind speed, cloudiness and humidity. By making weather data available, this application enables users to add more interactivity into their applications and access a third-party API.

The blocks provided by the International Space Station tracking extension is provided in Figure 23. This enables users to develop applications which utilize the location of the International Space Station and is demonstrated in the example app for the extension [51]. These blocks allow users to get the current latitude, longitude, altitude and velocity of the

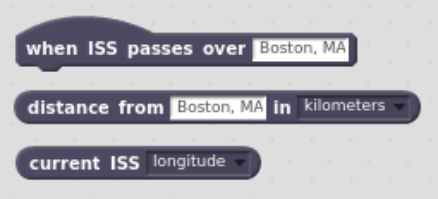


Figure 23: Accessing International Space Station Data with the Scratch Extension

ISS, compute the distance between the ISS and a location on Earth as well as invoke a set of scripts when the ISS is passing over a location. Like the weather extension, this extension enables users to leverage some network functionality by interacting with another third-party API and utilize the resultant information in their application. Unfortunately, both of these Scratch extensions are only allowed in ScratchX and cannot directly benefit from the large community on the Scratch website.

2.2 Approach

Making distributed programming accessible to novices requires the careful selection of programming abstractions, including visual representation, to enable the users to build distributed applications. The level of abstraction is very important for promoting user learning. Selecting too high of an abstraction will hide the underlying distributed concepts from the users whereas designing too primitive of abstractions can hinder easy usage by novices. We present the following abstractions to novices for developing distributed applications: *messages*, *Remote Procedure Calls* (RPC), and the *Room*.

2.2.1 Messages

Peer-to-peer communication capabilities are provided using messages. Messages are similar to the concept of an *Event* already present in other blocks-based programming environments such as Snap*!*. Users can broadcast custom events to execute all scripts listening for the given event. An example of an event in Snap*!* is given in Figure 24.

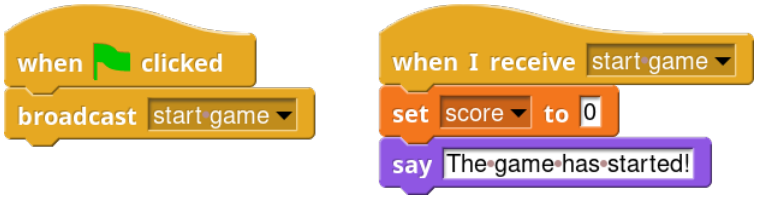


Figure 24: Event Example in Snap*!*

Similar to events, messages initiate the execution of corresponding scripts starting with the respective handler. Unlike events, messages enable users to communicate across devices and not solely within a single user project. Messages also can contain a structured data payload. This significantly increases their flexibility and capability as they can be used to share information rather than solely triggering execution.

We define four different ways of interacting with messages. Messages can be sent and received; sending can be performed either synchronously or asynchronously. Furthermore, users can send response messages after receiving a message. The corresponding visual representations for receiving messages and asynchronous sending are shown in Figure 25.

Figure 25: Receiving and Asynchronous Sending of Messages

In this example, the `when I receive` block is triggered when the given project receives an "update position" message. This message is expected to contain two fields, "x" and "y." The block exposes the contents of these fields by providing a variable for each of the expected fields. When a message is received, this block will automatically retrieve the contents of each field and assign it to the corresponding local variable to be used in the subsequent connected blocks.

Messages can be sent asynchronously using the `send msg` block. Like the `when I receive` block, this block contains a drop-down specifying the type of the message that will be received. After specifying the type of message, input fields are created for each of the expected fields following the drop-down. In the example from Figure 25, two inputs are created – one for each field in the "update position" message type. Each input is labeled with light gray text displaying the field name for the given input.

The last field in the `send msg` block is the target of the message. Unlike events, messages support peer-to-peer communication along with broadcasting. In this example, we are broadcasting the message to all users in the project.

Figure 26 shows the blocks for synchronous messaging. As the name suggests, `send msg and wait` block is used to send a message and wait for a response. The rest of the block is very similar to the asynchronous messaging block; both specify the message type, content and recipient. However, this command will send the message and block execution until the recipient responds to the message rather than simply sending the message and resuming

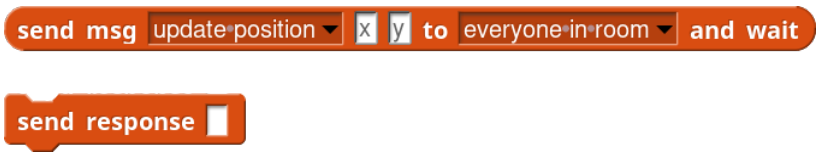


Figure 26: Synchronous Messaging

execution. The `send response` block is used to reply to messages sent synchronously. As this block is used to respond to a message, it can only be used in a message handler script.

As the messages contain structured data payload, it is important that messages are typed. A message type is composed of a name and a list of expected fields. For example, the blocks in Figure 25 are sending and receiving messages which are of type "update position." The "update position" message type defines two fields, "x" and "y." The message type allows the blocks to dynamically update according to the expected fields and provide a simple way to enforce the structure of the data payload. Message types also simplify the project code as the message handler specifies the message type for which it is listening. This enables the user to easily separate the concerns for each dedicated type of message used in the distributed application rather than manually handling each message in a single message handler.

Figure 27: Message Type Creation

An example of how a message type can be created is given in Figure 27. In this example, a message type called "Tic-Tac-Toe" is being created. This message type has two fields: "row" and "column." Fields can be added or removed using the small arrows following the fields.

Figure 28: Tic-Tac-Toe Message Handler Block

Figure 28 shows a message handler for the newly created message type. Once the "Tic-Tac-Toe" message type has been defined, it will be available in the messaging blocks as one of the options in the drop-down menus. After selecting the message type, the message handler

block dynamically updates to provide variables for each of the expected fields. Both the asynchronous and synchronous messaging blocks also contain a message type input which will include this new message type. Selecting the message type will dynamically update the given block to provide inputs for the respective fields of the message type as well.

2.2.2 Rooms

Another important distributed programming abstraction is the *Room*. The Room is a virtual network abstraction and consists of *Roles*. A Role is a named client in the given distributed application and is analogous to a project in most other blocks-based programming environments. The content of each Role is independent of one another; each has its own stage, sprite, scripts, and other project artifacts. A Room is created automatically for each user project. Grouping sub-projects into Rooms enables the environment to provide more support and feedback to the programmer. The environment is able to monitor the state of the other Roles in the room and configure the environment accordingly. One such example is the `send msg` and `send msg and wait` blocks. The recipient field of these blocks is populated with the names of the Roles in the user's Room. This prevents simple typographical errors and should help avoid mistakes when entering the address of the intended recipient.

Introducing the Room enables the environment to provide an easy interface for viewing and modifying the Roles in the room. This includes visualizing the Roles in the project, checking the occupants of each Role as well as providing a natural interface for providing debugging capabilities for the networking aspects of the application. From the Room editor, users can also create, duplicate, rename, and remove Roles. Additionally, users can also invite other users to specific Roles from this interface.

Figure 29 shows the room editor while viewing a project with four Roles. These Roles are "alice," "bob," "eve," and "super eve." Currently, "bob" is the only occupied Role and is occupied by "brian." The project name, "Caesar Shift," is displayed in the center of the editor. Clicking the + button will allow the user to create a new Role in the project; clicking on a Role will allow the user to rename, duplicate, remove, or invite another user to it. Users can also move between different Roles by clicking on the desired Role. Although this example contains four Roles, the names and number of Roles contained within a project's Room is configurable and can be modified to fit the goals of the desired application. For example, a Tic-Tac-Toe application may only have two Roles, "X" and "O," in its associated Room.

Introducing the concept of a Room also facilitates the simplification of network addressing. When sending messages within a single Room, the recipient can be specified by using

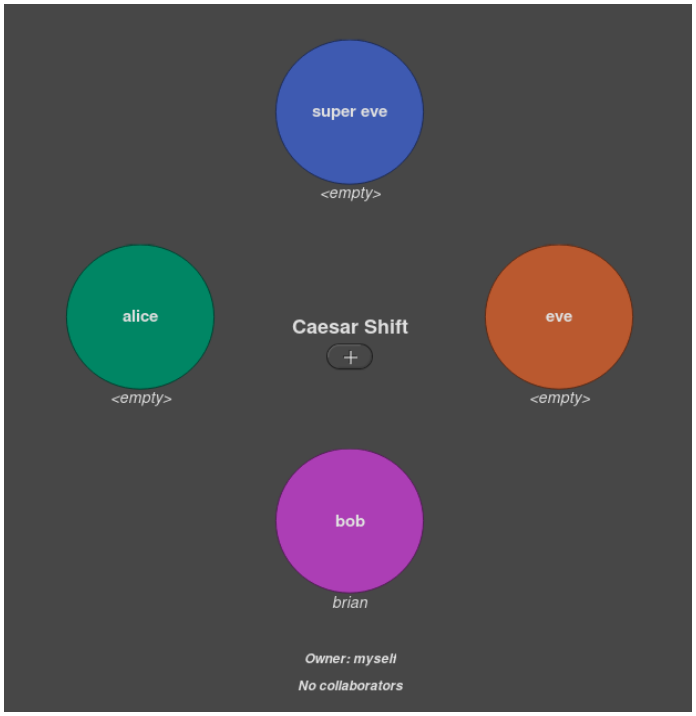


Figure 29: Viewing and Editing the Room

the Role name. When sending a message between Rooms, the recipient must be specified using the Role's fully qualified name, the "Public Role ID." The Public Role ID for a Role is the Role name, Room name and owner name joined by the @ symbol and was created to be human-readable while still uniquely identifying a single Role. For example, in Figure 29, the "alice" Role could be addressed using "alice" when messaging from within the Room or as "alice@Caesar Shift@brian" globally.

Along with providing a natural way of message addressing for novices, this approach also can serve as a simple introduction to concepts like the need for fully qualified names and private vs public IP addresses. The Room also enables the blocks to automatically configure themselves to assist the user when entering the recipient address for the message. When using the `send msg` block, the "target" field is a drop-down menu which contains the names for every available Role in the given project's Room. Figure 30 shows an example using the `send msg` block from the "super-eve" Role within the project from Figure 29. In this example, the drop-down for the message recipient is populated with the names of all the other Roles in the Room, "alice," "bob," and "eve," as well as the two different broadcasting options: "others in room" and "everyone in room."

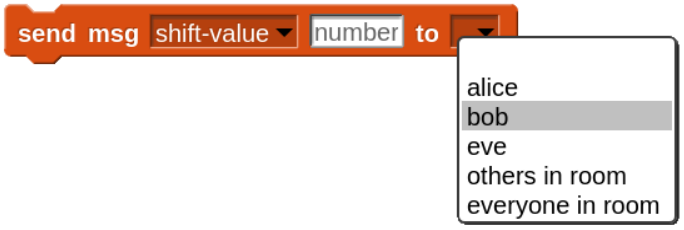


Figure 30: Sending a message within a Room

2.2.3 Remote Procedure Calls

Remote Procedure Calls (RPCs) are the highest level of abstraction among the provided networking abstractions. RPCs enable users to invoke functionality defined on the host server and can be used to provide access to external data sources and provide scaffolding for developing more complex applications such as Battleship or arbitrarily large Tic-Tac-Toe variants. Easy access to real-world data should make programming more relevant and engaging for a wider audience of students. Furthermore, leveraging server-side logic to provide scaffolding can be used to promote early successes among novices.

2.2.4 Services

Similar RPCs are grouped into categories called *Services*. Examples of such Services are "GoogleMaps," "Battleship," and "Weather." RPCs can be stateful in which their state can be shared amongst the other RPCs for the given Service. Additionally, state can be global or unique to each request's originating Room. For example, a Service providing scaffolding for developing a multiplayer Battleship game will likely want to maintain a unique state for each originating Room as each Room is a different instance of the game. Services providing access to an external resource will likely benefit from maintaining a cache in their global state to reduce the number of requests to the given external resource.

Maintaining state also enables some Services to be simplified significantly and accessible to younger users. One such example can be found in a Service providing access to map images. Maps can be valuable for visualizing real-world data and can be understood by very young users. However, utilizing maps can become rather complex as users will need to perform a coordinate transformation from the screen coordinates to the corresponding latitude and longitude on the map. This is not a trivial task for novice programmers as this transformation requires the user to understand the type of projection used by the underlying map (such as the Mercator projection) and implement the given projection so the user can transform the desired points. Allowing the Services to maintain state enables the given

mapping Service to record the user's current map region and expose RPCs to perform the transformation from screen coordinates to a geographic coordinate system.

Figure 31: RPCs for Coordinate Transformation

An example of RPCs for performing the given map projections is provided in Figure 31. Both blocks are invoking RPCs from the "GoogleMaps" Service. These blocks provide functionality for conversion between the image coordinates and geographic coordinates.

2.2.5 Invoking Remote Procedure Calls

As the name suggests, Remote Procedure Calls enable the users to invoke functionality in a remote location. Individual Remote Procedure Calls may be predefined or user defined and invoking RPCs is performed using the `call RPC` block. The block has zero or more named input arguments (passed by value) and an optional return value. To promote accessibility to novices, the block evaluates synchronously and is used similarly to blocks executing locally.

The `call RPC` block is designed to promote "tinkerability" and minimize user error when using RPCs. The first input for each block specifies the name of the Service used by the given block. After specifying this input, the second input is populated by all the valid RPC names for the given Service. Specifying the RPC will update the rest of the block according to the method signature for the given RPC. That is, the block will dynamically add an input field for each expected input and each input is labeled with light gray text like the `send msg` block.

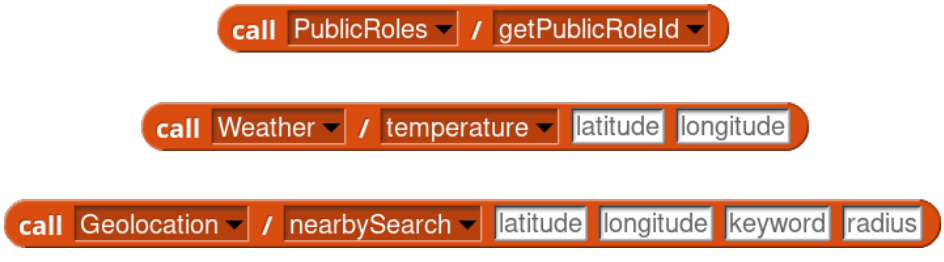


Figure 32: Dynamic Blocks for invoking RPCs

Figure 32 shows an example of using the `call RPC` block to invoke various RPCs. The first block is invoking the `getPublicRoleId` RPC from the "PublicRoles" Service. As this RPC does not accept any inputs (it simply returns the fully qualified name for the given

Role), there are no inputs following the RPC field. The second block is using the "Weather" Service to request the temperature at a given latitude and longitude. This is reflected in the block inputs as it has two additional inputs with "hint text" for the latitude and longitude inputs. The final block is using the `nearbySearch` function from the "Geolocation" Service. This RPC expects four inputs: "latitude," "longitude," "keyword," and "radius."

Along with providing immediate results to the user, RPCs can leverage the message abstraction to stream data back to the user. That is, invoking the RPC can simply notify the server to start sending messages to the address of the request origin. This enables RPCs to return large amounts of data as well as introduces another approach to programming to the user.

Sending messages also simplifies any transformation or manipulation required to use the data. When returning structured data in a block environment, one of the best ways to represent the data is by creating a two-dimensional list in which the nested lists are key-value pairs. For example, if the data contained a field named "age" with a value of 21, the nested list would have two items, "age" and 21 (in that order). Representing each field and value as a list allows us to then create an entire data structure by creating a list of these field and value pairs. If we want to return a list of these structured data elements, then our list gains another dimension as it will be a list of these lists of name and value pairs. Sending messages, instead of returning a 3-dimensional list, allows us to avoid explaining the necessary complexities to novice programmers by simply sending messages which are received one at a time. Each field is available as a script variable on the message handler block. This enables users to process each of the data elements without even using a single list.

Figure 33: Initiating the Sending of Earthquake Messages

Figure 33 and Figure 34 provide an example of an RPC in which the results are returned to the user using messages. Figure 33 shows the block invoking the RPC requesting the earthquakes for a given region (which optional inputs for additional constraints such as "startTime" and "endTime"). After invoking the RPC, the server will retrieve the given earthquakes matching the user's request and begin sending one message per earthquake to the origin of the request.

The user then can handle each earthquake as received as shown in Figure 34. Each message contains information about a single earthquake and the fields of the message represent the individual fields about the matching earthquake. In this case, the message handler is

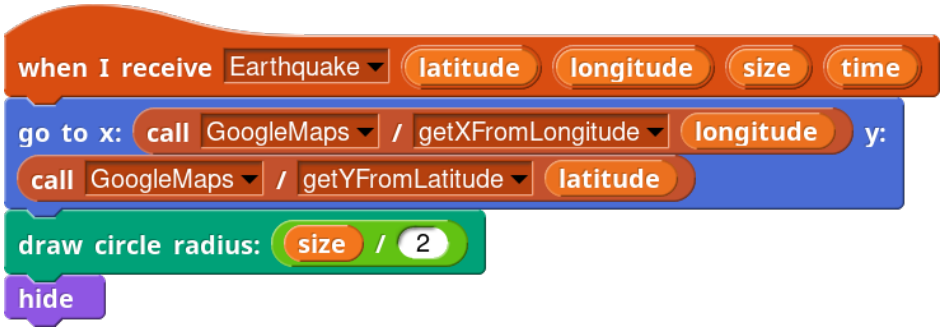


Figure 34: Receiving Earthquake Messages

performing a couple steps to visualize the given earthquakes. First, it is moving to the x and y coordinates corresponding to the latitude and longitude of the given point. Then it will draw a circle proportional to the magnitude of the earthquake. Streaming data from an RPC enables novices to develop applications using large, real-world data-sets without manipulating multi-dimensional lists.

2.2.6 Error Handling

When invoking an RPC, errors can be handled using the `error` block. This block is automatically set to the error value of the last RPC invocation. If the last RPC invocation resulted in no error, then this block will be empty. An example using the `error` block is shown in Figure 35.

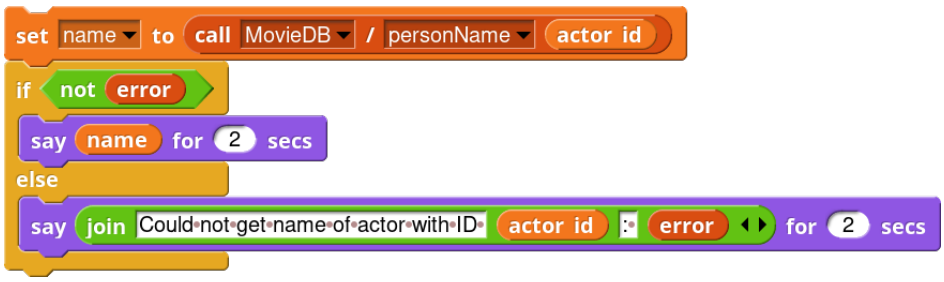


Figure 35: RPC Error Handling

In this example, an actor name is being retrieved for a given actor ID using the `personName` RPC from the "MovieDB" service. After invoking the RPC, the `error` block is used to check that the RPC completed successfully. If so, the given sprite will say the name of the requested actor. Otherwise, the sprite will say that the name could not be retrieved along with the specific error message.

It is worth noting that this is a different approach to error handling than the current standard in blocks-based programming. Errors usually result in current script terminating

and highlighting itself in red. When programming on a single machine, this is understandable as the cause of the error is on the single machine and should be corrected. When relying on other devices, such as with invoking RPCs, errors may occur on the remote device independent of the local code. This emphasizes the importance of resiliency and error handling in distributed applications as errors cannot be guaranteed to be avoided – even if all the local code is correct.

2.2.7 User-Defined Procedures

Although predefined procedures can be valuable for interacting with external data sources, it is important that users are able to create and execute custom remote procedures. Not only does this allow the users access to the concept of RPCs, but it also enables them to gain concrete experience with network latency and batching network requests. Supporting user-defined RPCs enables users to evaluate the performance differences between executing RPCs serially or batching usage of predefined RPCs into a function which is then executing in a single network request on the server.

Invoking custom procedures are supported through the use of the `call` RPC in the "Execute" Service. This RPC accepts an anonymous function ("ringified" blocks) as an input and then executes it on the server. A simple example is given in Figure 36. In this example, the user is executing a very simple function which simply returns the text: "hello from the cloud!"

Figure 36: Simple User-Defined Procedure

One strength of the provided abstractions for distributed programming lies in their flexibility and robustness. Although this is a very simple example of user-defined procedures, it demonstrates the flexibility of the Remote Procedure Call abstraction. Additionally, this allows users to start thinking about not only what the program should do but the environment in which it should be executed. User-defined procedures are discussed in more detail in Chapter 3.

2.3 Debugging Distributed Applications

It is important that novice programmers are supported during the entire development cycle of a distributed application. This includes providing scaffolding during the troubleshooting and the debugging process. When developing distributed applications, debugging can

become challenging as users need to consider the coordination of all the distributed clients within the given application rather than just the local execution of the program. This additional complexity should be alleviated through auxiliary concepts and interfaces accessible to the novice programmer.

These concepts must be powerful enough to provide insight to the overall behavior of the distributed application. The concepts must also be accessible through the design of natural, intuitive interfaces. As the target users are novice programmers, it is important that the abstractions have a low threshold and can be used with minimal direct instruction. To this end, we have designed capabilities for gathering insight into the distributed behavior of the given distributed application.

2.3.1 Room Debugger

Introducing the Room concept enabled the development environment to provide additional capabilities to the user which simplified various networking challenges. These included addressing and discovery over the network. The Room Debugger extends this scaffolding to include the runtime behavior between the given applications. It introduces the concept of a network trace and enables the users to capture and replay these traces. This enables users to first record a network trace during the execution of a distributed application and then replay the messages captured during the recording.

When using the Room Debugger, users must first capture a network trace. As one of the main design goals is to assist in debugging a faulty application, it is beneficial to ensure that the code does not change while trying to understand unexpected behavior. To prevent code changes during a captured trace, the user's Role is placed into a read-only state temporarily suspending any lively features of the environment. When recording the network trace, all messages sent within the Room are collected along with additional useful metadata. The message data includes the message type, data payload, and intended recipient. Additional metadata includes the actual recipients and the timestamp.

After capturing a network trace, users are able to replay the captured messages. During replay, users are presented with natural controls designed for accessibility and minimal directed instruction. To promote accessibility, we have designed an interface reminiscient of a media player for controlling the network trace replay. A slider representing the timeline of the network trace is presented with buttons for playing the replay at normal speed, single stepping, and jumping to the beginning or end of the replay. Messages are marked on the slider and color-coded to match the color of the sender.

The Room editor is overlaid with the last captured messages with respect to the current

time of the replay. Messages are displayed using a connection from the sender to the recipient and an icon color-coded to match the sender. To reduce cognitive load, the messages display the message type but none of the contents; the contents can be investigated by clicking the message icon. To simplify viewing sequences of messages, the Room Debugger can be configured to display multiple messages simultaneously. When viewing multiple messages simultaneously, the messages are labeled according to their relative ordering.

As understanding the application execution during the trace is the primary goal, the project is also read-only during the replay. Understanding the relationship between the network interactions and the local execution is important when debugging a distributed application. When the latest replayed message was received by the current Role, message handlers for the given block are highlighted. Highlighting the message handlers augments the network visualization by indicating the scripts executed as a result of the given message. To prevent confusing behavior when debugging during a network trace, messages received during the replay are ignored.

An example network trace replay is given in Figure 37. This project contains four Roles: "brian," "alice," "bob," and "eve." The replay controls inspired by the slider commonly used in media player applications are shown in the bottom of the figure. Messages are marked on the slider using lines colored after the corresponding sender's Role.

This example demonstrates a simple application based on the mesh networking project described in the next section. In this project, the "brian" Role is sending a message to the "eve" Role through the "alice" and "bob" Roles. The message is sent clockwise around the Roles in the Room until the "eve" Role receives the message. This behavior is apparent from the replay of the network trace shown in Figure 37. As this example is displaying multiple messages simultaneously, the messages are numbered according to the order in which they were sent.

2.4 Illustrative Examples

The expressiveness of the provided networking abstractions can be easily demonstrated through examples of distributed programs. These abstractions provide a low threshold for building simple projects yet still support the development of complex ones. In this section, we present examples demonstrating the simplicity to build basic distributed programs as well as the potential of the abstractions to develop more sophisticated applications.

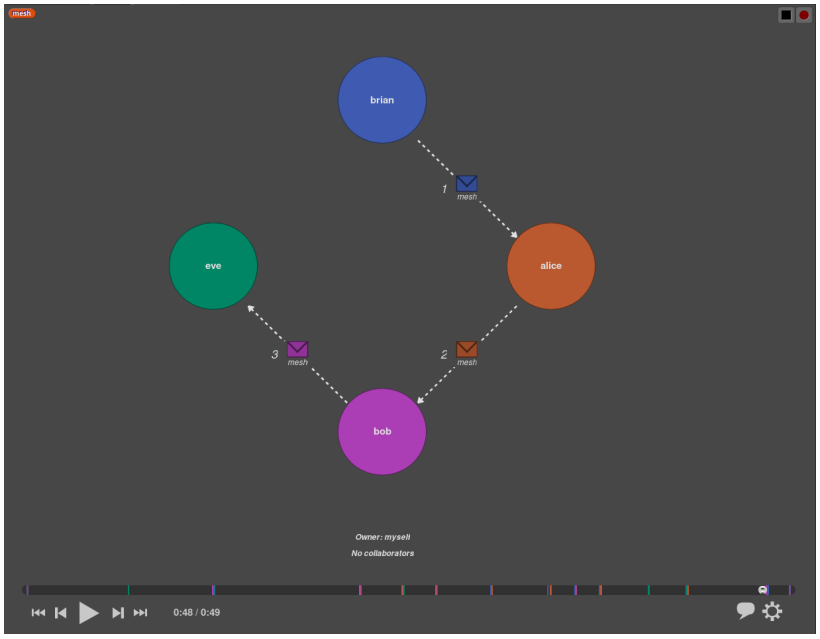


Figure 37: Replaying Messages in a Room

2.4.1 Basic Distributed Applications

One of the strengths of the provided networking abstractions lies in their low threshold. With just a few blocks, novices can develop an application demonstrating interesting concepts such as mesh networking or basic client-server architecture. These simple applications can make programming more social and provide more opportunities for discussions about distributed concepts such as resilience, security, and encryption.

Chat Application

Chat applications provide an easy introduction to distributed programming. They require virtually no coordination between clients and can be achieved with a very simple protocol. Using the provided programming abstractions, there are a number of different ways to implement a chat application. The entire application can be implemented within a single Room with a fixed set of Roles or the individual client applications can be implemented as individual applications which use the Public Role IDs for inter-room communication. Using inter-room communication is more flexible and allows the chat application to support a dynamic number of clients. However, supporting these clients introduces some additional complexity as it requires the logic for managing client connectivity including the connection

and disconnection of clients. As this example is highlighting the basic distributed applications, we will present a chat application supporting a fixed number of clients defined as the Roles in a single Room.

In this chat application, each Role can chat with the other Roles in the Room. A chat message contains both the message and the username of the person sending the message. That is, the applications define a custom message type called "chat" which contains two fields: "name" and "message." After defining this message type, the Role can easily send a chat message to the Room using the `send msg` block as shown in Figure 38. In this example, the message is broadcasted to "everyone in the room." This ensures the user's chat message will be sent to every Role, including that of the sender.

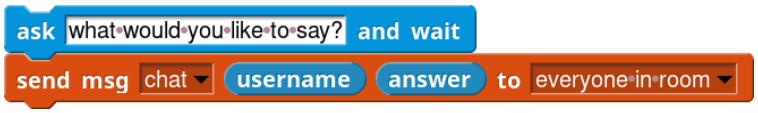


Figure 38: Sending Chat Message

In addition to sending messages, each Role will need to listen for the given message type and display the content of the chat message. Although each Role can certainly display the messages in a multitude of interesting and creative ways (such as printing each message in a scrolling window on the stage), a basic technique is shown in Figure 39. In this figure, the given sprite simply displays the most recently received chat message. The message is formatted with the sender's name preceding the message contents and separated with a colon.

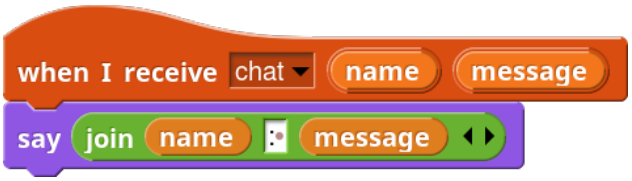


Figure 39: Receiving Chat Message

This is an example of a relatively minimal distributed application. Messaging in this application is simple as there is only a single message type, "chat," and messages are always broadcasted to all Roles in the given Room. Each Role displays the last chat message and every Role shares the same identical code. Although this example is simple, it demonstrates how a basic distributed application can be created using only four command blocks. This simplicity highlights the low threshold for developing distributed applications using the provided abstractions.

Mesh Networking

Our second basic example is a mesh networking application. Like the chat project, this example demonstrates the low threshold for developing distributed programs. Both examples provide an accessible, working example demonstrating some basic distributed concepts. These examples are also a foundation for motivated students to explore more complex distributed computing concepts such as resiliency and security.

In this example, users can form small mesh networks and send messages to other users in the network. First, users will define the physical layer of the mesh network by setting the fully qualified address of the application to which it can send messages. We expect the applications to form a ring in which one has a connection in the physical layer only to the subsequent node in the ring. After defining the physical layer, each application can send a message to another node in the circle using a logical address, such as the username of the intended recipient. Each individual application then contains the logic for routing each message with respect to the logical address of the intended recipient.

In this example, "next node" is the fully qualified address of the connected node of the given application. This represents the connected node in the physical layer of the mesh network. Every "mesh" message contains three fields: "sender," "receiver," and "msg." The "receiver" field defines the logical address of the intended recipient. The "sender" field defines the originating sender and "msg" defines the contents of the message.

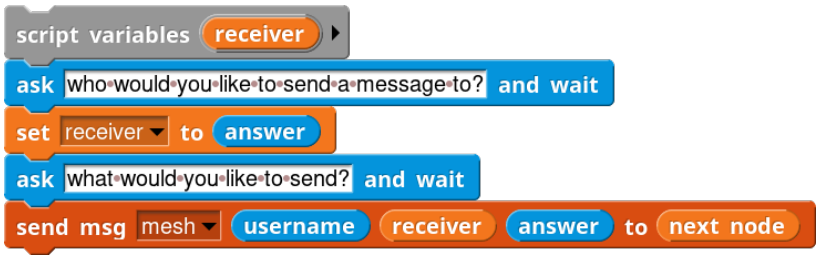


Figure 40: Sending Mesh Message

Figure 40 shows an example of sending a message in the mesh network. When sending a message, the user must provide the logical address of the intended recipient. In this example, the logical address corresponds to the occupant's username. The user also needs to provide the actual content to send, "msg". After constructing the message, the application sends the message to "next node" as this represents the only connection in the physical layer of the network.

Figure 41 shows an example demonstrating the routing logic in the mesh network. When the application receives a message from the mesh network, it will first check if the current

user's username matches the intended recipient. If so, then the message will be displayed to the user. Otherwise, the message will simply be forwarded to the next node in the mesh.

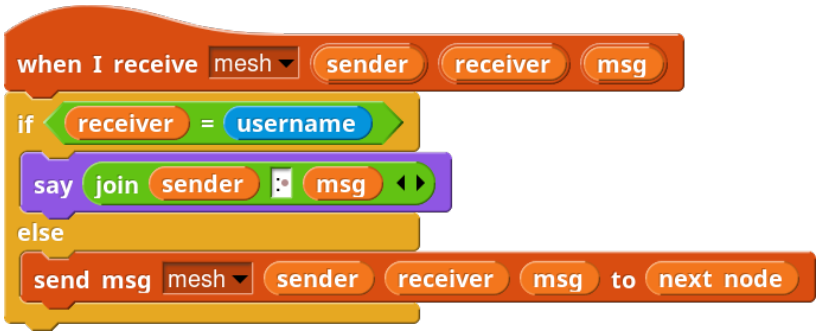


Figure 41: Receiving Mesh Message

2.4.2 Advanced Distributed Applications

In this section, we demonstrate the potential of the provided distributed programming abstractions for developing sophisticated distributed applications. This includes a demonstration of developing higher level messaging patterns, such as Publish-Subscribe. Additionally, the combination of first class functions, lists and message passing enables users to develop more sophisticated applications including MapReduce.

Publish-Subscribe

Publish-subscribe is a messaging pattern which decouples the producers of messages from the consumers of the given content. The consumers of content will "subscribe" to various types of messages and producers will "publish" messages. When the producers of content publish messages, these messages are sent to all consumers which have subscribed to the given type of message. There are generally two different approaches to specifying messages to be received: topic-based and content-based. The topic-based approach requires each message to contain not only content but also to specify a topic. Subscribers then can subscribe to specific topics, and they will then receive all messages with the given topic. Content-based filtering of messages enables the consumers to subscribe to messages based on the contents of the given message.

In this example, we will be presenting an example of topic-based publish-subscribe using the presented distributed programming abstractions. There are two different message types in this example: "publish" and "subscribe." As we are using topic-based message selection, the "publish" message has two fields, "topic" and "content." The "subscribe" messages have

a “topic” and “id” field. The “topic” field specifies the topic to which to subscribe and the “id” field contains a Role’s fully qualified address.

We will be using a broker to orchestrate the selection of messages to be sent to the consumers. The broker maintains a dictionary of subscriptions for each topic stored as key-value pairs. That is, each item in the “subscriptions” list variable (shown in Figure 42) contains the topic and a list of all subscribers for the given topic. When a “subscribe” message is received, the broker searches for the list of subscribers for the given topic and adds the new subscriber. If it finds that the given topic has no subscribers, it will create a new entry in the subscriptions list for the given topic.

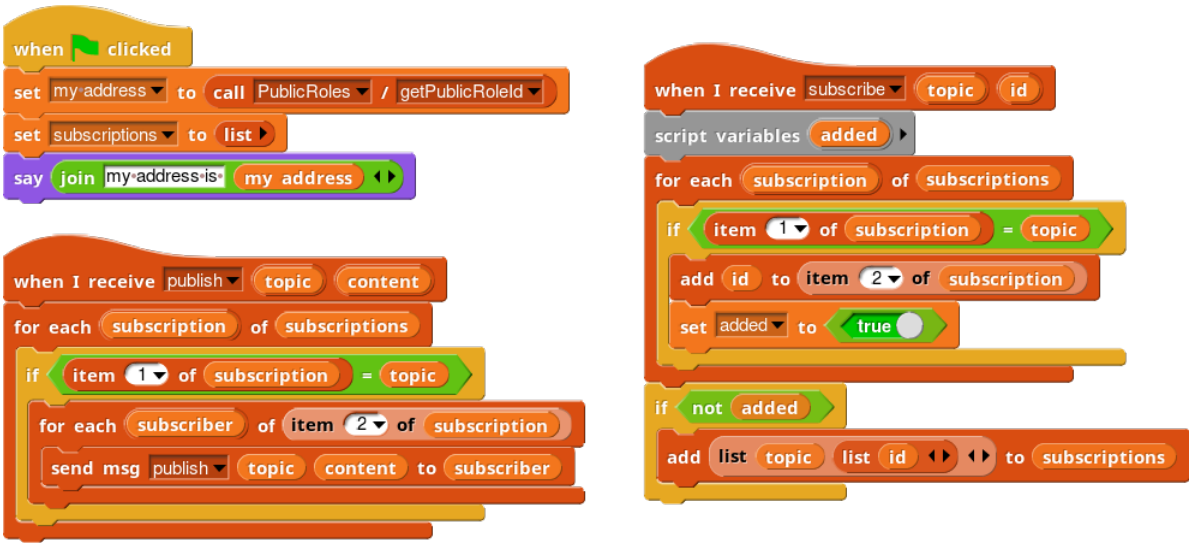


Figure 42: Publish-Subscribe Broker

When the broker receives a “publish” message, the broker searches for the list of subscribers for the topic of the given message. If it finds a list of subscribers for the given topic, the message is forwarded to each of the subscribers. The complete code for the publish-subscribe broker is shown in Figure 42.

An example application using this publish-subscribe broker is shown in Figure 43. In this example, the application is a message consumer. Pressing the “s” key prompts the user for a topic to which to subscribe. The provided topic is then sent in a “subscribe” message to the publish-subscribe broker along with the user’s fully qualified address (Public Role ID). Messages matching the given topic will then be received by the “publish” message handler. This will result in the received message topic and content being printed by the current sprite.

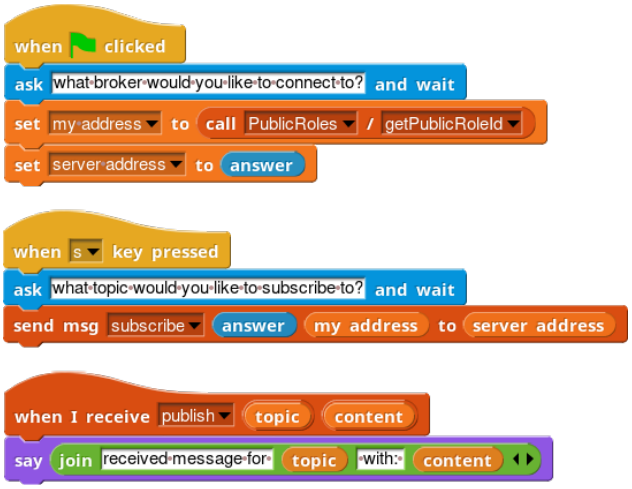


Figure 43: Example Publish-Subscribe Client

MapReduce

MapReduce is a common distributed computing framework used for data processing on large datasets [33]. In this model, computation is structured as *map* and *reduce* tasks which enable the entire computation to be easily parallelized and distributed to the nodes which contain the data. At a high level, data is stored in a distributed environment and then the map and reduce tasks are sent to the nodes containing the relevant data and performed on the local data. The map operation results in key-value pairs which are then grouped by key. The reduce operation is performed on the data collected for each given key.

In our basic blocks-based implementation of MapReduce, there is a master node which a number of connected worker nodes. Data can be stored on the worker nodes in the cluster and MapReduce jobs can be submitted which process data stored in the cluster and store the results back to the cluster. For brevity, we will focus predominantly on the client interactions with the cluster, the storage of data in the cluster and evaluation of the map step on the worker nodes.

Figure 44: Storing Data in MapReduce

Data can be stored on the cluster using a "store" message as shown in Figure 44 and submit jobs to process the data using the "job" message type. This message type contains the job name, inputs, outputs and, of course, the actual map and reduce tasks. An example of submitting the canonical "word-count" MapReduce job is shown in Figure 45. This job is run on the "books" data (already stored on the cluster) and writes the results back to the cluster as "word-count-results". The map step takes text and splits it into words and

returns key-value pairs where the key is the word and the value is "1". The reduce step takes a key-value pair where the key is a word and the value is a list of counts and returns a key-value pair where the key is the input word and the value is the sum of all the counts.

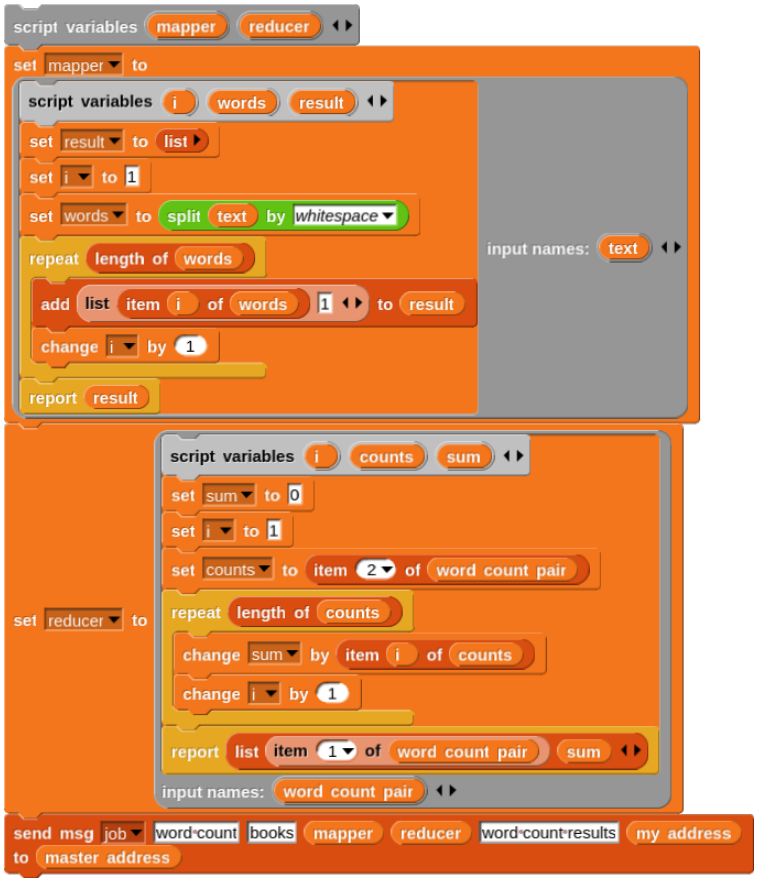


Figure 45: Submitting MapReduce Job

When data is stored on the cluster, the master node distributes the data across all the data nodes for storage as shown in Figure 46. The data is first partitioned into groups where each group corresponds to the data to be stored on a single worker node. Then a partition is sent to each connected worker node starting from a random index in the node list; this prevents the overloading of a single node when receiving lots of small data storage requests.

After storing data on the cluster, data can be processed by submitting client jobs consisting of a map and reduce task as shown in Figure 45. There are a number of steps in running a job in our implementation; for brevity, we will only present the evaluation of the map step on the connected worker machines. When the master node receives a job message, it first sends the map step to the connected worker nodes as shown in Figure 47.

Executing the map step on the workers is relatively straight-forward. First, the worker looks up the data stored under the name of the designated input. If the worker has any of

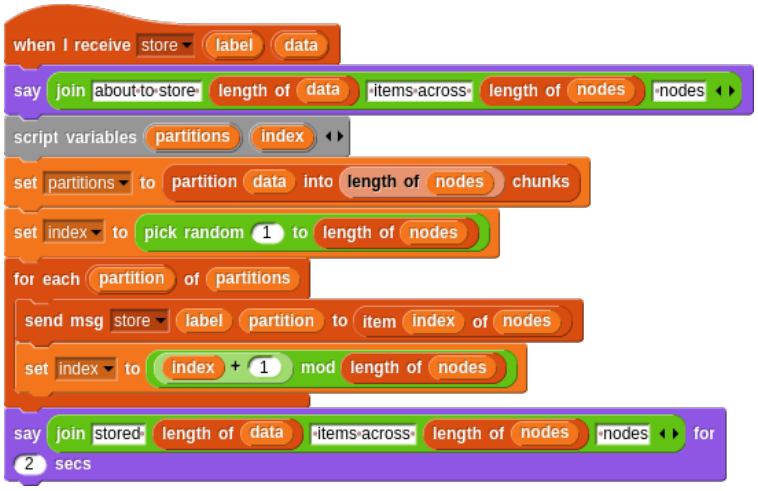


Figure 46: Distributing Data To Worker Nodes

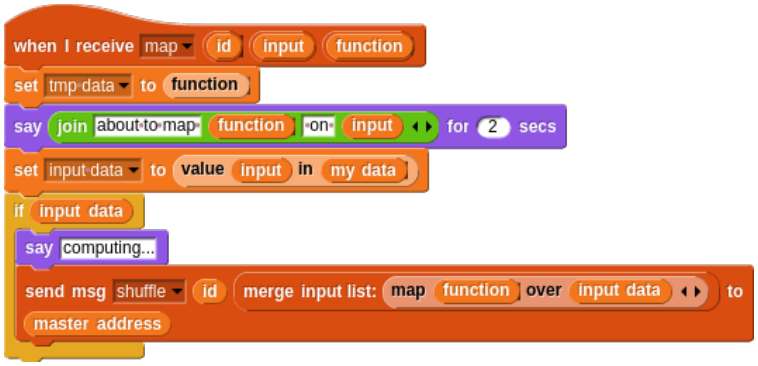


Figure 47: Executing Map Step on Worker Nodes

the given data, it will invoke the provided map function over the list of all the local input data. As each map step returns a list of key-value pairs, these individual lists are merged into a single list of all the resulting key-value pairs. This list is then sent to the master node to redistribute (shuffle) the data over the cluster. The master node will then shuffle all the data across the worker nodes and then, once all the map steps have completed, execute the reduce step of the given job.

Although this example is certainly far from a production ready implementation of MapReduce, it demonstrates the flexibility and capability of the provided abstractions for teaching advanced distributed programming concepts within an educational visual programming environment. Providing easy access to advanced networking applications in a familiar blocks-based environment facilitates incrementally introducing novice programmers to concepts of distributed computing including distributed storage, execution, coordination and parallel computation.

Chapter III

Remote Block Execution

Data locality and network latency are important considerations when developing distributed applications. Executing block functions on remote computing resources enables users to gain concrete experience with the implications of data locality and network latency. As these computing resources may not be blocks-based programming environments, this presents a number of different research challenges. One such challenge is cross-compiling code from a flexible, lively block environment to a textual programming language. There are also semantic challenges about the appropriate block behavior in non-native environments, conforming to the concurrency model of the original environment and security considerations when compiling blocks to a textual language for execution in a new environment.

3.1 Background and Related Work

Execution of blocks in other environments is closely related to the compilation of blocks to text code as well as combined textual and visual environments. Compilation of blocks to textual code is one approach to supporting the fundamental mechanics of executing blocks in non-native environments. However, converting between blocks and textual code often presents a number of unique challenges.

3.1.1 Converting Blocks to Text

There have been a few different projects providing support for compiling blocks into text code. Two noteworthy examples can be found in Snap4Arduino and Snap*!*. Additionally, Blockly [43] provides a blocks-based interface for manipulating underlying textual programs. Unlike Scratch, Snap*!*, and GP [77], Blockly does not provide a virtual machine for executing underlying block primitives. Instead, Blockly provides a visual interface which then generates textual code. As Blockly does not include a virtual machine for the visual language, textual code is generated from the visual representation and then must be executed in an environment supporting the given textual language.

Snap*!* [87] provides support for generating code from block scripts using a feature called "codification." This features exposes blocks for defining templates for mapping blocks to text. Each block can consist of two types of code mappings, "code" and "header" mappings. The "code" mapping is the mapping for the individual block; the "header" mapping defines a dependency (such as a function definition) for the block which needs to be added before

the given block's "code" template. After defining these mappings, the user can use a block for converting a single set of scripts to code given these templates.

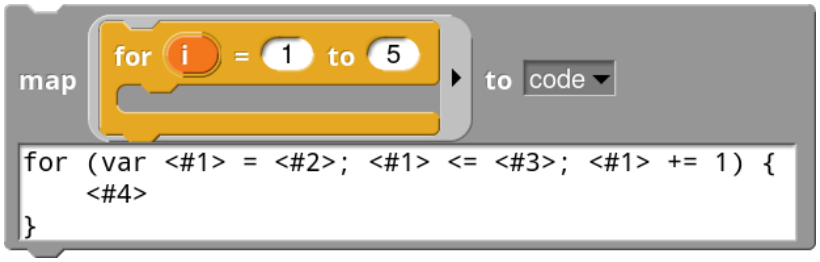


Figure 48: Defining a Code Mapping from Snap! to C

An example code mapping is provided in Figure 48. In this example, the user is defining the code mapping for the given `for` loop. The template defines the overall structure of the generated code and uses "<#N>" as a placeholder for the inputs where "N" is the index of the block's input. In the given example, "<#1>" corresponds to the `i` variable, "<#2>" corresponds to `1`, "<#3>" corresponds to the max of the loop (5), and "<#4>" corresponds to the body of the for loop (currently empty). Unlike many other programming languages, `for` is not a primitive construct in Snap*!* but is a custom block. Despite being a custom block, this particular code mapping does not use the block definition at all in the generated code.

After defining the code mapping, users can use these mappings to convert individual scripts using the `code of` block. The `code of` block uses the defined code mappings to generated textual representations of the given block script. An example is shown in Figure 49 demonstrating the use of the `code of` block to generate code for a given script. In this example, we are generating C code for a `for` loop which prints the iteration number on each iteration.

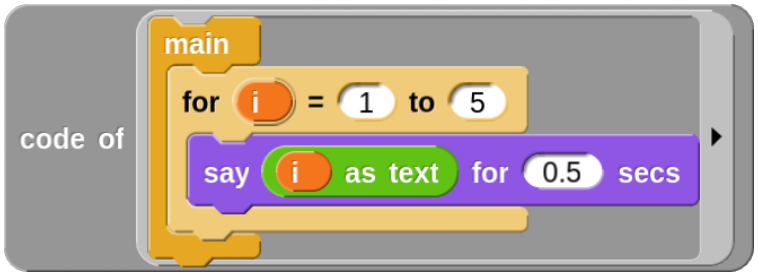


Figure 49: Generating Code From Blocks in Snap*!*

This approach to code generation certainly has its strengths and limitations. As the user can provide arbitrary code templates for any block (including custom blocks), the generated code can be quite simple and legible. Generating simple code should make the textual

```c
#include <stdio.h>
int main()
{
    int i; for (i = 1; i <= 5; i++)
    {
        printf("%d", i);
        printf("\n");
    }
    return(0);
}
```

Figure 50: Generated Code From Figure 49

representation appear more accessible as opposed to generation techniques which generate more complex textual code. Furthermore, as the user can define arbitrary code mappings, the target language can be very easily changed to provide users examples of how the given script may be written in a variety of different languages.

There are also a number of limitations with this approach. Users can only generate code for a single script and not an entire sprite or project. This allows the code generation to be much simpler as the generated code does not need to make any considerations with respect to concurrency. Without code generation support for multiple scripts, this approach cannot be effectively used for executing arbitrary scripts. Consequently, the execution of the generated code will likely behave differently than the original block functions (examples of such scripts are provided in Section 3.2).

Another limitation of this approach is the requirement of the user to define the custom mappings for each of the blocks that will be compiled. This significantly limits the use of this feature by novice users as custom blocks require defining a new code mapping for the given block. Requiring the user to define a new code mapping for every new custom block definition requires the user to already understand how to write the analogous behavior in the target programming language. Although this is often the case for instructors, this cannot be expected of the actual novice users. Therefore, this feature can be useful for creating examples and demos but not for actual use by novices without restricting the supported blocks they can use in their projects.

Snap4Arduino extends Snap*!* to support developing code to run on the Arduino [4]. Unlike the codification feature of Snap*!*, Snap4Arduino generates code for the entire user project without requiring the user to define templates for mapping the block to text. As Snap4Arduino is targeting the Arduino platform, it also generates the appropriate initialization and helper functions as well as ensures that the program contains the main loop.

The Snap4Arduino environment does not support all the features of Snap*!*. These include

asynchronous broadcasting, custom blocks, multi-dimensional lists and lambda expressions. The stage, sprites, and sounds are also unsupported elements of the project when executing on Arduino.

The code generation also does not conform to the semantics of the concurrency model and will exhibit slightly different behavior during execution than Snap*!*. One example can be found in the approach for broadcasting events. In Snap*!*, the `broadcast` block will run a script concurrently. Specifically, the scripts responding to the event will be queued and executed after the current script completes. However, when running the same block on the Arduino, the `broadcast` blocks are converted into function invocations which are then evaluated synchronously. Although this is a subtle distinction, changing the execution semantics when running the code on Arduino can result in unexpected behavior and bugs that can be very difficult to troubleshoot.

3.1.2 Combined Visual and Textual Environments

After learning to program in a visual programming environment designed for educational purposes, users often want to "graduate" to a textual programming language such as Python or Java. There have been a number of studies investigating this transition as well as simply providing both options to users [137, 65, 19, 50] or providing textual language syntax on blocks [39, 70]. There are a number of visual programming environments designed to address this transition [6, 41, 57, 131, 10, 9, 88]. These environments facilitate the transition from block programming to textual programming by enabling users to view the generated code from the given blocks. An example of this is provided in Figure 51.

In this example, the program will create a turtle named "Tina" and draw ten filled circles of random colors at random locations on the screen. This program is available in the block environment on the left and also shown as code in the right pane. Selecting a block will also highlight the corresponding code that has been generated for it on the right. In Figure 51, the block which sets the turtle's speed is currently selected and the corresponding python code ("turtle.speed(10)") is highlighted on the right. This makes the transition easier by allowing users to view their projects in python as well as easily determine the relationship between the individual blocks in their program and the individual lines of code in the generated python program.

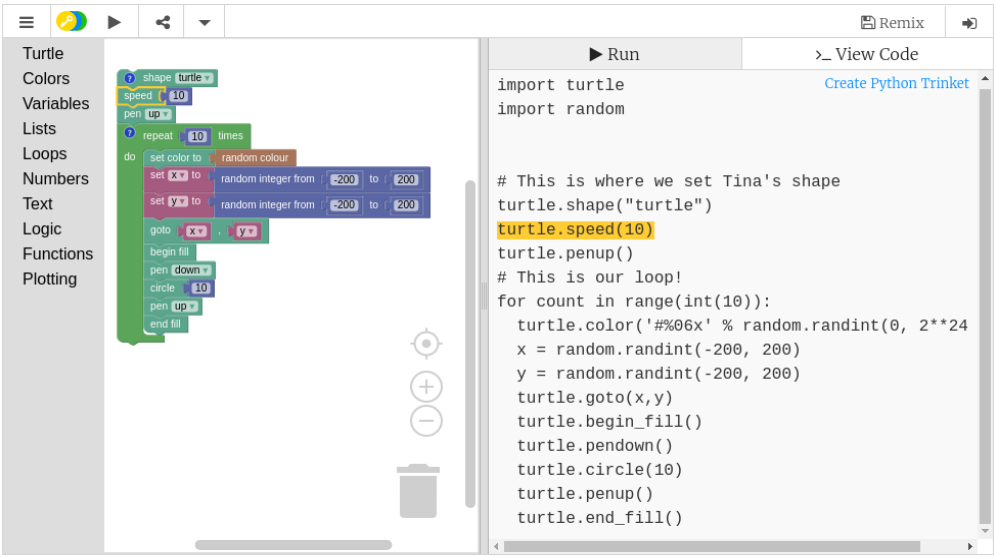


Figure 51: Editing Blocks and Text Together in Trinket

3.2 Compiler Design

3.2.1 Configurable Block Behavior

One of the most important design considerations is supporting block execution in a new environment which may have different capabilities or may require different implementations to achieve analogous behavior. One simple example is the movement blocks. In a server environment, these blocks may not have any associated behavior as the environment has no graphical output and, thus, sprites cannot move. Consequently, the movement blocks should effectively be ignored when a program is evaluated in this environment. In a robotics environment, the movement blocks could still have defined behavior as they may be used to move some given robot. Although this behavior is defined and natural, it is drastically different from the initial implementation and would require replacing the underlying implementations of the supported movement blocks.

To promote portability, we have decoupled the environment from the generated code allowing it to be provided at runtime. The generated code treats the implementation for the primitive blocks analogously to system calls in the underlying execution environment and similar to the design of the Java Native Interface [73]. An overview of this architecture is shown in Figure 52.

Although these techniques can be applied across target languages, the compiler currently outputs JavaScript code. Its ubiquity enables the generated code to be run across many

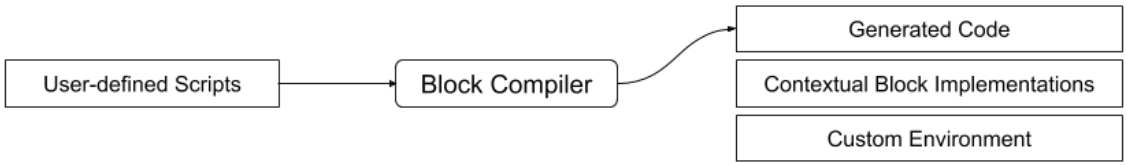


Figure 52: Environment-Independent Block Compiler Design

platforms and further promotes portability. As many blocks-based programming environments are web-based and run in the browser, the prominence of JavaScript in the browser also makes it an appropriate target language.

Compiling user-provided blocks generates a JavaScript function which accepts a single argument, `__ENV`, as its input. The `__ENV` input is the contextual block implementations for all the primitive block types supported by the originating block programming environment. When blocks are invoked in the generated JavaScript, these blocks are simply invoking the associated primitive block implementation from this `__ENV` input.

Figure 53: Simple Example Script

```
function(__ENV) {
    // Initialization code instantiating the Sprite
    //   (assigned to the "sprite" variable)
    __ENV.forward.call(
        sprite,
        __ENV.reportSum.call(
            sprite,
            __ENV.variable.call(sprite, "speed"),
            5
        )
    );
}
```

Figure 54: Generated Code for Figure 53

A simplified[1] example of the generated code for the script in Figure 53 is provided in Figure 54. In this example, the user is using three blocks which correspond to the `forward`, `reportSum`, and `variable` functions in the generated code. In the generated code, each block invocation retrieves the corresponding function from the contextual block implementations

[1]This example has been simplified to demonstrate the decoupling of the user code and the underlying block implementations and how this affects the generated code. Additional complexities resulting from addressing later concerns, such as concurrency, have been removed from the example for clarity.

for the given environment, `__ENV`. The function is then invoked with a reference to the corresponding sprite used as the caller of the function (denoted by `sprite` in this example).

3.2.2 Concurrency Model

Blocks-based programming environments have been shown to enable novices to develop highly concurrent applications naturally even without directed instruction [76, 84]. This is largely due to the underlying concurrency model used by many blocks based environments like Snap*!* and Scratch. However, this custom concurrency model adds an additional constraint as the generated code must adhere to it, too.

These concurrent blocks-based programming environments provide a simple, intuitive concurrency model. All threads (or scripts) share a single process and use passive scheduling. Yielding control of the process is determined by the use of a set of specific blocks. Generally, blocks which execute for a specified duration or perform repetitive tasks will result in yielding control of the process. Loops relinquish control at the end of each iteration. Blocks which include a duration, such as `wait` and `doSayFor`[2] blocks, will yield control until the duration has been completed. Performing network requests or other asynchronous requests also yield until the operation has been completed. Additionally, some languages have provided custom blocks, such as the `warp` or `all at once` blocks [2], to enable the users to prevent yielding after each iteration in loops and speed up complex computations or graphic effects.

One important consideration in trans-compiling the user's blocks into JavaScript was maintaining this model of concurrency. The browser environment, use of passive scheduling, and the ubiquity of JavaScript made it a powerful candidate for the target language. Generating code which can be executed in the browser, although not necessarily a formal requirement for executing blocks on the server, provides flexibility in future work such as optimizing local block functions when applying blocks-based programming to more computationally intensive domains. Also, the use of passive scheduling in JavaScript (the event loop) and dynamic typing reduces the amount of necessary overhead in the compiled code and simplifies the generated output code.

To conform to the same concurrency model, the generated JavaScript uses the underlying event loop (and promises) to allow scripts to yield control to the next script. Specifically, threads yield in the generated JavaScript by creating functions to resume the thread and placing them on the JavaScript event queue. As loops yield control after every iteration, they are converted into asynchronous, recursive functions. Blocks which prevent yielding, such as the `warp` block, are compiled into two invocations: one invocation upon entering

[2]The `doSayFor` block displays a dialog box with a provided message by the sprite and then waits until the given duration has been exceeded.

the block and another invocation on exiting the given block. When invoked, the compiled function corresponding to this block will increment and decrement a counter (local to the current script) on block entry and exit, respectively.

An additional challenge lies in the use of yielding blocks which return a value. More advanced blocks-based programming languages, such as Snap*!* and GP, support lambda expressions. As lambda expressions may contain loops and be the input to another block, all inputs may need to be resolved asynchronously before executing the given function.

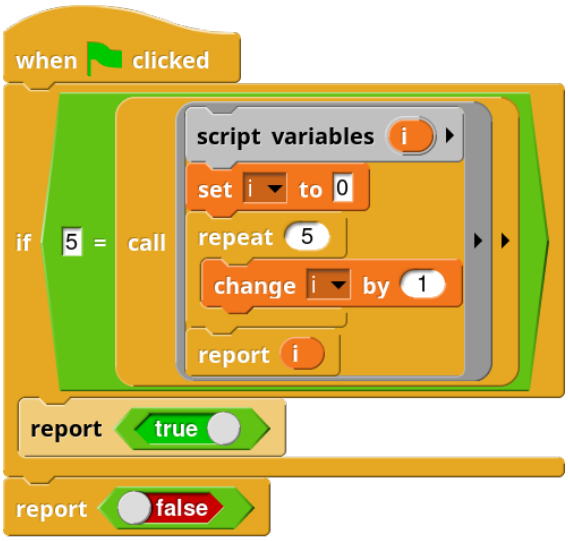


Figure 55: Lambda Expressions

Figure 55 shows an example of a more complex example of a short block script using an anonymous function. In this example, the anonymous function will initialize a counter to 0 then increment it 5 times in a loop and return the value. This returned value is then compared to 5 in the conditional of an `if` statement. If they are equal (which they should be), the script will return "true." As the anonymous function contains a loop, it will yield control of the process during the evaluation of the conditional.

The generated JavaScript code will need to support asynchronous inputs to arbitrary blocks and structures such as `if` statements. To support the evaluation of arbitrary asynchronous function inputs, the generated code provides its own function, `callMaybeAsync`, for calling the underlying functions given the desired caller and inputs. This function first resolves all the inputs, as needed, and then invokes the function as the caller. Functionally, this behaves very similarly to promises in JavaScript as promises allow the user to use the results from asynchronous functions and synchronous functions uniformly. However, it is worth noting that the standard promise behavior is to always resolve their inputs (which may or may not be asynchronous) asynchronously whereas in our case, the generated code needs

to resolve the inputs asynchronously if and only if at least one of the inputs is asynchronous. Otherwise, the inputs should be evaluated synchronously to ensure that the thread does not yield control of the process.

```
function() {
    // SPromise is a promise which evaluates synchronously if possible
    return new SPromise((resolveFn, rejectFn) => {
        callMaybeAsync(
            sprite,
            __ENV.doDeclareVariables,
            'i'
        ).then(() => callMaybeAsync(
            sprite,
            __ENV.doSetVar,
            'i',
            '0'
        ).then(() => new SPromise((resolveLoop, rejectLoop) => {
            function doLoop_item_14 (item_14) {
                return callMaybeAsync(
                    sprite,
                    __ENV.doIfElse,
                    item_14-- > 0,
                    () => {
                        callMaybeAsync(
                            sprite,
                            __ENV.doChangeVar,
                            'i',
                            '1'
                        ).then(() => callMaybeAsync(
                            sprite,
                            __ENV.doYield,
                            doLoop_item_14,
                            item_14
                        )).catch(rejectLoop)
                    },
                    () => {
                        callMaybeAsync(
                            sprite,
                            resolveFn,
                            callMaybeAsync(
                                sprite,
                                __ENV.variable,
                                'i'
                            )
                        )
                        .then(() => resolveLoop())
                        .then(rejectLoop);
                    }
                ).catch(rejectLoop);
            }
            callMaybeAsync(
                sprite,
                doLoop_item_14,
                '5'
            )
        })))
        .catch(rejectFn)
    });
}
```

Figure 56: Generated Code for Lambda Expression in Figure 55

```
function(__ENV) {
    // Initialization code instantiating the Sprite
    //   (assigned to the "sprite" variable)
    callMaybeAsync(
        sprite,
        __ENV.doIf,
        callMaybeAsync(
            sprite,
            __ENV.reportEquals,
            '5',
            callMaybeAsync(
                sprite,
                __ENV.evaluate,
                // fn for the anonymous function
            )
        ),
        function() {
            callMaybeAsync(
                sprite,
                __ENV.doReport,
                true
            )
        }
    ).then(() => callMaybeAsync(sprite, __ENV.doReport, false))
}
```

Figure 57: Generated Code for Figure 55

A simplified example of the code generated from the lambda expression in Figure 55 is provided in Figure 56. The generated code consists of a single function which returns a promise (specifically, a promise which resolves synchronously when possible). When compiling the `doReport` block inside of a lambda expression, it is compiled to the resolve function for the promise corresponding to the parent lambda expression. In Figure 56, this can be seen on line 34 as the `doReport` block has been compiled to the function `resolveFn`. This is an exception to the standard approach of compiling blocks to invoke functions from the provided primitive block implementations. As the `doReport` block is not compiled to invoke the associated function from the environment, it is unable to be overridden when used in lambda expressions.

Figure 57 shows an example of the code generated for the remainder of the code given in Figure 55. Overall, this example behaves as expected; the `callMaybeAsync` function is used to invoke all functions to allow it to wait to resolve any inputs which are asynchronous and yielding the control of the process. On line 11, the `__ENV.evaluate` function is used to invoke the generated function from Figure 56 (omitted on line 13 for brevity).

3.2.3 Closures and Function Portability

Executing blocks on the server requires the compilation of individual lambda expressions. As these expressions may reference variables in an outer scope, it is important that the compiler supports the compilation of closures and not solely lambda expressions. Additionally, as closures may also broadcast messages invoking other scripts – which may mutate global variables – the block functions may have more dependencies than simply referencing variables in a closure.

A simple example of a lambda expression is provided in Figure 58. This lambda expression takes a single input, `#1`, and returns the sum of the input argument and 5. It is worth noting that this is the simplest type of function that we might want to compile; it has no references to any variables in an outer scope and has no dependencies on the behavior of the enclosing environment.

Figure 58: Basic Lambda Expression

Closures provide some additional complexity as the function now has references to variables in an outer scope. An example closure is shown in Figure 59. This example is a natural extension of the example provided in Figure 59. Rather than adding simply adding five to the input, the closure adds the value of variable `x` (defined in an outer scope) to the input argument.

Figure 59: Basic Closure

Functions defined in blocks-based programming environments can also have more dependencies on the original environment. This includes closures that trigger other scripts mutating a shared variable. An example of this can be found in Figure 60. In this example, the function (on the right) is the target function to compile and execute in a different environment. However, the original environment contains an event handler (shown on the left) which modifies a variable from an outer scope named `enclosed var`. In the user's function, the variable is first set to zero, then the "`change var!`" event is broadcasted and the script waits until `enclosed var` has been changed to a non-zero value. Finally, the script reports the value to which `enclosed var` has been changed. In the context of the function's environment, the broadcast function will result in the execution of the script on the left.

This script will set `enclosed var` to fifteen resulting in the main function completing and returning fifteen.

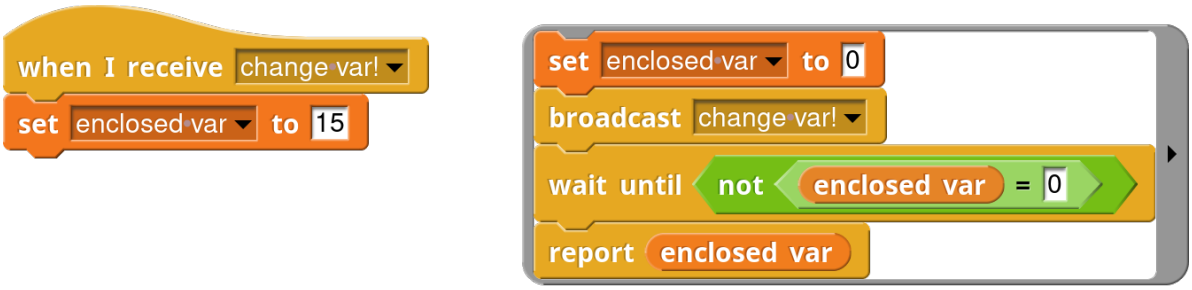


Figure 60: Closure with Other Script Dependencies

Functions may also include custom blocks and depend on the block definitions from the original environment. This results in the user's functions potentially having a number of different dependencies from various aspects of the originating project including referenced variables, custom block definitions and even other scripts. One primary expectation when compiling blocks to execute in alternative environments is the analogous behavior when executing the blocks. It is important that we provide a flexible compilation of the project which supports compiling portable functions and closures with the aforementioned dependencies.

We guarantee that the dependencies for the block are included and available in the JavaScript output by generating code for all input elements to the compiler. When serializing a block function, we also serialize the dependencies of the given blocks. This ensures that we will generate code for all dependencies included in the serialized block function. However, if the block has no external dependencies, it may not include any additional elements in its serialized form. To support this form of serialization, the compiler maintains an internal state representation of the input and updates this representation as it parses the serialized block function. After parsing all the provided input, the compiler then generates the corresponding JavaScript code for its accumulated internal state.

As discussed in Section 3.2, compiling a serialized project generates a function which accepts the block implementations for the given environment at run time, `__ENV` from Section 3.2.1. After generating the code for all the dependencies for the block function (which may include the entire project), the code for the provided block function can be generated. As the compiler outputs a JavaScript function, the generated code for the input blocks can be added as a return value of this function. Generating the JavaScript code for an entire project enables the generated function to reference any necessary dependencies in the generated JavaScript closure. The resulting code represents a higher-order function which generates the desired user provided closure.

Generating a higher-order function to construct the user provided closure or block function provides additional interesting characteristics such as supporting the creation of generators. An example of a generator for Fibonacci numbers is given in Figure 61. In this example, the provided function references two variables from an outer scope: `prev` and `current`. These variables are both initialized to a value of 1 and are updated on each invocation of the given function. As described above, compiling this function outputs a JavaScript function which creates this given generator. The resulting higher-order function can then be used to instantiate generators which each will output numbers from their corresponding locations in the Fibonacci sequence.

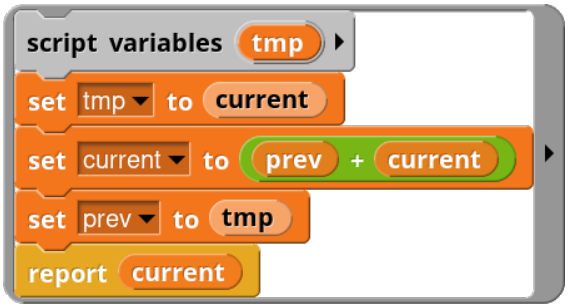


Figure 61: Fibonacci Generator

3.2.4 Security Concerns

Security is an important consideration for any shared, publicly accessible resource. As we are allowing users to provide custom block functions to be executed, there is a larger attack surface which must be carefully secured. In this section, we will discuss potential attack vectors and how they can be mitigated.

One obvious attack vector is the `reportJSFunction` block provided by Snap*!*. The `reportJSFunction` block allows the user to embed arbitrary, custom JavaScript code in a block and execute it like any other block in the user's program. This can be problematic and provides a compelling reason not to open questionable applications created in Snap! (or any extension). Executing arbitrary blocks on the server exacerbates this security risk created by allowing users to embed arbitrary JavaScript in blocks.

The decoupling of the environment implementations of primitive blocks provides an easy and effective solution to the first of these security concerns. Although the primary focus has previously been on supporting the capabilities of the underlying execution environment, the contextual block implementations can be used to carefully restrict capabilities and functionality. Despite being supported by the underlying environment, some capabilities pose a security risk and should be disabled when executing on a shared resource such as the

`reportJSFunction` block. This vulnerability can be mitigated by defining a contextual block implementation which simply throws an exception on execution. This will ensure that any user code will compile but will fail if the user tries to execute any unsafe blocks.

Another similar attack vector can be found in the user's ability to provide arbitrary text input in a number of different locations in the project. As block-based programming languages do not need to parse textual concrete syntax to construct the abstract syntax, they can be more flexible with respect to the valid characters for text input (including variable names) and can include spaces, quotes, and escape characters. This flexibility increases the risk of code injection attacks and can occur in sprite names, stage names, variable values, block input fields, and variable names. Although this security vector has unique characteristics in a blocks-based environment due to fewer restrictions on user input, it can be addressed by carefully sanitizing user inputs similar to conventional approaches for preventing cross-site scripting attacks.

Finally, the execution of blocks on the server also introduces risk of denial of service attacks (both intentional and unintentional) as users can easily create infinite loops or error-prone code which consumes excessive resources. As the intended users are novice programmers, inefficient or error-prone code is not only a possibility but an expectation. To address this issue, we have extended the scope of the configurable functionality of the execution environment of our compiled JavaScript code. Specifically, we provide capabilities for providing a custom implementation for `callMaybeAsync`, the function used for resolving inputs and evaluating functions with the given inputs. Using a custom `callMaybeAsync` function, we can provide our own preconditions before evaluating each function. This allows us to easily add a timeout precondition in which the script terminates after the execution duration has exceeded the given threshold.

3.3 Execution RPC

3.3.1 Execution Semantics

Remote block execution is supported using one of the existing networking abstractions, Remote Procedure Calls. This provides users the capabilities of executing the blocks on a remote machine while making it clear that the input function is being executed remotely. Transparency about the location of the execution allows the students to experience the impact of network latency when accessing network resources and, more importantly, how this latency can be mitigated using techniques such as batching.

An example of a block function executed on the server is provided in Figure 62. In this example, the provided function is converting the `movie ids` variable (referenced in an outer

scope) into the corresponding movie title using the `map` block. After executing the `map` block, the result is then returned by the given closure and returned to the user as the result of the `call` RPC. For large lists of movie ids, this approach can easily demonstrate the benefits of batching network requests as opposed to simply calling them in serial.

Figure 62: Batching RPC Requests

The remote execution of the RPC is a concept that is exposed to the user and this is reflected in the semantics of the execution RPC. When referencing dependencies from an outer scope, such as variables or other scripts, the function is evaluated with a copy of the given dependencies. Changes to local copies of variables are not reflected in the execution of functions that are executing on the server. Similarly, changes to other dependencies, such as custom block definitions or dependent scripts, are also not reflected in the remote execution of functions.

Modifications to remote copies of variables and project state behave similarly. Although the executed function may have modified dependencies from an outer scope, such as changing the values of variables in the outer scope, these changes are not synchronized with edits on the client. After executing the blocks on the server, the user is only provided with the output of the function. Any modifications to any of the copied dependencies of the block function are discarded.

Figure 63: Setting Variable During Remote Block Execution

A clarifying example is provided in Figure 63. Here, the function is simply setting an enclosed variable and then returning that the given variable has been updated. When this function is evaluated on the server, a copy of the variable, `enclosed var`, is serialized and submitted with the function to the server. The function is then compiled and executed on the server and the result is returned to the user. As the value of `enclosed var` is never used in the function, the update to `enclosed var` has no impact on the execution of the function nor the user's environment.

3.3.2 Modified Block Implementations

When executing blocks remotely, a number of the blocks require modifications to ensure the desired behavior during execution. This includes both removing unsupported functionality and updating block implementations.

One obvious example of block implementation modification are blocks pertaining to the visual appearance of sprites and the stage. This includes blocks modifying the graphic effects, using the "pen" and changing the costume. These and blocks relating to audio or user input also perform no operation when executed remotely. Surprisingly, the motion blocks need to be functional on the server as these blocks could be used to provide an alternative way to perform various simple mathematical operations. For example, the motion blocks could be used to perform conversions between polar and Cartesian coordinate systems as shown in Figure 64. Although this would be an unorthodox use of the remote block execution functionality, the function is certainly well-defined and should be supported.

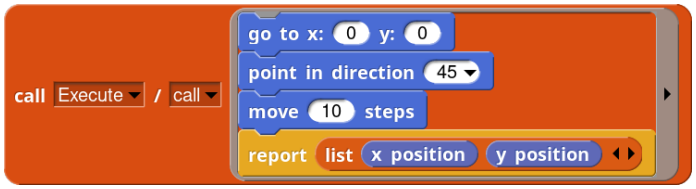


Figure 64: Converting Point from Polar to Cartesian

As the blocks are no longer executing in a Role, the networking blocks also require modifications to their underlying implementations. However, as the blocks are not executing on the actual client, some block behavior must be precisely defined and could be ambiguous. One such ambiguity lies in the address of the executing code when performing network operations. Both RPCs and message passing can involve sending a response or result to the originating client. When executing blocks on the server (on the behalf of the given user), it is not immediately clear if the executing function should simply use the same address as the originating client or if it should have its own unique address.

One simple case is the use of the `callRPC` block which invokes a remote function and returns the result. These blocks should certainly recognize the executing function as the recipient and the result of the RPC request should be the returned by the given `callRPC` block. This is also consistent with the expected usage as this supports the batching of RPC invocations and returning a list of the results for each individual invocation.

A more complex case arises when considering invoking RPCs which send messages to the sender as with the "Earthquakes" Service. As the results of these RPCs are returned indirectly as a stream of messages rather than as a simple response, the resultant data should

be sent to the user when batching these RPC requests. Although sending the resulting streams of data back to the function could still be captured by dependent functions and returned to the user, this approach would be significantly more complex and likely too difficult for even more advanced programmers. Sending the resulting streams of messages back to the user suggests that the functions executed remotely should be evaluated in a context which shares the same address as the originating client. That is, the blocks should be executed as if the originating client executed them.

Chapter IV

Collaborative Editing

4.1 Background and Related Work

Collaboration is important both in education and software development. There have been many efforts for improving collaboration in these areas. These include platforms such as Github, Bitbucket, and Gitlab as well as management tools such as SVN, git, and mercurial. Examples also include tools that support concurrent, online collaborative editing such as Google Docs, Cloud9, Drawp for School, and Screenhero [44, 16, 45, 46, 26, 35, 117]. However, providing concurrent, online collaborative editing can be quite challenging and can require fundamental decisions regarding the synchronized content, supported operations, and the strength of the consistency model.

One ubiquitous online collaborative tool is Google Docs. Google Docs is a word processing tool enabling users to collaborate by synchronizing the state of the content, i.e. the document, without sharing the actual state of the editors of each user. This allows users to work together on different parts of a paper without synchronizing the users' screens (like screensharing software). User efficiency is promoted by allowing users to easily work simultaneously on different parts of a given document and only see the output of the other user along with the location of the user's cursor.

To enable concurrent editing, Google Docs uses a technique called Operational Transformations [37]. Operational Transformations is an edit-based approach to collaboration in which concurrent edits are modified to perform the same operation when applied after one another. An example of this is provided in Figure 65.

In this example, two users are editing the same text, "abc", simultaneously. One user adds a "z" to the beginning of the text (creating "zabc") while the other user removes the "b" from the text (creating "ac"). That is, the first user performs an "INSERT" edit event with the arguments 0 (the position at which the character should be added) and "z" (the character to add). The second user's action results in the creation of a "DELETE" edit event with the argument 1 (the position of the character). At this point, if both events were simply broadcasted to the other user, the second user's text would behave as expected and would be "zac" but the first user's text would be "zbc" as the "INSERT" event changed the index of the "b" character that the second user intended to remove. Operational transformations overcomes this issue by transforming the concurrent operations based on the other simultaneous edits. In this example, this results in the "DELETE" event being transformed

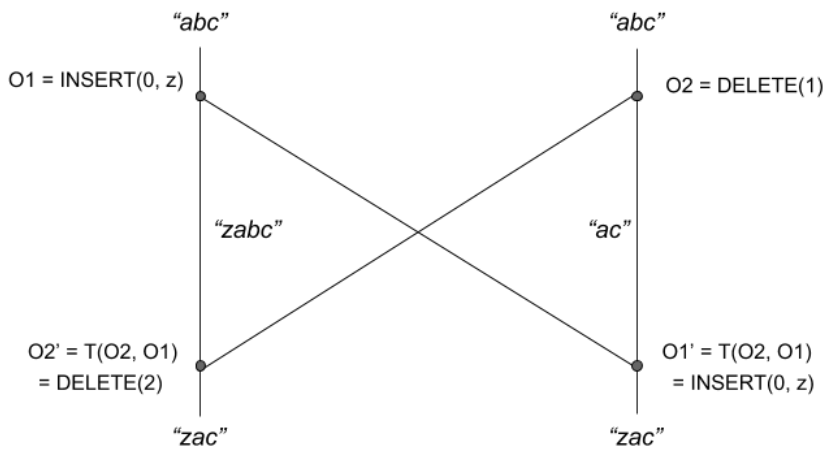


Figure 65: Simplified Example of Operational Transformations

for the first user with respect to the first users' concurrent edit so that the index is now 2 (corresponding to "b" in the first user's text).

Although the idea behind operational transformations is relatively simple, defining a correct transformation function can be rather challenging. There have been a number of different specific algorithms for implementing operational transformations [69]. Unfortunately, these transformation algorithms for operations can get complex making them difficult to prove convergence as well as implement.

Alternative approaches to collaborative editing include Conflict-Free Replicated Data Types (CRDT) and differential synchronization [118, 42]. Conflict-Free Replicated Data Types include both an edit-based technique, Commutative Replicated Data Types (CmRDT), and a state-based approach, Convergent Replicated Data Types (CvRDT). The main idea behind CRDTs is defining a data type without any conflicts for the supported operations. In the edit-based approach, CmRDT, the operations can be applied in any order as they are all commutative. This greatly simplifies concurrency control as the collaborating clients can simply broadcast their edits to all other clients. As CRDTs ensure there will not be any conflicts, a client can simply apply all received edits and be certain that its state will converge to the correct state. CRDTs provide a strong form of eventual consistency [118, 72].

CRDTs have qualities that make them quite appealing as a method of collaboration. These include the ability to prove convergence more simply than in operational transformations as well as the simplicity providing by guaranteeing commutativity in the data type. CRDTs have been created for a number of existing data structures including registers, sets, and graphs [119, 138, 3, 143]. However, there have been some data structures which have been shown not to be able to be represented using a CRDT, specifically structures which

require a globally consistent state as a precondition for an operation being performed. Some structures, such as directed acyclic graphs, will constrain actions that can be performed to ensure that a global property is not violated (such as having no cycles in a graph). As this precondition for an operation depends upon global information and cannot ensure that the global state is consistent (as a concurrent operation may have changed the given global property), it cannot guarantee that the current operation will not result in a structure which violates the given property of the data structure (such as having a cycle in a graph).

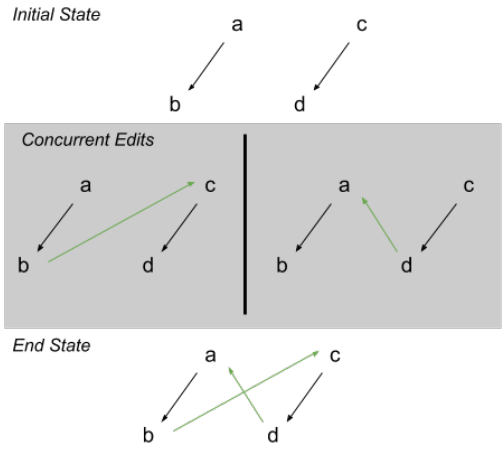


Figure 66: Concurrent Directed Acyclic Graph Edits Require Global State Information

In Figure 66, users are collaboratively editing a directed acyclic graph with edges $\vec{ab}$ and $\vec{cd}$. As the current graph structure is a directed acyclic graph, users cannot add any edges which could create any cycles; that is, users cannot add any edges from a node to one of the node's ancestors. Both users simultaneously add an edge to the graph; the first user adds the edge $\vec{bc}$ and the second user adds edge $\vec{da}$. These actions individually do not create any cycles and are both reasonable actions on this graph. However, applying both actions results in a cycle and the data structure is no longer a directed acyclic graph because each action violates the precondition required for the other user.

The example in Figure 66 demonstrates some limitations of CRDTs with respect to data structures with global state constraints. A precondition with directed acyclic graphs for adding an edge requires that the destination node is not an ancestor of the source. As any other client could be adding an edge which could affect this precondition, the client would need to be able to ensure that no concurrent edits were going to change the state of the data to violate this given precondition. It cannot be ensured that concurrent edits will not change the state to violate this precondition, as such directed acyclic graphs supporting adding edges cannot be implemented as a CRDT [119]. However, this issue is directly related

to the actions supported on the data structure and there have been CRDTs designed for some more restricted forms of these graph structures such as the Monotonic DAG [119].

Differential synchronization takes a significantly different approach than CRDTs to provide simultaneous collaborative editing support and builds upon three-way merging. It is a simple method of collaboration designed to be easily incorporated into existing applications [42]. By requiring the clients to store a "shadow" copy of the project (without the user's edits), the clients can compute a difference (diff) and send the edits as patch operations to the server to merge into the current version. When a client receives changes from other users, the changes are applied to both the client's shadow and text following which the client can then diff his/her code against the updated shadow and send updates to the other client. It is worth noting that the application of the edits to the shadow copies should always be able to result in the current version of the text from the server whereas applying the edits to the client text is only a best-effort, fuzzy patch. An overview of this technique is given in Figure 67.

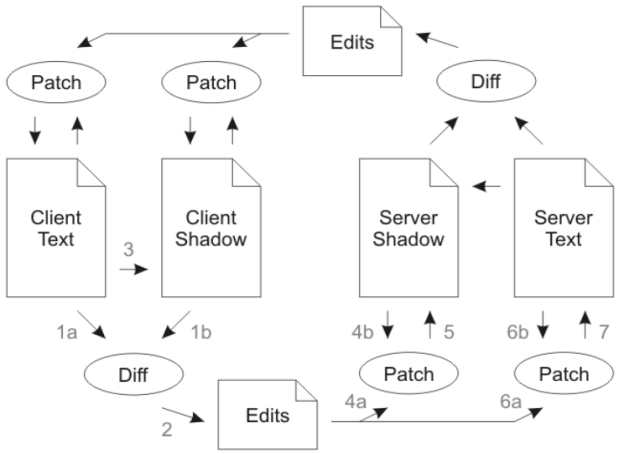


Figure 67: Overview of Differential Synchronization [42]

Collaboration in Visual Programming Environments

As previous versions of Scratch and Snap! were written in Squeak, a dialect of Smalltalk, they were able to utilize Nebraska, a collaboration toolkit for Squeak projects [92]. Nebraska enables a project to share its "world" with others by creating a server which allows clients to connect to it and render the current state of the shared project. Clients can then interact with their view of this shared world. Their events will be sent to the server and these interactions (such as mouse clicks or key presses) will be applied to the shared world on the server. This form of collaboration provided by Nebraska behaves similarly to collaboration

using screensharing. The entire environment is synchronized and no distinction is made between the project content and the individual user editors.

This approach circumvents many of the challenges with collaborative editing. Edits by collaborators are sent as primitive interactions with the world (such as mouse clicks, drags, and keyboards events). Managing collaboration by sending these primitive interactions avoids the complexities of conflicting concurrent edits as the edits are always independent and cannot conflict. When two users try to make conflicting edits simultaneously, both sets of actions will be applied on the project serially. This results in the first action being applied successfully and the second interaction interacting on a different project state. As the project has changed, the second interaction will result in unexpected and unintended behavior because the clicking and typing locations are now interacting with changed parts of the project.

EmfCollab is a tool enabling simultaneous, collaborative editing of Eclipse Modeling Framework (EMF) models [38]. Collaborative editing is supported using a client-server architecture in which a master copy of the model is saved on the server and slave copies are stored on each of the clients. When a user interacts with the local model, emfCollab intercepts the command and first applies it to the master copy of the model and then on the local copy. If the command fails on the master copy, emfCollab cancels the operation locally as the operation conflicts with another simultaneous action. An example of the emfCollab architecture is shown in Figure 68.

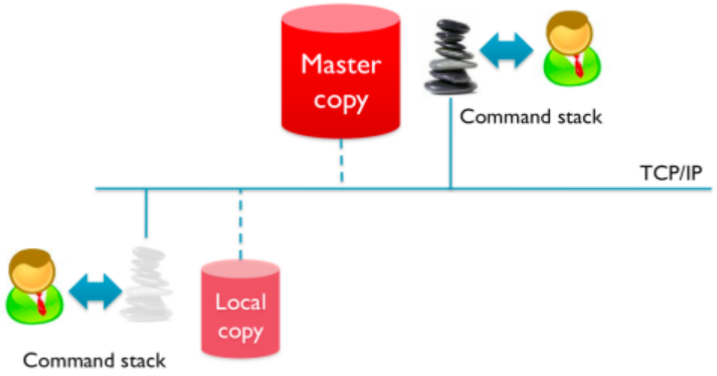


Figure 68: Architecture of emfCollab [38]

EmfCollab is able to bring collaborative editing to EMF models and support multiple users editing the same model simultaneously. As it requires the user command to be accepted by a central server first, it can ensure that the master and slave model copies will be identical. However, this requirement of each command being synchronously accepted by a single server

and applied to a single, master copy creates a bottleneck on collaborative editing, especially if the network has high latency.

AtoMPM is a framework for generating custom visual design environments in the cloud [127]. One interesting thing about AtoMPM lies in its ability to support two different types of collaboration: *Screenshare* and *Modelshare*. Screenshare behaves as expected and shares the content of the user's screen with the given collaborators (as is done in Nebraska). Alternatively, modelshare provides a collaboration experience like that of Google Docs where the content of the project is shared but the user's editor and current view of the project is not.

When multiple users are collaborating in AtoMPM, each client maintains a log of operations for the given client. In the case of conflicting edits, the later conflicting edit may be rejected and the given user action will be lost. If the action is rejected, the user will be notified of the failed action. When both users are online, this issue is not as significant because the number of actions affected should be relatively minimal. The feedback should be given to the user shortly following the given edit. In an offline model, the client may have to trace back a series of operations that are no longer valid to the original failing operation [28]. Unlike emfCollab, AtoMPM allows users to continue working on their project and will revert conflicting actions rather than canceling them on the client immediately. This no longer requires the action to have to wait until it is accepted or rejected before being applied locally and allows clients to work offline (even if those actions could potentially be canceled on reconnect).

WebGME, a web-based generic modeling environment, enables users to build custom visual design environments which support simultaneous collaborative editing as well as automatic version control [81]. In this environment, users are able to first design the syntax of their custom visual language using a meta-model. They are able to immediately use this newly created visual language. Users can further customize WebGME by developing custom methods of visualization, code generation, and many other components to create a powerful, domain-specific design environment.

WebGME provides another approach to collaboration in the context of visual programming environments. Projects in WebGME are automatically version controlled and support collaboration where only the content of the user's project is synchronized (as in Google Docs). Automatic version control provides a natural way to resolve concurrent edits (consistent with the approach used by *git*). Concurrent edits result in one of the commits being placed on its own branch. If these edits are not conflicting, they can be automatically merged to the same branch. If they are conflicting edits, they are not automatically merged and the two users will now be working on different branches which can be merged manually (at which point

the user can resolve the conflicts). As WebGME supports multiple custom visual editors for nodes in the model, collaboration in this environment can have the unique property of occurring between different editors operating on the same data. Collaborating users can be working on a project in which the content may actually appear differently within their respective editors (although the underlying project data is the same).

SLIM provides an alternative, simple approach to collaborative modeling [129, 5]. Unlike the previous approaches, SLIM avoids resolving conflicts by ensuring that they cannot happen by locking parts of the model that are already being edited by another user. This simple technique guarantees that there will not be any conflicting actions. It cannot be used when collaborating with offline models (as in [38]). Unlike the other approaches, this enables users to collaborate on separate parts of large models but ensures no conflicts by prohibiting collaboration on the same parts of models. This approach certainly will be effective when the models are large and there are fewer users (and the users are working on disjoint parts of the model). This will not allow users to collaborate on the same parts of any models as allowed by the other approaches to collaboration in visual programming environments.

4.2 Challenges

4.2.1 Conceptual Challenges

Before addressing the technical challenges of collaborative editing, it is important to determine an appropriate, consistent, and feasible conceptual model for collaborative editing. Specifically, it is important to clearly define the expected behavior including exactly which elements should be synchronized during collaboration. One possibility is to synchronize the entire editor as in Nebraska [92]. This would result in the users sharing not only the project content but also the editor state including the currently selected block category and sprite.

Alternatively, it may be more appropriate to simply synchronize the content of the current project like in Google Docs, WebGME, emfCollab, and SLIM [46, 81, 38, 129]. This would allow the users to work on different parts of a project simultaneously by only synchronizing the underlying project data and not the actual states of their individual editors. Synchronizing only the project content should promote the user productivity as they can edit different parts of a project simultaneously, unlike when sharing the entire editor state. However, a shared editor state may be more useful pedagogically as users could easily demonstrate the entire process of performing various tasks.

Lively programming environments, such as Scratch and Snap*!* [80, 76], introduce additional conceptual challenges when considering collaborative editing. There are no execution and editing modes; blocks are always responding to events and can always execute. Users

can easily discover the behavior of blocks; clicking on a block will execute it. Running scripts can also be edited and modified during execution. Although the liveness of this environment is unconventional, it provides immediate feedback for novice programmers and promotes the potential for exploratory learning.

Although a lively programming environment can promote learning, it introduces complexities when synchronizing the project content. As it is always running, there is not a clear distinction between the project content and output as is present in many other environments. Specifically, the program output is constantly evolving due to both program execution and user interaction. Consequently, synchronizing only the project code will not guarantee that both users will have the same program behavior or output, which may be somewhat counterintuitive.

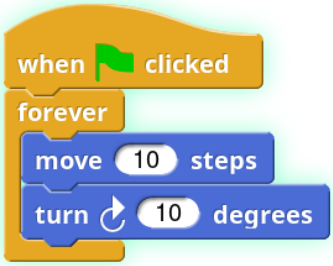


Figure 69: Block Execution in Lively Environment

Figure 69 shows a script which moves a sprite in a circle indefinitely. The faint highlight around the blocks indicates that the script is currently executing. In a lively environment, these blocks are still editable during execution. This allows the user to make modifications such as increasing the input to the `turn` block and immediately he/she will see the effects of the given modification. The user could also modify the execution by dragging the given sprite to another location during execution (although this may be difficult if the sprite is moving quickly). Stopping the execution will stop all executing scripts but there is no concept of an initial state to which the project execution state can be restored. Furthermore, stopping the execution does not prevent scripts from responding to future events or user actions; it simply terminates all currently executing scripts.

Networking support in a lively environment introduces another dimension of complexity with respect to collaboration. When the entire program execution is restricted to a single device, the program execution has virtually no side effects on external applications nor dependencies on other applications. This is not the case when building a distributed application. As the program may be sending and receiving messages from other clients, the execution of a single program can impact the behavior of other executing programs.

As lively environments are always in a state of execution, collaboration results in multiple instances of the given project being executed simultaneously. When collaboratively editing a Role in a distributed application, this can be particularly problematic as it will result in the execution of multiple instances of the same Role. As the distributed application was likely not designed to support multiple instances of the given Role, this will likely lead to unexpected behavior.

Figure 70: Collaboration in a Distributed Application

An example of unexpected side effects when executing multiple instances of the same Role is provided in Figure 70. In this example, the users are developing a Tic-Tac-Toe game with two Roles: "X" and "O." Suppose that each Role alternates in waiting for the given user to make a move and then the Role sends the "move" message to the other Role. When a Role receives the "move" message, it will record the move and update the game board accordingly.

Although this approach will be effective generally, it will not behave as expected if there are duplicates of at least one of the Roles. Currently, "X" is occupied by a user named "brian." Two users, "steve" and "john," are collaboratively working on the second Role, "O." As the environment is lively, this means that there are actually three running Roles in this distributed application (one Role is running for each user). The application will no longer behave as expected. Suppose the "X" Role moves first. "X" will play locally and send the "move" message to the "O" Role. As "john" and "steve" are both occupying the "O"

Role, they will both receive the "move" message and each will individually play in response. However, this will result in two (potentially conflicting) "move" messages being sent back to the "X" Role.

Under these circumstances, the other Roles in the Room no longer behave as expected. Multiple Role instances only exist when users are collaborating. Programming a distributed application to be robust to the side effects of collaboration should not be a design consideration for the novice programmers developing these applications.

4.2.2 Technical Challenges

Visual programming also presents its own technical challenges as it is fundamentally different from textual programming. These environments have a different underlying data structure. Whereas text can be represented as a "growable array" [74], each script in a blocks-based programming environment is a tree. This provides a unique challenge for collaboration as this data structure is more complex and some more recent approaches to collaborative editing have been shown not to be applicable for this type of structure [72].

Supporting collaborative editing in a tree data structure is quite complex as the elements in the data structure are not independent of one another. The data structure is a directed acyclic graph with the global constraint that each node can only have one parent. Additionally, in a block environment, the blocks can only be placed in an input slot of another. This corresponds to an underlying tree data structure where the input blocks are the child nodes of the containing block. Unlike most tree data structures, the nodes in the tree have a fixed number of uniquely named edges to their children as a block's inputs are unique and can only be occupied by a single block at any given time. These uniquely named edges can only be occupied by a single block at a time and introduces yet another constraint on the underlying data structure and further complicating concurrent editing.

4.3 Approach

To address these conceptual challenges to collaboration within lively, blocks-based programming environments, we present a model of collaboration designed to promote productivity within potentially distributed applications. As lively environments are always running, we propose synchronizing only the project content. Specifically, we synchronize the scripts, custom blocks, sprites, costumes, and sounds of the project. This will promote user productivity as collaborating users can edit different parts of the project simultaneously (unlike in models of collaboration like Nebraska). Aspects of the project which can be modified by the execution of the project, such as variable values and sprite positions, are not synchronized.

When executing a program, it is expected that there is only a single user per Role. As it is best to closely emulate the execution circumstances while debugging, it is recommended to only have a single user per Role while testing an application's distributed aspects as well. Regardless, it is important that collaboration in a distributed program results in minimal unexpected side-effects for the rest of the project.

As executing multiple Roles (due to collaboration) in a distributed application can result in unexpected behavior in the rest of the application, only one role is allowed to send messages when collaborating. However, all collaborating roles are able to receive messages and share the same Public Role ID. This can still result in some unexpected behavior. A collaborating Role that cannot send messages will not be able to affect the rest of the distributed application. As a result, the blocked Role will only see the effects of the Role which is allowed to send messages. Although this may not be expected for the user occupying the blocked collaborating Role, this approach minimizes the amount of unexpected behavior and limits it to only the collaborating Roles. As side effects in this scenario cannot be avoided, limiting the side effects of collaboration to only the collaborating Roles allows the consequences to be understandable and explainable.

One driving design decision in developing this model of collaboration was not to restrict the capabilities of the users when collaborating. To support collaboration in lively blocks-based programming environments, we have developed an operation-based model of collaboration composed of 45 different supported operations. These include operations for editing blocks, custom blocks, sounds, costumes, sprites, and the environment such as variables and the stage.

As there are many supported actions, a simple, strongly consistent model of collaboration (similar to emfCollab [38]) is used to manage complexity. When collaborating, all operations are submitted to a central server providing a total ordering of all user actions. The server accepts any operation applied by a client working on an up-to-date version of the project; operations submitted by clients on out-of-sync versions of the project are rejected. This method of collaboration ensures strong consistency as the clients are required to be up-to-date before any project modification will be allowed. Given the complex underlying data structure and many supported operations, the strongly consistent model enables us to ensure that the projects will not be malformed and replicas will never diverge.

Block Operations

Sixteen actions are defined for editing and manipulating blocks. These actions include the expected actions for block programming such as `addBlock`, `setBlockPosition`, `moveBlock`

and `removeBlock`. As first class lists require the use of blocks which can dynamically add or remove inputs, we define two additional actions `addListInput` and `removeListInput`.

Block inputs can be either blocks or primitive values. Blocks can be provided as the input to another block using a `moveBlock` action and block inputs can be replaced using the `replaceBlock` action. Boolean inputs can be modified using `toggleBoolean` actions and color fields are modified by the `setColorField` action. The value of text field inputs can be modified using the `setField` action type and comment blocks can be updated using `setCommentText` action.

There are also actions created for some lesser known features in block programming languages. These include changing a block's type using the `setSelector` action. An example of this functionality is provided in Figure 71 in which an `if` statement is being changed to a `repeat until` loop. Blocks can also be "ringified," or converted into anonymous functions by wrapping the block with a gray ring, using the `ringify` action type. The ring can be removed using the `unringify` action type. It is worth noting that the `unringify` action is different from the `removeBlock` action; removing a block results in the target block being removed with all inputs whereas the `unringify` action will only remove the outermost ring and leave the contained block unmodified.

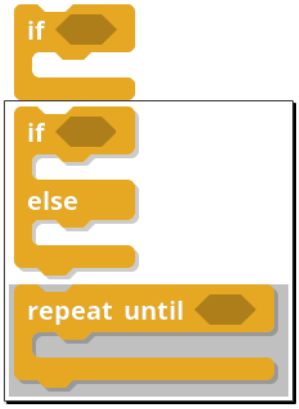


Figure 71: Changing a Block's Type

Custom Block Operations

Custom blocks are much simpler in terms of modification capabilities; this is reflected in the number of action types pertaining to custom blocks. Six types of actions are used for modifying custom blocks. Custom blocks are created using the `addCustomBlock` action and removed using `deleteCustomBlock`. When custom blocks are created, they can be added to a number of different categories including "motion," "control," "variables," "operators," "looks," "sound," and "sensing." The `setCustomBlockType` operation is used to

change the block's category and corresponding type (shape of the block). The custom block method signature (block text and inputs) can be modified using `updateBlockLabel` and `deleteBlockLabel`. There is also an additional action type, `deleteCustomBlocks`, which is a convenience action for performing bulk deletions of custom block definitions.

Sprite Operations

Sprites can be edited and modified using nine different action types. Creation and deletion of a sprite is performed using `addSprite` and `removeSprite` actions. Importing and duplicating sprites is supported through the use of `importSprites` and `duplicateSprite` actions. Sprites have additional properties including name, rotation, and dragging settings. These properties can be modified using `renameSprite`, `toggleDraggable`, and `setRotationStyle` actions. Finally, sprites can also be attached to one another to enforce shared properties such as position and rotation. Sprite attachment and detachment is supported through the `attachParts` and `detachParts` actions.

Resource Operations

Projects can also use audio and visual resources within their application called "Sounds" and "Costumes." Sounds can be modified using `addSound`, `removeSound`, and `renameSound`. Costumes are modified with analogous action types, `addCostume`, `removeCostume`, and `renameCostume`. Additionally, the actual image content can be modified using `updateCostume` action.

Networking Operations

Supporting distributed programming abstractions requires the definition of additional types of operations. Message types are a new concept which is independent of the concepts or operations in existing blocks-based programming environments. Editing message types is supported with two new operations: `addMessageType` and `removeMessageType`. These operations enable users to define and remove message types, respectively. Interacting with the new blocks for messaging and RPCs is enabled using the standard operations for manipulating blocks.

Miscellaneous Operations

There are also a few actions which do not fit into any of the earlier categories. These include actions for editing the stage dimensions, `setStageSize`, and editing variables. Variables are defined using `addVariable` and removed using `deleteVariable`. Specific variable blocks can be renamed using `setBlockSpec`. There is also an operation for opening new

projects, `openProject`. This action is slightly different from the other actions; it is not shared during collaboration and it cannot be rejected.

Chapter V

Novice-Friendly Version Control

Version control is a very powerful tool when developing complex applications promoting collaboration and enabling users to easily move between different versions of the code for a given software application. These types of features, particularly being able to revert the code base to an earlier version of the project, can be especially useful when a project has been broken and the user would like to go back to a working version. Currently these features are provided in advanced tools designed for experienced software developers. Novice programmers are certainly expected to make many mistakes yet the most powerful tools for development and debugging are inaccessible to them until they become more experienced (at which point, they are hopefully making fewer errors).

Although this is an ironic characteristic of these version control tools, it is an understandable one. The concept of version control can be relatively complex and could be challenging to explain to novices. Providing tooling which presents even a subset of these concepts to the user in a meaningful, simplified way is also challenging and could impose additional cognitive load to the user and be detrimental to the learning of the core programming concepts. Despite these challenges, the ability to provide a familiar, intuitive way to be able to interact with the history of a user's project would be beneficial to budding programmers.

5.1 Background and related work

Version control tools are ubiquitous in software development and large scale software projects. They are often quite powerful and can help manage complex projects and collaboration. However, these powerful features often result in complex tools which are rather challenging to learn [22, 56]. Consequently, there have been a number of efforts focusing on bringing explicit instruction about version control into the classroom [111, 85, 106, 121], simplifying version control tools [101, 67] as well as providing version control capabilities into visual programming environments [81, 96, 46].

5.1.1 Version Control in the Classroom

There have been a number of different approaches to teaching version control in the classroom. These include simply requiring the students to use a version control tool, like Git, for working on engineering projects [121] as well as using version control as an assignment submission system [71, 49, 106]. These approaches also targeted techniques for introducing

distributed version control to students as well as techniques for promoting best practices among students when using version control.

In [71], students were taught version control through creating an infrastructure for assignment submission and grading using Git. Using version control for all assignments and grading feedback allowed students to become immersed in the tooling and gain hands-on experience. This approach was used in a number of different classes including a software engineering, compiler and even a CS1 course for non-CS engineering majors.

In class, the Thayer method of moving the work into the classroom was used. That is, the professors actively programmed in front of the class while the students would follow along. The Gutenberg method of teaching by answering questions about the reading was also used. Even with the non-CS engineering majors, the version control submission system was used for all the labs and students were able to gain enough of an understanding of Git to complete even the early labs.

A study by Haaranen and Lehtinen provided a similar approach to teaching Git in the classroom [49]. In this study, git was taught alongside the regular course material and used as a platform to distribute class materials (similar to [71]). Unlike the previous study, this study provided an explicit progression for incrementally introducing concepts in distributed version control.

Harranen and Lehtinen found predominantly positive student feedback with respect to incorporating Git into their course. Although over one-quarter of students reported having no experience with Git before the course, 92% of students described their attitude towards using Git in the course as positive. Students also provided a number of positive comments toward learning Git including "Introduction to git was really useful for me!" and "Also using git for course material reduced file management by a lot and made submitting solutions fast and simple." Negative student feedback commented on the difficulty of learning Git, desire for more practice and use of Git as a course platform.

Singer and Schneider provided one particularly interesting approach to incorporating version control in the classroom [121]. In this study, students formed teams and developed large class projects. Students were provided tools for version control, issue tracking, and a web-based commit newsfeed. The newsfeed also included a leaderboard presenting the commits for each team member and supported student comments on commits presented in the newsfeed. Weekly updates were emailed to the team members containing summary of personal commit activity, the milestones reached during the week and the current leaderboard.

Unlike the previous studies, this study focused on promoting best practices with version control. By incorporating some gamification techniques highlighting the commit counts and activity of other users, the platform rewarded smaller, frequent commits. Additionally, the

incorporation of a leaderboard introduced competitive elements for comparing commit counts between team members. Emailing weekly digests was also designed to provide motivation by showing user progress and promote engagement through the use of emailed reminders.

Students provided a lot of positive feedback pertaining to the introduced gamification techniques. Although the competitive nature of the leaderboard was not always comfortable for students, Singer and Schneider found that it did promote more frequent, smaller commits. The weekly digests also received positive feedback as they provided a quick overview and could help show progress to the student.

The first two studies provide examples of incorporating distributed version control into the classroom to help the students learn by making Git an essential part of class. Both studies found that using a Git-based infrastructure for dissemination of class materials and submission of class assignments was a reasonably effective way to teach version control with college aged students. Furthermore, students also provided feedback to the lesson materials, a pleasant side-effect to using a version controlled infrastructure. In [71], Git was able to be effectively included in courses for non-CS engineering majors as well as CS majors. This is particularly impressive as distributed version control tools can be known for their complexity and may be more challenging for non-CS majors.

5.1.2 Simplified Version Control and Visual Programming

Existing distributed version control tools are often criticized for their complexity [56, 22]. Although these tools are providing rather abstract and sophisticated features and capabilities, there have been some efforts to design more accessible, simple distributed version control tools [101, 67]. Additionally, some visual programming and collaborative editing environments provide their own approaches for version management [81, 96, 46]. These version management tools are often designed to provide similar capabilities to common version control tools without requiring the same cognitive load.

One approach to improving usability and accessibility of version control systems can be found in Gitless [101, 113]. De Rosso and Jackson performed a redesign of Git, a common version control system, addressing flaws in conceptual design. Specifically, they targeted conceptually challenging aspects of Git including eliminating the concept of an "assumed unchanged" file, the staging area, and stashing capabilities. Gitless also modifies the concept of a branch. Unlike in Git, Gitless branches include the working versions of files. When making a new branch, any uncommitted changes are not copied to the new branch and remain on the original branch.

When evaluated with real users, Gitless showed promise as an effective, simplified version

control tool [113]. Novices showed a statistically significant improvement in satisfaction and decrease in frustration. However, no significant difference was found in efficiency, difficulty or confusion. Some more advanced Git users expressed an appreciation for some missing Git concepts, specifically the stash and staging area: "Gitless was easier to use for the tasks these sessions asked me to perform, but I really like having a Git stash and staging area to work with in Git."

In [67], the applicability of version control to a non-technical audience is investigated. To this end, a web-based tool is developed for authoring E-learning content. This included a number of differences including renaming "branches" to "working copies," customized views for comparing differences and merge requests. One surprising challenge resulted from incomplete information when viewing differences. Unlike software development, where programming abstractions can promote assessing changes in isolation, authoring E-learning content presented challenges when viewing content changes in isolation.

This is an interesting approach to creating accessible version control support with similarities to Gitless. Like Gitless, the developed software supported a subset of the usual version control capabilities. Additionally, both made changes to the Git conceptual model to provide more accessible and easily understandable version control tools. However, unlike Gitless, Kreiser uses more commonplace terminology and an intuitive interface in an effort to make the core version control concepts accessible to a specific, non-technical domain.

One example of version control integration in a visual environment can be found in WebGME [81]. WebGME provides version control capabilities automatically. When a user is working in WebGME, every action is implicitly committed to the current branch. This guarantees that project edits will not be lost and that any state can be restored as the user can simply revert to the given commit corresponding with the state of the project which the user would like to revisit.

A similar approach is also used in other collaborative online tools. Although they are not visual programming environments, ShareLatex and Google Docs provide similar functionality with respect to implicit commits and allowing the user to "track changes" made to the current document [96, 46]. These changes contain not only the change but metadata about the change including author and creation time.

5.2 Approach

Providing distributed programming capabilities to novices can result in increasingly complex applications. Especially when developing more advanced applications, novices can be prone to making errors or mistakes while programming. Making some powerful features of

version control accessible to novices can help them to recover from errors. Additionally, providing accessible version control concepts can introduce novice programmers to important software engineering concepts implicitly.

In developing novice-friendly version control capabilities, the primary goal is to support users in reviewing changes to a project and restoring their projects to previous states. The version control capabilities should require minimal explicit instruction. As lively blocks-based programming environments have been designed for use in exploratory, constructivist settings [78, 79], it is important that these capabilities are accessible with minimal direct instruction.

To enable users to easily interact with the history of their project, we have designed another mode for the development environment called "Replay Mode". In Replay Mode the project becomes read-only and the user is presented with familiar controls similar to those used in media players like YouTube. These controls include a slider with tick marks for each action, a play button for moving through time normally (or at different rates depending upon more advanced settings), single stepping buttons and buttons for jumping to the beginning or the end of the current project's timeline. Furthermore, there are buttons for toggling subtitles which print the name of the currently occurring event and a settings button which enables the user to configure settings such as the replay speed and the maximum inactive duration between events. Figure 72 illustrates the updated user interface when viewing a project in Replay Mode.

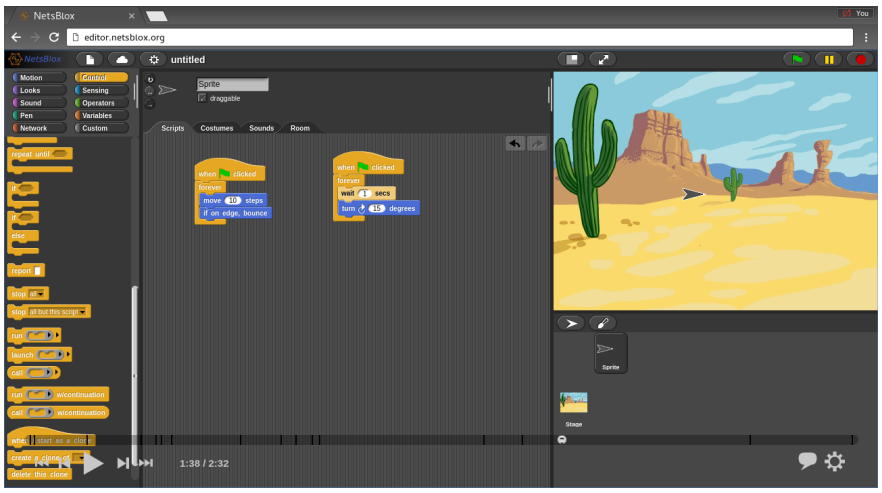


Figure 72: Viewing Project in Replay Mode

One fundamental capability of reverting to earlier versions of a project lies in the ability to revert project changes. Reverting project changes builds upon the operation-based infrastructure described in Chapter 4. As each project is composed of the previously defined operations, reverting an operation can be performed by computing the inverse of the given

operation. These inverse operations are computed based upon the type of the action, and it is computed based only on the contents of the given operation. If the operation requires additional information for inversion, the required information must be added to the operation during creation.

For example, consider the operation for resizing the stage, `setStageSize`. It certainly needs the new "width" and "height" parameters for the new stage. As the operation needs to be able to be inverted, it also needs to contain the current stage width and height. Consequently, before applying the operation, it saves the current stage width and height. This makes computing the inverse of the operation trivial; the inverse is simply a `setStageSize` operation using the old width and height.

Some operations cannot be undone with only a single operation from the operations defined in Section 4.3. One such example is the `moveBlock` operation which connects blocks together. Suppose the `turn` block was dropped between two other movement blocks and results in the script shown in Figure 73. As the `turn` block was placed between the `move` and `if on edge, bounce` blocks, reverting this operation should remove the given `turn` block and reconnect the remaining two motion blocks. This requires two different operations to restore the project to the previous state: one to remove the `turn` block and another to reconnect the remaining motion blocks.

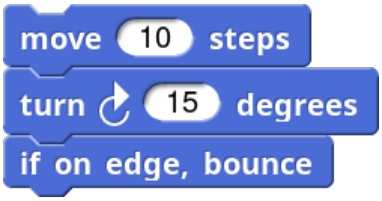


Figure 73: Operation Requiring Multiple Inverse Operations

As an operation may require multiple operations to restore a project to the previous state, we introduce the concept of a batch event. Batch events are an event which is composed of other events. As the events are contained within a single parent event, all the events are either accepted or rejected together. If accepted, the project is locked and batched events are applied sequentially without interruption. Currently, batch events are used only when inverting operations.

5.2.1 Fine-grained Reversion

Using invertible operations, the project content can be represented using a queue of the operations. Every operation can be applied to this queue. The only exception is when the project is in Replay Mode. As Replay Mode allows the users to view the history of the

project, the application of operations and their inverses should not be recorded as edits to the given project. Along with this project queue of all operations made on the project, operations can also be added to other queues to enable more fine-grained reversion of various aspects of the project. This provides the technical foundation for undo and redo capabilities.

Fine-grained reversion includes enabling the user to revert objects, such as sprites or the stage, to previous versions. Along with the global queue of actions applied for the given project, we can also define queues for specific aspects of each sprite or stage in a project. These aspects include the scripts, costumes or sounds for a given sprite.

Different aspects of the sprite are independent of one another; operations affecting any individual aspect of a sprite or the stage cannot affect any other aspect. For example, edits to the costumes of a sprite have no effect on edits to the block scripts of the given sprite. This independence ensures that each queue can be extended with operations (or inverses of existing operations) without any implications on the actions in other aspects. Independence of operations between queues enables operations from each queue to be constrained only by the relative order with respect to the other actions in the same queue. This allows the user to revert to past versions of each of these individual aspects easily.

One alternative approach would be to create operation queues based on the user authoring the original operation. This certainly could be useful when collaborating and is similar to the current behavior in applications such as Google Docs [46]. As discussed in Chapter 4, blocks-based programming uses a more complex underlying data structure than those used for many other tasks, such as text editing.

Blocks-based programming languages require the user to construct a forest of elements rather than simply a growable array of characters. As blocks have a set number of unique inputs, the nodes in these trees are constrained to only having a fixed number of named branches. This complexity introduces a number of constraints upon the data structure that result in non-trivial dependencies between operations. When organizing operations by author, there is no guarantee that different operation queues will have independent operations. To the contrary, it is actually quite likely that operation queues will contain conflicting actions.

One such example is shown in Figure 74. Two users, Alice and Bob, are collaborating on a project containing a single `turn` block. Alice places a `sum` block in the `turn` block's input and then removes it. Then, Bob places a `random` block in the same block's input and removes it. Alice and Bob's actions are represented with blue and red circles, respectively. The state of the block is provided before and after each operation.

This scenario demonstrates a simple example of dependent operations. If the users both had individual operation queues containing each of their operations, both should be able to

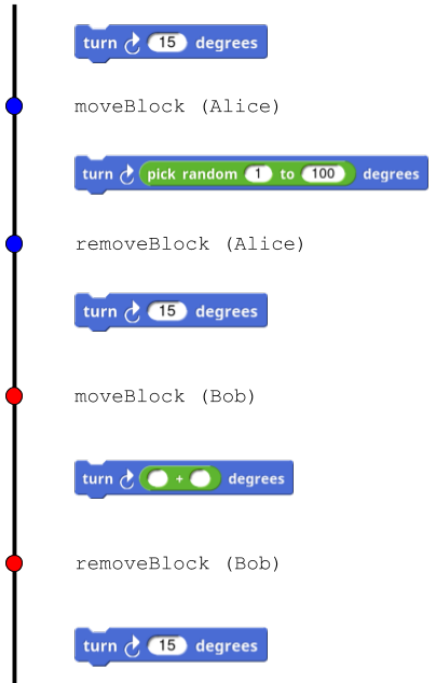


Figure 74: Dependent Operations in User-Based Operation Queues

revert their own operations independent of the state of other queues. However, as reverting the last action results in the input slot being occupied for each user, reverting the last operation for both users would result in the input being occupied by two different blocks simultaneously (which is impossible). As a result, they would be unable to both revert their last action.

An attempt to remedy this could be made by disallowing reverting operations which cannot currently be applied. Although this approach could be effective, it is certainly not ideal. As the operation queues allow a linear progression through the applied operations, disallowing a single operation can make all preceding changes inaccessible. The preceding changes could be important changes which are independent of the blocked event; this creates an unintuitive and seemingly arbitrary reason for losing any potentially important preceding changes.

Chapter VI

Empirical Support

6.1 Background and Related Work

6.1.1 Effectiveness of Educational Programming Environments

Educational programming languages have proven to be quite effective for introducing computer programming to young learners [139, 20, 110, 83]. These include simplified textual programming languages [21], blocks-based programming languages [110, 83] as well as other visual programming languages [30, 134]. Programming has also been shown to be beneficial for education and development beyond computer science in areas such as metacognitive development, computational thinking and promoting student engagement [23, 25]. This section provides an overview of the effectiveness of educational programming languages in computer science, student development, and engagement.

Meerbaum-Salant, Armoni, and Ben-Ari found that students were able to learn some fundamental computer concepts using Scratch. These concepts including loops, conditionals and events although they struggled with some other concepts: initialization, variables, and concurrency [83]. However, the authors conjecture that these shortcomings could be overcome by modifications to the curriculum and teaching process.

Denner, Werner, and Ortiz introduced computer programming in an after school class of 59 6th grade girls which met twice weekly for 14 months [34]. In this class, students used a visual programming environment designed specifically for developing games. Student games were then analyzed to detect use of various concepts including conditions, events and parallelism. 82% of student projects used conditions or events, 36% used parallelism and 29% used character variables. Similarly, in [79], Maloney, et al. found that students were able to learn basic computer programming concepts in a self-guided after school program with Scratch.

Seymour Papert claimed in "Mindstorms: Children, computers and powerful ideas" that "teaching the Turtle to act or to "think" can lead one to reflect on one's own actions and thinking" [98]. Many others also saw the potential for programming on education and student development [100]. This impact of programming on cognitive and metacognitive development has been supported empirically. In [23, 25], LOGO was found to improve student performance on cognitive and metacognitive assessments. More recently, [62] found that robotics programming improved student metacognitive development and geometric thinking.

"Computational thinking" (CT) involves "solving problems, defining systems, and understanding human behavior by drawing on concepts fundamental to computer science" [141] and has been defined as a fundamental skill for everyone. As defined by Wing, computational thinking is composed of six principles. The first principle is conceptualizing the problem and thinking at multiple levels of abstraction. Secondly, computational thinking is a fundamental skill for functioning in society. Thirdly, it consists of using cleverness to solve problems rather than thinking like a computer. Computer science draws upon both mathematics and engineering to build virtual worlds which exist with their own constraints and not necessarily the constraints of the physical world. The fifth principle focuses on the ideas and the approaches used to solve problems and not just the artifacts of the given task or exercise. The final principle is that CT should be a central part of human endeavors [141].

CT has already impacted a number of educational programs ranging from primary school through college [114, 105, 140]. Computational thinking has been incorporated into a number of curricula in general [18, 90, 7]. Educational programming environments are often used as a tool to promote and teach computational thinking such as in [18, 90, 7]. One example can be found in [15] where robotics programming was effectively used to teach aspects of computational thinking to children as young as four years old. In [20], a Scratch-based curriculum was shown to improve mathematics and problem solving skills (one aspect of computational thinking). Additionally, [114] found blocks-based programming to be an effective prerequisite for computational thinking among young students and [89] found App Inventor for Android to be an effective platform for introducing programming and computational thinking to K-12.

Educational programming environments have found empirical support for improving retention and promoting engagement among students [110, 142, 79, 61, 76]. In [61], Kelleher, et al., found that middle school girls using Storytelling Alice spent 42% more time programming than girls using the original Alice. Maloney, et al., found sustained engagement among Scratch users in an urban, after-school program without formal guidance [79]. Kaleliougl et al. found that girls performed equally with their male counterparts using a Code.org based curriculum which suggests equal engagement of both sexes [58]. Given this empirical support for promoting student engagement and retention, it is important to be able to provide tools which make computational thinking easily accessible [48].

Some educational programming environments have been able to make programming more engaging by making it more relevant. One example of this can be found in [142] in which App Inventor for Android was used to enable users to develop applications for their mobile devices. One particularly impressive example from [142] involves one student who built an

app which was used by a non-profit in Helsinki to organize a thousand-person event; this is certainly uncharacteristic for an undergraduate general education course.

6.2 NetsBlox

To evaluate the presented distributed programming abstractions, we have implemented the contributions in a blocks-based programming environment called "NetsBlox." NetsBlox is an extension of Snap*!*. Snap*!* is both open source and is one of the most advanced blocks-based programming environments with support for anonymous lists and functional programming concepts such as anonymous functions and custom blocks. As functional programming support is also an important prerequisite for the remote block execution described in Chapter 3, Snap*!* was a particularly useful foundation for NetsBlox.

6.3 Case Studies

We hypothesize that the presented networking abstractions will provide a number of benefits to student learning and engagement. We believe that these abstractions will enable novices to develop distributed applications and hence will enable users to develop a better understanding of important distributed computing concepts. Finally, we also believe that providing access to additional resources and making programming more social will also improve student interest and engagement. In this section, we present three case studies in which we evaluate these hypotheses.

6.3.1 SSMV Summer Camp

This study was conducted with 24 high school students who attended the School for Science and Math at Vanderbilt (SSMV) [126]. The program is a partnership between Vanderbilt University and Metro Nashville Public Schools. It is designed to teach research skills to high school students.

Students were taught for three hours per day for five days. As students may not have prior programming experience, the students first were taught basic computer programming concepts for the first two days. These concepts included control flow, events, lists, and custom blocks. After the brief introduction to programming, one day was spent introducing Remote Procedure Calls and another for message passing. On the final day, students were allowed time to work on projects of their own choosing.

Students were given both a survey, a pre- and post-test, and then provided anonymous

feedback at the end of each day. The survey collected demographic information and attitudinal information about networking and computer programming in general. Additionally, the survey contained questions about about the usage of networking functionality in their final projects. This includes measuring the students' level of agreement to questions such as "I am interested in computer science" and "I know how to build a networked application." The pre- and post-tests[1] contained 6 questions and were used to assess computational thinking, networking and concurrency competency. These included questions about the behavior of connected devices in a mesh network and adaptations of the Two Generals Problem, a classical problem in distributed computing.

Students showed significant improvements in both the computational thinking and networking sections ($p < 0.01$). The effect sizes for the improvements in computational thinking and networking competency were 0.69 and 0.76, respectively. Scores in computational thinking showed an average improvement of 16.3 percentage points (pp). The networking section scores showed a slightly higher improvement of 19.3 pp. Although students showed improvement in the concurrency section, they struggled more with these questions. This is likely a result of the lack of alignment between the questions and the curriculum. Due to time constraints, concurrency topics were not explicitly discussed.

	Pre-Test		Post-Test	
	M	SD	M	SD
Comp. Thinking	78.0	30.0	93.4	14.5
Networking	59.0	26.2	78.1	24.3

Table 6.1: Student Scores in CT and Networking Sections

For the final project, students were allowed to create a project of their choosing. These included a variety of projects such as an online, multiplayer "Tron" clone and an application which enabled users to explore an interactive map of the world and learn statistics about countries (such as population) by clicking on the country. Results showed 85% of students reported using distributed features in their final projects. Of the students using distributed features, 55% reported using Remote Procedure Calls in their final project and 77% reported using messages.

Student feedback also provided promising, positive support for the experience particularly with respect to the networking capabilities. Many responses reported interest and positive experiences pertaining to the networking capabilities. Some examples of such responses are given below.

- "We learned how to do this in chat rooms, as well as across servers. I thought this

[1]The complete pre- and post-tests can be found in the appendix.

function was really neat, and was the most interesting part about coding to learn. I really enjoyed being able to expand beyond just one computer and be able to work through multiple places."

- "I liked this because it was cool to see how you can communicate with people across the globe using a few lines of code"
- "I really enjoyed getting into multi-computer messages and games because it is my first experience making anything close to an online program."
- "Even though this week is over, I'm still going to continue working on coding to improve my skills."
- "As we continue to learn more about the program, the easier and more natural the coding seems. This has been true throughout our learning with NetsBlox"

6.3.2 Budapest Summer Camp

During July 2017, we conducted a week-long study with 16 students in Budapest, Hungary. The students were self-selected by their interest in computer science and were between the 6th and 11th grade. Students were taught for 6 hours per day for 5 days. As in the SSMV study, prior programming experience could not be assumed.

The courses followed the curriculum from the SSMV study. Students were assessed using a survey and pre- and post-tests. The assessment materials used in the SSMV study were translated into Hungarian by a native speaker and were used to assess student understanding and disposition towards programming.

Students showed statistically significant improvements in both computational thinking and networking competency ($p < 0.05$; $p < 0.01$, respectively). The effect sizes for the improvements in computational thinking and networking competency were 0.72 and 1.01, respectively. Computational thinking scores increased by an average of 21.4 percentage points (pp) and scores in networking increased by an average of 20.3 pp. Similar to the SSMV study, students struggled more in the concurrency section. This is likely due to the use of the same curriculum and lack of explicit discussion of concurrency topics.

	Pre-Test		Post-Test	
	M	SD	M	SD
Comp. Thinking	67.2	31.3	88.5	27.7
Networking	51.6	21.4	71.9	19.0

Table 6.2: Student Scores in CT and Networking Sections

This study also concluded with the students developing final class projects of their choosing. These programs were quite varied and included a number of interesting applications such as an encrypted chat client and a hangman game. It was encouraging to see interest in the networking capabilities by the younger students as well. One of the 6th graders made a simple distributed program with two clients. When the game started, one Role would send a "hello" message and the other Role would wave in response. Although this game was simple, it provided positive support for the effects of networking capabilities on student engagement and interest.

6.3.3 Fifth Grade Science Classroom

In the fall of 2017, we were invited to a 5th grade science classroom to teach a lesson using NetsBlox. This lesson focused on integrating computer science concepts with concepts from their science curriculum. The school was an all-female private school in Nashville, Tennessee. We taught a 45-minute class with four total sections.

Like the previous case studies, we could not assume prior programming experience. The lesson contained a driving problem which introduced many of the basics of computer programming, including control flow, variables, and events. Additionally, the motivating example included an introduction to one networking abstraction, Remote Procedure Calls. Remote Procedure Calls were used to provide access to the scientific content. This consisted of two real-world data sources, map images, and current weather data.

Interactive Weather Application

The lesson was focused on developing an interactive weather application using NetsBlox. The completed application showed the user's current location on a map and allowed her to change the zoom levels using the +/- keys. Clicking on the map displayed the city name of the given location with the current temperature. During the lesson, the instructor first demonstrated a concept as the students followed along. Then the students were prompted to complete a related task on their own. For example, the teacher may demonstrate the addition of increasing the map zoom and the students will then be prompted to add capabilities for decreasing the zoom level. The completed application is shown in Figure 75.

The interactive weather application was selected as the motivating example for multiple reasons. First, it was a relatively simple application which easily incorporated scientific concepts such as latitude and longitude during development. After development, the application enabled students to explore temperature trends throughout the world (as many of them did). This exploration enabled them to gain insight into weather variations using an application

Figure 75: Interactive Weather Application

that the students wrote themselves. Using their own application to inspect current temperatures throughout the world provided both relevance to the programming exercise while also promoting scientific inquiry and curiosity.

This motivating project was decomposed into segments which motivated a number of important computer programming concepts while still being practical for a 45-minute lesson. First, the programming environment was explained to the students. This included an overview of the concepts of the blocks, hat blocks (blocks which start a script), stage, sprites, and green flag (usually used to start the application). The liveness of the environment was also demonstrated. Secondly, costumes were explained including the related blocks.

After introducing costumes, the `call RPC` block was introduced with the "getMap" RPC from the "GoogleMaps" Service. As this block returns the requested map as a costume, the map was then set as the costume for the stage using the blocks presented previously. Next, students were encouraged to modify the arguments to the "getMap" RPC. Specifically, we focused on how changing the "zoom" argument affected the resultant map costume. This lead us to an introduction of variables as a means to change the actual value for the "zoom" argument without requiring the user to manually change a hard-coded value for the given argument.

Using a variable for the current zoom level naturally lead to introducing key events as the means for the user to increase or decrease the zoom level (and update the map). Increasing the zoom level was completed together as a class. Students were then prompted to add support for decreasing the zoom level. Adding the zoom capabilities completed the creation of the interactive map.

Displaying the weather was presented by first introducing the problem of notifying the sprite when the stage has been clicked. This motivated the introduction of events as a simple means of communication between different sprites (or the stage). Next, we had the sprite move to the pointer when the stage was clicked. The differences between the screen coordinates and the Earth's coordinates (latitude and longitude) were discussed and the coordinate conversion RPCs are introduced. Using the computed latitude and longitude, the students then used the `temperature` RPC from the "Weather" Service to retrieve the current temperature for the given location.

User Feedback and Results

After the lessons, anonymous feedback was collected via a short survey about the student's experience. The survey consisted of five questions focused on their interest in the lesson and interest in learning more about programming.

Students responded positively to programming. This was reflected both in the responses about the individual lesson and their interest on programming in the future. The majority of students (89.8%) reported that they found the class interesting. When prompted if they would like to know more about coding, 85.7% of students reported that they would like to know more. Additionally, when asked if they would like to do a class "creative coding project," over 80% confirmed that they would like this class project in the future.

Students that found the lesson interesting were also asked what they found interesting about the lesson. As this prompt was open-ended, the responses were varied. Nonetheless, they showed some general trends particularly relating to the networking capabilities and student perceived self-efficacy.

Responses were classified by their relationship to the provided networking capabilities. These types of comments are based on specific mention of features provided by the networking functionality. This included comments pertaining to the map and temperature functionality of the application. As the maps were provided by Google Maps, one comment which specifically mentioned "controlling Google" was also considered a reference to the resources provided by the networking capabilities.

Using these criteria for classification, we labeled the student feedback pertaining to the networking capabilities provided in NetsBlox. We found that 28% of student responses were directly related to features provided by the networking capabilities. Some responses are included below:

- "I found the fact that you could use network to figure out temperature in different places"

- "I liked where it showed us the maps and the temp"
- "The way we could get maps was very interesting"
- "We got to know the temperatures of different places."
- "It was cool to make a weather app!"

These responses provide some interesting insight into the students understanding and mental processes. A number of the responses, such as "We got to know the temperatures of different places" and "I liked where it showed us the maps and the temp," highlight the students' interest in the incorporated science content provided by the Remote Procedure Call. One student responded that "I liked that we got to make something we can actually use in everyday life." Although this does not explicitly reference networking or weather capabilities, it supports our hypothesis that the inclusion of real-world data should make programming more relevant and meaningful for some students.

Comments were also classified by their relationship to students' perceived self-efficacy. These types of comments are characterized by the improved perceptions of the student's capabilities or the accessibility of the lesson material (such as expressing surprise that she was "able to make an app that easily"). Student comments were considered to be related to the students' perceived self-efficacy if the comment referenced the students capabilities and independence (often through the use of reflexive pronouns or stating capability such as "I can make") relative to the lesson material. We found that 17% of student responses were related to feelings of self-efficacy and empowerment. Some responses are provided below:

- "Being able to do it myself. Plus it was super cool."
- "Today I found it interesting that thats all it took to make that. I thought it would take longer"
- "what I found interesting is that I can make a small app."

There were a number of responses which were not considered related to networking capabilities nor student self-efficacy. Despite not being a part of a broader theme, these responses were still noteworthy and provided positive feedback about the overall lesson. These include comments reflecting a deeper understanding (and possible appreciation) of programming such as "how many coding things you really need to use to just do a little thing" and "You really have to pay attention to do the right thing."

These responses provide some interesting insight into the students understanding and mental processes. The prominence of responses pertaining to features provided by the networking capabilities, without specific prompting, provides some qualitative support for our

hypothesis that these features should make programming more interesting and engaging. The empowering comments around perceived self-efficacy seem to suggest that there may be additional benefits to student self-efficacy following the lesson. Although noteworthy, this requires further investigation and could be an interesting opportunity for future work.

6.4 Discussion

We have presented three case studies investigating the following hypotheses:

- The networking abstractions will enable novices to develop distributed applications.
- Building distributed applications with these abstractions may enable users to develop a better understanding of important distributed computing concepts.
- Providing access to additional resources and making programming more social will improve student interest and engagement.

The first two case studies used NetsBlox to present the given abstractions to students in two summer camps. In these studies, students were introduced to computer programming and then distributed programming in an accelerated, one-week curriculum. During the study, students developed a number of different distributed applications including chat and mesh networking applications.

These two studies provided strong support for our first hypothesis. Using the present distributed programming abstractions, students were able to develop distributed applications as part of the curriculum. Furthermore, students were also able to use these abstractions in their own final projects. These included various distributed applications such as turn-based multi-player games, real-time multi-player games and chat clients.

The second hypothesis was supported by the first two studies. To assess student understanding of distributed abstractions, pre- and post-tests were developed to assess understanding of various distributed computing concepts. These included variations of the Two General's problem and the behavior of nodes in a mesh network. In both summer camps, the students showed statistically significant improvements in questions pertaining to distributed computing concepts. These improvements were large as both studies showed approximately a 20% improvement from the pre-test to the post-test scores. The effect sizes for the studies reflected this with values of 0.76 and 1.01.

The third hypothesis is also supported in these studies. As the students were able to choose their final projects, the use of distributed programming abstractions suggests student interest in these concepts. The significant use of the distributed programming capabilities

(84.62% in the first study) suggests these concepts were interesting to the students. Although this is a relatively informal metric, it provides promising support for the hypothesis that making distributed programming accessible to novices will positively influence student interest and engagement.

The last case study focused on using the Remote Procedure Call abstraction to integrate computer science and science curriculum. Students in this study were younger than the previous case studies. These students were not self-selected and were students in a regular fifth grade science class. Despite not using many of the different networking concepts, this study investigated the benefits of integrating real-world datasets (through Remote Procedure Calls) to student interest and engagement.

This case study investigated the effects of the distributed programming abstractions on student engagement, the final hypothesis. Most students (about 90%) found the lesson interesting. Of these students, 28% of these students (25% of all students) explicitly stated features provided by the distributed programming primitives as the interesting aspects of the lesson. Over 25% of students cited the features provided by the distributed programming abstractions as the parts of the lesson which they found interesting. This provides significant support for promoting engagement and interest among novice programmers.

These case studies provide positive support for our hypotheses. However, there are still a number of remaining questions and weaknesses. The first two studies consisted of self-selected students and were not a representative sample of a general high school population. Furthermore, these studies were all relatively short in duration. Future work could investigate if these results will generalize to the high school classroom or to a course of extended duration, such as a high school course.

Chapter VII

Conclusion and Future Work

In this dissertation, we have presented abstractions for making distributed computing accessible to novice programmers. This includes designing a compiler to enable the execution of blocks in alternative execution environments, such as remote server environments. We have also presented an approach to collaborative editing in a lively, blocks-based environment and a technique to bring version control capabilities to novices designed to require minimal direct instruction. Finally, we have conducted case studies investigated the effectiveness in making distributed computing accessible to novices using an implementation of these prior contributions called NetsBlox.

7.1 Contributions

Distributed Programming Abstractions

In this work, we presented novel abstractions for enabling novices to develop distributed applications in a blocks-based programming environment. These abstractions provided support for using message passing between blocks-based applications and an abstraction for local networks and clients, the Room and Roles. Rooms and Roles also provide a natural way to generate understandable network addresses for each application. Additionally, Remote Procedure Calls (RPCs) provide a standard way to interact with external resources, utilize scaffolding for developing more challenging applications and execute custom block functions on the hosting server.

Remote Block Execution

To support the execution of custom block functions in an RPC, we have developed a technique for executing arbitrary block functions in alternative environments. This included designing a cross-compiler from the original block language to JavaScript. The compiler supported configurable block behavior in which the actual block implementations are decoupled from the compiled code and can be specified at runtime. Conforming to the concurrency model of the origin blocks-based environment was also a driving goal in the design. Additionally, considerations for supporting closures and ensuring safety in the generated code was also addressed and discussed.

Collaborative Editing

Collaboration in lively, blocks-based programming introduces a number of conceptual and technical challenges. We presented a model of collaboration for use in a lively environment which promotes productivity. This included a strongly-consistent, operation-based approach to collaboration in these blocks-based environments. Moreover, we also addressed additional complexities such as the mitigation of unexpected behavior as a result of collaboration in a lively, distributed application.

Novice-Friendly Version Control

Version control can be very beneficial but difficult to understand for novices. Furthermore, modern version control tools predominantly operate on text content and present changes accordingly. We presented version control capabilities in a blocks-based programming environment. This version control is not only designed to be accessible for novices with minimal direct instruction, but also presents a natural way to enable users to interact with the history of their projects.

Empirical Support

We introduced NetsBlox, an implementation of the discussed abstractions and concepts. Three case studies have been presented to evaluate the effectiveness of the prior contributions using NetsBlox. These case studies investigated the effectiveness of teaching programming to two summer camps and fifth graders at a private school in Nashville, Tennessee. In the summer camps, we found that students were able to develop some basic distributed applications over the course of a single week. Students also showed significant improvement in computational thinking and understanding of networking concepts. When creating projects of their choosing, we found the strong majority of students utilized distributed programming capabilities.

We found similar positive results when investigating student engagement in the third study. This study utilized Remote Procedure Calls to teach programming and science concepts simultaneously to students in the fifth grade. About 90% of students reported finding the class interesting and over 25% of students explicitly cited features provided by the networking capabilities as the interesting aspect(s) of the lesson.

7.2 Future Work

Deploy Block Programs to Alternative Environments

We have currently designed and implemented a compiler for generating JavaScript with configurable blocks implementations. This compiler has been utilized to enable users to execute block functions on the hosting server. The configurable block implementations enables the target execution environment to provide block implementations corresponding to the capabilities of the given environment. This capability could support the deployment to other types of execution environments as well.

Deploying user's block programs to other platforms provides significant pedagogical opportunities. These include deploying projects "to the cloud" as a long-running application and could be used to teach concepts such as deploying services to support client applications. Currently, block programs only execute in the browser (with the exception of executing specific functions on the server) and this new feature could introduce the concept of creating a web service which executes independent of any user.

Robotic platforms or the Internet of Things could provide more interesting deployment targets. This could enable users to program their applications and deploy the generated code on the given device. After deployment, the given device would not necessarily need a physical or network connection to run but could still utilize the sophisticated features of the original block environment. This includes concurrency (even if parallel execution is not supported on the device) and the powerful functional programming features provided by blocks-based programming environments such as Snap*!*.

Intelligent Learning Environments

The development of the operation-based infrastructure used for both the collaborative editing and the version control support provides many opportunities for data collection and mining. This provides opportunities for gaining insight into many difficult pedagogical challenges. These include early detection of students who may need help and behaviors of successful students. Additionally, it would be interesting to apply contemporary machine learning techniques to these operations to provide intelligent scaffolding tools for the students.

The use of these operations in collaborative environments also provides an opportunity for providing for learning about student behavior while collaborating. This includes simple things such as division of labor and potentially the effects of the actions of each student. For example, it could be interesting to investigate collaboration scenarios in which student

actions are not beneficial and actually cause problems with one another. These scenarios could provide another opportunity for intervention by an intelligent system.

Classroom Studies

The provided case studies have shown promising results when using NetsBlox as a vehicle for the provided abstractions and capabilities. However, these studies have been of short duration. Additionally, two of the studies were performed with self-selected students with an interest in computer science. It would be interesting to investigate effects in a longer study in an environment like an after school program.

Although the core contributions have been evaluated, some of them were not implemented at the time of the studies. Specifically, it would be interesting to investigate the effects of introducing the remote block execution on student understanding of Remote Procedure Calls and on student thinking with respect to distributed computing. Unlike some of the previous abstractions and concepts, executing custom block functions can be used to easily demonstrate the importance of considering network latency when developing a distributed system.

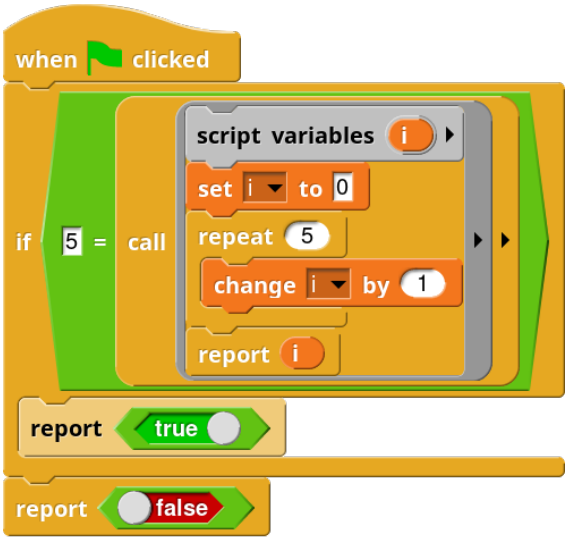
The version control capabilities also provide interesting opportunities for future studies. Could activities such as annotating the history of the student project have positive effects on student metacognition? Enabling students to interact with the history of his or her project should be beneficial to debugging a program with a recently introduced bug. This potential benefit to productivity and student problem solving could also provide another interesting aspect to explore.

Appendix A

Sample Generated JavaScript Code

In this appendix, we present the advanced example block script from Chapter 3 and the complete generated code output. This example invokes an anonymous function which yields control of the process during execution and introduces some complexities of preserving the implicit concurrency model in blocks programming languages such as Snap*!* and Scratch. Although the compiler can optionally generate a JavaScript file including all dependencies, such as the environment context, we will only provide the output of the compiler when only generating the JavaScript code for the given block program (without bundling any dependencies).

A.1 Sample Input Block Scripts



A.2 Generated JavaScript output

```
function anonymous(__ENV) {
  const SPromise = __ENV.SPromise;

  class Variable {
    constructor(name, value) {
      this.name = name;
      this.value = value;
    }
  }

  class VariableFrame {
    constructor(parent) {
      this.parent = parent;
      this.vars = {};
    }

    get(name, silent) {
      let result = null;
      if (this.vars[name]) {
        result = this.vars[name];
      } else if (this.parent) {
        result = this.parent.get(name, silent);
      }
      if (result === null && !silent)
        throw new Error(
          "a variable of name '" + name + "' does not exist in this context"
        );
      return result;
    }

    set(name, value) {
      this.vars[name] = new Variable(name, value);
    }
  }

  class Stage {
    constructor(name, variables, customBlocks) {
      this.name = name;
      this.variables = new VariableFrame(variables);
      this.customBlocks = new VariableFrame(project.customBlocks);
      this.isSprite = false;
      this.variables.set("__SELF", this);
      this.project = project;
    }

    onFlagPressed() {}

    onUserEventReceived(event) {}

    emit(event, wait) {
      if (wait) {
        project.sprites
          .concat([project.stage])
          .forEach(obj => obj.onUserEventReceived(event));
      } else {
        setTimeout(() => {
          project.sprites
            .concat([project.stage])
            .forEach(obj => obj.onUserEventReceived(event));
        }, 0);
      }
    }

    getTimerStart() {
      return project.timerStart;
```

```
  }

  resetTimer() {
    project.timerStart = Date.now();
  }

  getTempo() {
    return project.tempo;
  }

  setTempo(bpm) {
    return (project.tempo = Math.max(20, +bpm || 0));
  }
}

class Sprite extends Stage {
  constructor(name, variables, customBlocks) {
    super(name, variables, customBlocks);
    this.clones = [];
    this.isSprite = true;
    this.xPosition = 0;
    this.yPosition = 0;
    this.direction = 90;
    this.costume = 0;
    this.size = 100;
  }

  clone() {
    let clone = Object.create(this);
    this.clones.push(clone);
    setTimeout(() => clone.onCloneStart(), 0);
  }
}

__ENV = __ENV || this;
var project = {
  variables: new VariableFrame(),
  customBlocks: new VariableFrame(),
  timerStart: null,
  tempo: 60,
  sprites: []
};
project.stage = new Stage(
  unescape("Stage"),
  project.variables,
  project.customBlocks
);

project.stage.onFlagPressed = function() {
  var self = this;
};

project.stage.onUserEventReceived = function(event) {
  var self = this;
};

project.stage.onKeyPressed = function(key) {
  var self = this;
};

// for each sprite...
var sprite;

sprite = new Sprite(
  unescape("Sprite"),
  project.variables,
  project.customBlocks
);
```

```
sprite.xPosition = 0;
sprite.yPosition = 0;
sprite.direction = 90;
sprite.draggable = true;
sprite.rotation = 1;
sprite.size = 100;
sprite.costumeIdx = 0;

sprite.onFlagPressed = function() {
  var self = this;

  (function() {
    var __CONTEXT = new VariableFrame(self.variables);
    callMaybeAsync(
      self,
      __ENV.doIf,
      callMaybeAsync(
        self,
        __ENV.reportEquals,
        unescape("5"),
        callMaybeAsync(
          self,
          __ENV.evaluate,
          function() {
            return new SPromise((callbackitem_3, rejectitem_3) => {
              var context = new VariableFrame(arguments[0] || __CONTEXT);
              var self = context.get("__SELF").value;
              __CONTEXT = context;
              callMaybeAsync(
                self,
                __ENV.doDeclareVariables,
                unescape("i"),
                __CONTEXT
              )
                .then(() =>
                  callMaybeAsync(
                    self,
                    __ENV.doSetVar,
                    unescape("i"),
                    unescape("0"),
                    __CONTEXT
                  ).then(
                    () =>
                      new SPromise(
                        (
                          resolve_item_14_1518137777382,
                          reject_item_14_1518137777382
                        ) => {
                          function doLoop_item_14(item_14) {
                            return callMaybeAsync(
                              self,
                              __ENV.doIfElse,
                              item_14-- > 0,
                              () => {
                                callMaybeAsync(
                                  self,
                                  __ENV.doChangeVar,
                                  unescape("i"),
                                  unescape("1"),
                                  __CONTEXT
                                )
                                  .then(() =>
                                    callMaybeAsync(
                                      self,
                                      __ENV.doYield,
                                      doLoop_item_14,
                                      item_14,
                                      __CONTEXT
```

```
                                    )
                                  )
                                  .catch(reject_item_14_1518137777382);
                              },
                              () => {
                                callMaybeAsync(
                                  self,
                                  callbackitem_3,
                                  callMaybeAsync(
                                    self,
                                    __ENV.variable,
                                    unescape("i"),
                                    __CONTEXT
                                  ),
                                  __CONTEXT
                                )
                                  .then(() => resolve_item_14_1518137777382())
                                  .then(reject_item_14_1518137777382);
                              },
                              __CONTEXT
                            ).catch(reject_item_14_1518137777382);
                          }
                          callMaybeAsync(
                            self,
                            doLoop_item_14,
                            unescape("5"),
                            __CONTEXT
                          );
                        }
                      )
                    )
                  )
                  .catch(rejectitem_3);
              });
            },
            __CONTEXT
          ),
          __CONTEXT
        ),
        function() {
          callMaybeAsync(self, __ENV.doReport, true, __CONTEXT);
        },
        __CONTEXT
      ).then(() => callMaybeAsync(self, __ENV.doReport, false, __CONTEXT));
    })();
};

sprite.onKeyPressed = function(key) {
  var self = this;
};

sprite.onUserEventReceived = function(event) {
  var self = this;
};

sprite.onCloneStart = function(event) {
  var self = this;
};

project.sprites.push(sprite);

project.timerStart = Date.now();
__ENV.__start(project, __ENV);

function callMaybeAsync(self, fn) {
  let args = [].slice.call(arguments, 2);

  // Wait for any args to resolve
```

```
    let result = __ENV.SPromise.all(args);

    result = result.then(args => fn.apply(self, args));

    return result;
  }
  callMaybeAsync = __ENV.callMaybeAsync || callMaybeAsync;

  project.sprites.forEach(sprite => sprite.onFlagPressed());
  project.stage.onFlagPressed();
}
```

Appendix B

Case Study Assessment

This appendix contains the pre- and post-tests used to assess computational thinking, networking and concurrency concepts.

Name:_____ Date:_____

Part I - Computational Thinking (CT) Questions

Question 1:

Emma writes code which says

Repeat 2
[Do a math problem]
Write an essay

while

John writes code which says
Repeat 2
[Do a math problem
Write an essay]

Which of the following statements is correct?

a. Both Emma's and John's code say: Do 2 Math problems and then write 2 essays.
b. Emma's code says: Do 2 Math problems and then write one essay, while John's code says: Do 2 Math problems and then write 2 essays
c. Emma's code says: Do 2 Math problems and then write an essay, while John's code says: Do a Math problem, write an essay, then Do a 2nd Math problem and then write a 2nd essay
d. Emma's code says: Do a Math problem, then write an essay, and then Do a 2nd Math problem, while John's code says: Do a Math problem, then write an essay, then Do a 2nd Math problem and then write a 2nd essay

Question 2:

Consider the following program (NOT nested)

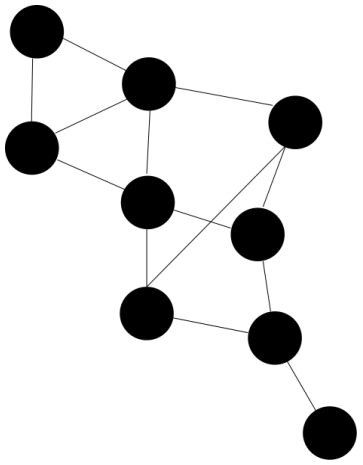
If (quiz-score is equal to 10)
Then: Get the ‘You’re a pro’ sticker
Else: _____
If (quiz-score is greater than 7)
Then: Get the ‘Good job’ sticker
Else: _____

Bill gets a score of 9 on the quiz while Janet scores 10 points on the quiz. What stickers should Bill and Janet receive based on the above program?

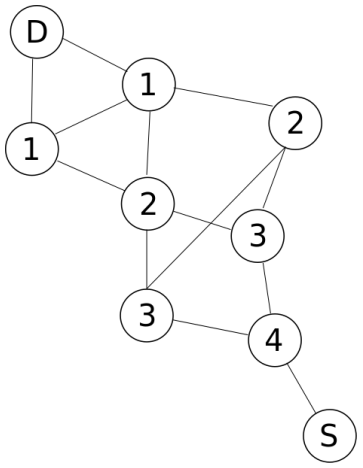
a. Bill: ‘Good job’ sticker; Janet: ‘You’re a pro’ sticker
b. Bill: ‘Good job’ sticker; Janet: ‘Good job’ sticker
c. Bill: ‘Good job’ and ‘You’re a pro’ stickers; Janet: Good job’ and ‘You’re a pro’ stickers
d. Bill: ‘Good job’ sticker; Janet: Good job’ and ‘You’re a pro’ stickers

Part II - Networking Questions

Question 1:
Suppose you have a network of devices as shown below where a device is represented by a circle and the other devices that it can directly message are connected to it.

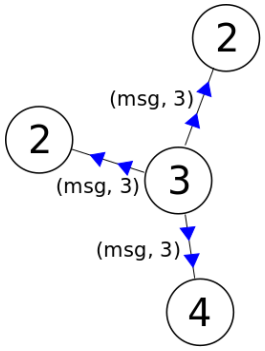


If you want to send a message from the bottom node to the top node through these connected devices, one technique is to first give each node a number representing the distance from the top node as shown below:



Then, starting at the source (S) device, each device can send a message containing the original message and its own number to everyone connected to it when it receives a message from another device with a number higher than its number.

For example, when the “3” device receives a message (“msg”), it will send it to the three nodes connected to it:

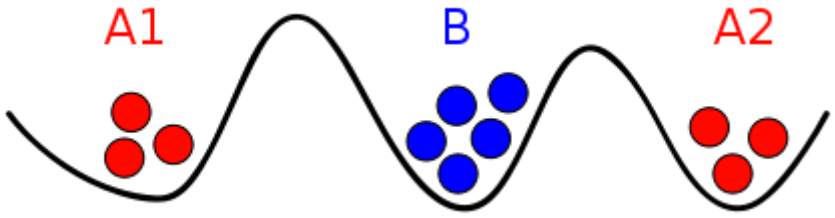


This will result in the “2” nodes broadcasting “(msg, 2)” to all their connected nodes since 2 < 3 but node “4” will ignore the message and will NOT broadcast it to its neighbors as 3 (from the message) is less than its number, 4.

This technique will result in the message being received by the “D” device in the fewest number of steps. In this example, it is important that each device sends the received message **and** its own number. Why?

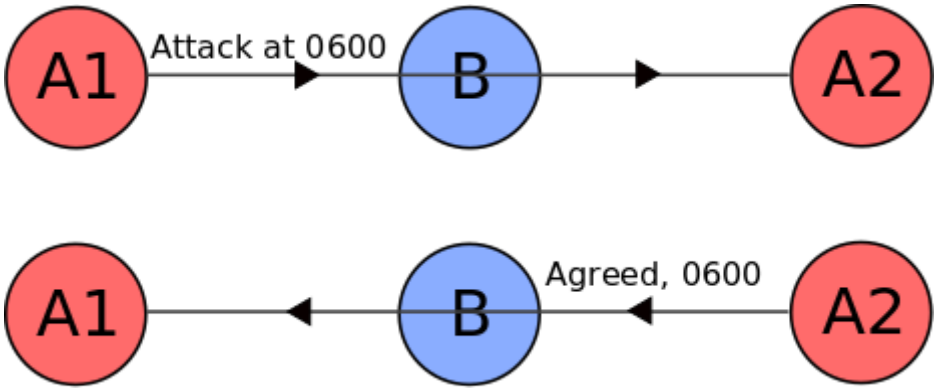
What would happen if each device sent only the message (“msg” in the above example) and not its own number?

Question 2:
Two armies are planning on attacking a city and are located on opposite sides of the city. Both armies would like to make sure that they attack at the same time; if they attack individually, they are sure to lose. As they are on opposing sides of the city, they must send a messenger through the city to communicate with the opposing camp and the messenger may be captured.



These armies would like to agree on the time of their attack but are communicating unreliably as **any** messenger could be captured and may not deliver the message. For example, if *A1* decides to attack at 0600, it may send the message “We should attack at 0600 on June 28” to *A2* but if the messenger gets captured, *A2* will not attack at that time and *A1* will be defeated.0

In order to solve this problem, suppose we require a single confirmation message. That is, *A1* will send a time to attack and wait for a confirmation from *A2*. Upon receiving the confirmation, *A1* will attack at the proposed time:



That is, *A1* will attack if it receives an agreement message about the proposed time and *A2* will attack if it receives a proposed attack time from *A1.*

Using this messaging technique, are the two armies guaranteed to attack at the same time? If not, which army might attack alone?

Part III - Concurrency Questions

Question 1: ***It's a rat race (condition) out there…***
reverse reverses the order of the digits, and **right** rotates the digits to the right (the one furthest on the right moves over to the left). If NUMBER starts with a value of **123**, what are the *different, unique values* of NUMBER at the end? (select all that apply) *Note: "**set**" blocks run instantaneously and "atomically" (i.e., aren't interrupted by any other script)*

☐ 123	☐ 124	☐ 125	☐ 132	☐ 133	☐ 142	☐ 213	☐ 214	☐ 231	☐ 232	☐ 312	☐ 313	☐ 321	☐ 322	☐ 412	☐ 421	☐ 423	☐ 424	☐ 431	☐ 432	☐ 433

Question 2: If 75% of a program's code is parallel, what is the maximum speedup with ∞ helpers? (select ONE)

○	○	○	○
25x	75x	1.33333x	4x

Name:_____ Date:_____

Username(s): _____

Part I - Computational Thinking (CT) Questions

Question 1:

You are training a robot to avoid obstacles as it moves. To make things more interesting you tell the robot to turn right to go around the obstacle if the color of the obstacle is taller than 5 inches. Otherwise, the robot should turn left to go around the obstacle. How will you program your robot to follow these instructions using an **If-Then-Else** structure?

If: _____

Then: _____

_____

Else: _____

_____

Question 2:

Tommy writes code which says

Repeat 4
[Read a book]
Say hello

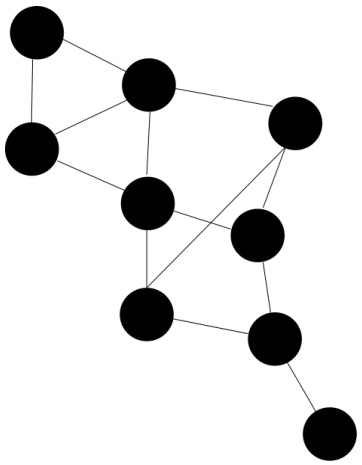
while
Elizabeth writes code which says
Repeat 4
[Read a book
Say hello]

Which of the following statements is correct?

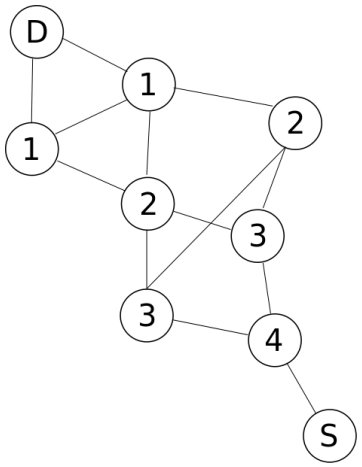
a. Tommy's code says: Read a book, then say hello, and then read 3 more books, while Elizabeth's code says: Read a book, say hello, read 2nd book, say hello, read 3rd book, say hello and read 4th book and say hello
b. Tommy's code says: Read 4 books and then say hello once, while Elizabeth's code says: Read 4 books and then say hello 4 times
c. Both Tommy's and Elizabeth's code say: Read 4 books and then say hello 4 times.
d. Tommy's code says: Read 4 books and then say hello once, while Elizabeth's code says: Read a book, say hello, read 2nd book, say hello, read 3rd book, say hello and read 4th book and say hello

Part II - Networking Questions

Question 1:
Suppose you have a network of devices as shown below where a device is represented by a circle and the other devices that it can directly message are connected to it.

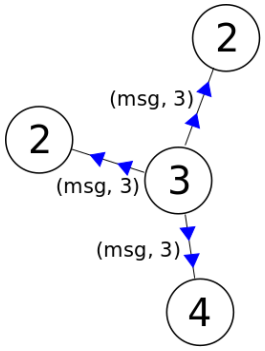


If you want to send a message from the bottom node to the top node through these connected devices, one technique is to first give each node a number representing the distance from the top node as shown below:



Then, starting at the source (S) device, each device can send a message containing the original message and its own number to everyone connected to it when it receives a message from another device with a number higher than its number.

For example, when the “3” device receives a message (“msg”), it will send it to the three nodes connected to it:



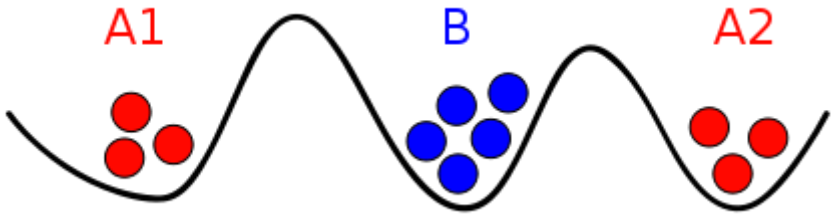
This will result in the “2” nodes broadcasting “(msg, 2)” to all their connected nodes since 2 < 3 but node “4” will ignore the message and will NOT broadcast it to its neighbors as 3 (from the message) is less than its number, 4.

This technique will result in the message being received by the “D” device in the fewest number of steps. However, as each node broadcasts more than one copy of the message, there will be multiple copies of the message being sent through the network on the way to device “D”.

How many times will “D” receive the message?

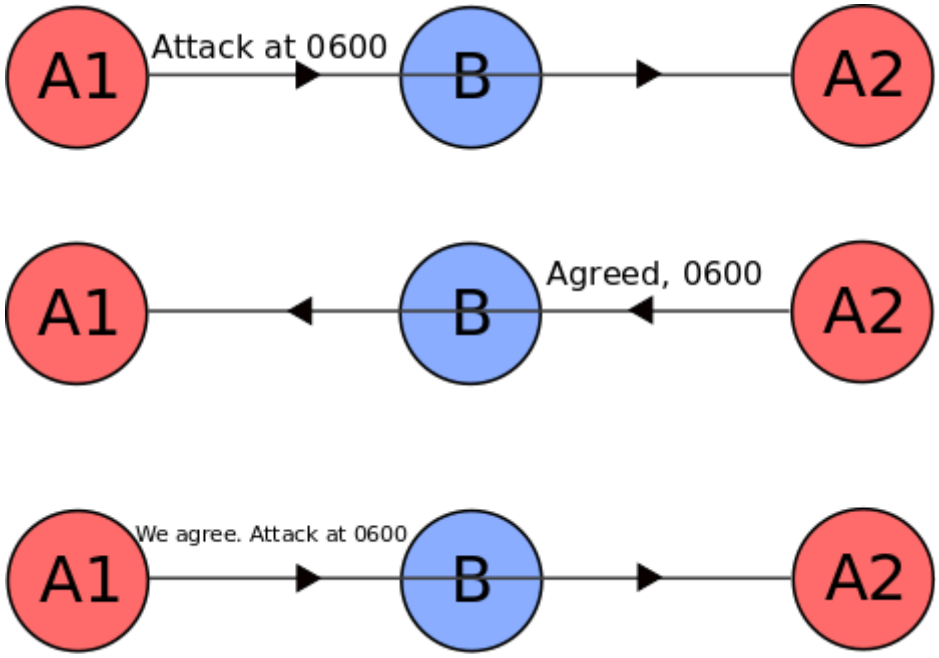
a. 0
b. 1
c. 2
d. more than 2

Question 2:
Two armies are planning on attacking a city and are located on opposite sides of the city. Both armies would like to make sure that they attack at the same time; if they attack individually, they are sure to lose. As they are on opposing sides of the city, they must send a messenger through the city to communicate with the opposing camp and the messenger may be captured.



These armies would like to agree on the time of their attack but are communicating unreliably as **any** messenger could be captured and may not deliver the message. For example, if *A1* decides to attack at 0600, it may send the message “We should attack at 0600 on June 28” to *A2* but if the messenger gets captured, *A2* will not attack at that time and *A1* will be defeated.0

In order to solve this problem, suppose we require two confirmation messages. That is, *A1* will send a time to attack and wait for a confirmation from *A2*. In order to make sure that *A1* received the attack message, *A2* will wait for a confirmation about its agreement to the given time:



That is, *A1* will attack if it receives a confirmation about the proposed time and *A2* will attack if it receives a confirmation message about its agreement message.

Using this messaging technique, are the two armies guaranteed to attack at the same time? If not, which army might attack alone?

Part III - Concurrency Questions

Question 1: ***It's a rat race (condition) out there…***
reverse reverses the order of the digits, and **right** rotates the digits to the right (the one furthest on the right moves over to the left). If NUMBER starts with a value of **456**, what are the *different, unique values* of NUMBER at the end? (select all that apply) *Note: "**set**" blocks run instantaneously and "atomically" (i.e., aren't interrupted by any other script)*

Which of the following **cannot** be the final value of NUMBER?

1. 475
2. 547
3. 647
4. 466

Question 2: If 80% of a program's code is parallel, what is the maximum speedup with ∞ helpers? (select ONE)

○	○	○	○
20x	80x	1.25x	5x

Appendix C

Fifth Grade Science Study Short Answer

In this appendix, the student responses for the open-ended survey questions in the Harpeth Hall case study are presented. There were two different prompts. The first question was *"What did you find interesting about today?"* and was only required for the students that found the lesson interesting. The second question was *"What would be an interesting way to use coding?"*

C.1 Categorized Student Responses

We have classified the student responses by responses related to the presented distributed programming abstractions and responses emphasizing positive perceived student self-efficacy. As responses may be related to both the distributed programming abstractions and student self-efficacy, a single response may appear in both lists. Unrelated responses are provided after the selected student responses.

C.1.1 Distributed Programming Abstractions

- "The way we could get maps was very interesting"
- "I liked where it showed us the maps and the temp."
- "I thought it was cool how we could control Google."
- "that to code to make a weather app it doesn't need all the complex things."
- "I found the fact that you could use network to figure out temperature in different places."
- "the way you can put codes together to make a weather app"
- "how to get the temperature in your area"
- "that you can actually find the tempature"
- "I found that you could program the sprite to different tasks like tell you the weather."
- "We got to know the temperatures of different places."

- "I found it interesting because I thought it was cool that you could make a weather app that easily."*
- "It was cool to make a weather app!"
- "that we could find the exact temp and location"

C.1.2 Perceived Self-Efficacy

- "Being able to do it myself. Plus it was super cool."
- "what I found interesting is that I can make a small app."
- "Today I found it interesting that that's all it took to make that. I thought it would take longer."
- "I thought it was way cool that this program simplified coding enough for me to understand!"
- "I found interesting is that we got to make our own app."
- "I found it interesting because I thought it was cool that you could make a weather app that easily."*
- "i found interesting on how you could make your own program"
- "I liked that we got to make something we can actually use in everyday life."

C.1.3 Miscellaneous

- "Everything" (2)
- "that we got to code an app!!!"
- "I found it interesting because I would never thought that you could do that much with just moving a few bars and typing in a few letters."
- "I like all the different categories for all the different tools"
- "i liked learning how to do coding from scratch not just doing directions."
- "that Brian was nice and he explained everything well and helped us"

- "how much I didn't know about how careful I had to be while programing so I didn't make a mistake."
- "learning how all the coding works"
- "that it would read your code in seconds"
- "I liked it all and it was really fun. I thought it was cool and I learned a lot."
- "Playing with the code/ messing with it."
- "I thought it was interesting how the command blocks could be put together to make new commands."
- "how many coding things you really need to use to just do a little thing"
- "learning to code"
- "I have coded before but I liked the site we used and what we made."
- "the sprite"
- "You really have to pay attention to do the right thing."
- "Learning how to use it."
- "I found interesting that you could have a block of codes and if you pressed a certain button it would do the code you asked it to."
- "All of the different blocks went together to form an app."
- "i found today interesting because i have never done coding before and it was interesting to learn something different."
- "It was amazing that you can do that"
- "that you have to put the blocks in a specific spot"
- "I thought about how many things you can do."
- "I liked that the coding website was so so so so so so organized in the right slots and everything."

References

[1] Multiplayer pong! *URL: https://scratch.mit.edu/projects/10089707/, accessed Oct* (2017). 2.1.4

[2] All at once (block). https://wiki.scratch.mit.edu/wiki/All_at_Once_(block). Cited 2018 February 2. 3.2.2

[3] Aslan, K., Molli, P., Skaf-Molli, H., and Weiss, S. C-set: a commutative replicated data type for semantic stores. In *RED: Fourth International Workshop on Resource Discovery* (2011). 4.1

[4] B, R. Snap4arduino. *URL: http://snap4arduino.rocks/, accessed Oct* (2017). 2.1.2, 3.1.1

[5] Bajaj, M., Zwemer, D., Peak, R., Phung, A., Scott, A. G., and Wilson, M. Slim: collaborative model-based systems engineering workspace for next-generation complex systems. In *Aerospace Conference, 2011 IEEE* (2011), IEEE, pp. 1–15. 4.1

[6] Ball, T., Burckhardt, S., de Halleux, J., Moskal, M., Protzenko, J., and Tillmann, N. Beyond open source: the touchdevelop cloud-based integrated development environment. In *Proceedings of the Second ACM International Conference on Mobile Software Engineering and Systems* (2015), IEEE Press, pp. 83–93. 3.1.2

[7] Barr, D., Harrison, J., and Conery, L. Computational thinking: A digital age skill for everyone. *Learning & Leading with Technology 38*, 6 (2011), 20–23. 6.1.1

[8] Barve, Y., Patil, P., Bhattacharjee, A., and Gokhale, A. Pads: Design and implementation of a cloud-based, immersive learning environment for distributed systems algorithms. *IEEE Transactions on Emerging Topics in Computing* (2017). (document), 2.1.3, 2.1.3, 15, 2.1.3

[9] Bau, D. Droplet, a blocks-based editor for text code. *Journal of Computing Sciences in Colleges 30*, 6 (2015), 138–144. 3.1.2

[10] Bau, D., Dawson, M., and Bau, A. Using pencil code to bridge the gap between visual and text-based coding. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), ACM, pp. 706–706. 3.1.2

[11] Baumgart, I., Heep, B., and Krause, S. Oversim: A flexible overlay network simulation framework. In *IEEE Global Internet Symposium, 2007* (2007), IEEE, pp. 79–84. 2.1.3

[12] The Beauty and Joy of Computing. http://bjc.berkeley.edu/. Cited 2016 May 14. 2.1.2

[13] Beckhusen, J. *Occupations in Information Technology.* US Department of Commerce, Economics and Statistics Administration, US Census Bureau, 2016. 1.1

[14] Begel, A. Logoblocks: A graphical programming language for interacting with the world. *Electrical Engineering and Computer Science Department, MIT, Boston, MA* (1996), 62–64. 2.1.2

[15] Bers, M. U., Flannery, L., Kazakoff, E. R., and Sullivan, A. Computational thinking and tinkering: Exploration of an early childhood robotics curriculum. *Computers & Education 72* (2014), 145–157. 6.1.1

[16] Bitbucket. *URL: http://bitbucket.com/, accessed March* (2018). 4.1

[17] Bless, R., and Doll, M. Integration of the freebsd tcp/ip-stack into the discrete event simulator omnet++. In *Proceedings of the 36th conference on Winter simulation* (2004), Winter Simulation Conference, pp. 1556–1561. 2.1.3

[18] Brennan, K., and Resnick, M. New frameworks for studying and assessing the development of computational thinking. In *Proceedings of the 2012 annual meeting of the American Educational Research Association, Vancouver, Canada* (2012), pp. 1–25. 6.1.1

[19] Brown, N. C., Altadmri, A., and Kölling, M. Frame-based editing: Combining the best of blocks and text programming. In *Learning and Teaching in Computing and Engineering (LaTICE), 2016 International Conference on* (2016), IEEE, pp. 47–53. 3.1.2

[20] Brown, Q., Mongan, W., Kusic, D., Garbarine, E., Fromm, E., and Fontecchio, A. Computer aided instruction as a vehicle for problem solving: Scratch programming environment in the middle years classroom. *Retrieved September 22* (2013). 6.1.1

[21] Brusilovsky, P., Calabrese, E., Hvorecky, J., Kouchnirenko, A., and Miller, P. Mini-languages: a way to learn programming principles. *Education and Information Technologies 2*, 1 (1997), 65–83. 6.1.1

[22] Church, L., Söderberg, E., and Elango, E. A case of computational thinking: The subtle effect of hidden dependencies on the user experience of version control. 5.1, 5.1.2

[23] Clements, D. H. Effects of logo and cai environments on cognition and creativity. *Journal of Educational Psychology 78*, 4 (1986), 309. 6.1.1

[24] Clements, D. H. Metacomponential development in a logo programming environment. *Journal of Educational Psychology 82*, 1 (1990), 141. 2.1.1

[25] Clements, D. H., and Gullo, D. F. Effects of computer programming on young children's cognition. *Journal of educational psychology 76*, 6 (1984), 1051. 6.1.1

[26] Cloud9. *URL: http://c9.io/, accessed March* (2018). 4.1

[27] Cooper, S., Dann, W., and Pausch, R. Alice: a 3-d tool for introductory programming concepts. In *Journal of Computing Sciences in Colleges* (2000), vol. 15, Consortium for Computing Sciences in Colleges, pp. 107–116. 2.1.2, 2.1.2

[28] Corley, J., Syriani, E., Ergin, H., and Van Mierlo, S. Cloud-based multi-view modeling environments. In *Modern Software Engineering Methodologies for Mobile and Cloud Environments*. IGI Global, 2016, pp. 120–139. 4.1

[29] Cox, R., Bird, S., and Meyer, B. Teaching computer science in the victorian certificate of education: A pilot study. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (2017), ACM, pp. 135–140. 2.1.2

[30] Crews, T., and Ziegler, U. The flowchart interpreter for introductory programming courses. In *Frontiers in Education Conference, 1998. FIE'98. 28th Annual* (1998), vol. 1, IEEE, pp. 307–312. 6.1.1

[31] Dann, W., and Cooper, S. Education alice 3: concrete to abstract. *Communications of the ACM 52*, 8 (2009), 27–29. 2.1.2, 2.1.2

[32] Dasgupta, S., Clements, S. M., Idlbi, A. Y., Willis-Ford, C., and Resnick, M. Extending scratch: New pathways into programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on* (2015), IEEE, pp. 165–169. 2.1.2, 2.1.4, 2.1.4

[33] Dean, J., and Ghemawat, S. Mapreduce: simplified data processing on large clusters. *Communications of the ACM 51*, 1 (2008), 107–113. 2.4.2

[34] Denner, J., Werner, L., and Ortiz, E. Computer games created by middle school girls: Can they be used to measure understanding of computer science concepts? *Computers & Education 58*, 1 (2012), 240–249. 6.1.1

[35] Drawp for school. *URL: https://www.commonsense.org/education/app/drawp-for-school-create-learn-collaborate, accessed March* (2018). 4.1

[36] Drescher, G. L. Object-oriented logo'. In *Artificial Intelligence and Education* (1987), vol. 1, pp. 153–165. 2.1.1

[37] Ellis, C. A., and Gibbs, S. J. Concurrency control in groupware systems. In *ACM Sigmod Record* (1989), vol. 18, ACM, pp. 399–407. 4.1

[38] emfCollab: Collaborative Editing for EMF models. http://qgears.com/products/emfcollab/. Cited 2017 October 4. (document), 4.1, 68, 4.1, 4.2.1, 4.3

[39] Federici, S. A minimal, extensible, drag-and-drop implementation of the c programming language. In *Proceedings of the 2011 conference on Information technology education* (2011), ACM, pp. 191–196. 3.1.2

[40] Feng, A., Gardner, M., and Feng, W.-c. Parallel programming with pictures is a snap! *Journal of Parallel and Distributed Computing 105* (2017), 150–162. 2.1.2

[41] Fisch, R. Moenagade - the mouse enabled game development tool. http://moenagade.fisch.lu/, 2017. [Online; accessed 20-September-2017]. 3.1.2

[42] Fraser, N. Differential synchronization. In *Proceedings of the 9th ACM symposium on Document engineering* (2009), ACM, pp. 13–20. (document), 4.1, 4.1, 67

[43] Fraser, N., et al. Blockly: A library for building visual programming editors. *URL: https://developers.google.com/blockly* (2018). 3.1.1

[44] Github. *URL: http://github.com/, accessed March* (2018). 4.1

[45] Gitlab. *URL: http://gitlab.com/, accessed March* (2018). 4.1

[46] Google docs. *URL: http://docs.google.com/, accessed March* (2018). 4.1, 4.2.1, 5.1, 5.1.2, 5.2.1

[47] Gorman, J., Gsell, S., and Mayfield, C. Learning relational algebra by snapping blocks. In *Proceedings of the 45th ACM technical symposium on Computer science education* (2014), ACM, pp. 73–78. 2.1.2

[48] Guzdial, M. Education paving the way for computational thinking. *Communications of the ACM 51*, 8 (2008), 25–27. 6.1.1

[49] Haaranen, L., and Lehtinen, T. Teaching git on the side: Version control system as a course platform. In *Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education* (2015), ACM, pp. 87–92. 5.1.1

[50] Han, A., Kim, J., and Wohn, K. Entry: visual programming to enhance children's computational thinking. In *Adjunct Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2015 ACM International Symposium on Wearable Computers* (2015), ACM, pp. 73–76. 3.1.2

[51] Hanning, K. Scratch iss tracker extension. *URL: https://khanning.github.io/scratch-isstracker-extension/, accessed Oct* (2017). 2.1.4, 2.1.4, 2.1.4

[52] Hanning, K. Scratch weather extension. *URL: https://khanning.github.io/scratch-weather-extension/, accessed Oct* (2017). 2.1.4, 2.1.4

[53] Hechmer, A., Tindale, A., and Tzanetakis, G. Logorhythms: Introductory audio programming for computer musicians in a functional language paradigm. In *Frontiers in Education Conference, 36th Annual* (2006), IEEE, pp. 11–16. 2.1.1

[54] Hohmann, L., Guzdial, M., and Soloway, E. Soda: A computer-aided design environment for the doing and learning of software design. In *Computer Assisted Learning*, I. Tomek, Ed., vol. 602 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 1992, pp. 307–319. 1.2

[55] Howell, F., and McNab, R. Simjava: A discrete event simulation library for java. *Simulation Series 30* (1998), 51–56. 2.1.3

[56] Isomöttönen, V., and Cochez, M. Challenges and confusions in learning version control with git. In *International Conference on Information and Communication Technologies in Education, Research, and Industrial Applications* (2014), Springer, pp. 178–193. 5.1, 5.1.2

[57] Kalelioğlu, F. A new way of teaching programming skills to k-12 students: Code.org. *Computers in Human Behavior 52* (2015), 200–210. 3.1.2

[58] Kalelioğlu, F. A new way of teaching programming skills to k-12 students: Code.org. *Computers in Human Behavior 52* (2015), 200–210. 6.1.1

[59] Kay, A. Squeak etoys, children & learning. *online article 2006* (2005). 2.1.1, 2.1.2

[60] Kelleher, C., Cosgrove, D., Culyba, D., Forlines, C., Pratt, J., and Pausch, R. Alice2: programming without syntax errors. In *User Interface Software and Technology* (2002). 2.1.2

[61] Kelleher, C., Pausch, R., and Kiesler, S. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (2007), ACM, pp. 1455–1464. 6.1.1

[62] Keren, G., and Fridin, M. Kindergarten social assistive robot (kindsar) for childrens geometric thinking and metacognitive development in preschool education: A pilot study. *Computers in Human Behavior 35* (2014), 400–412. 6.1.1

[63] Klahr, D., and Carver, S. M. Cognitive objectives in a logo debugging curriculum: Instruction, learning, and transfer. *Cognitive Psychology 20*, 3 (1988), 362–404. 2.1.1

[64] Kölling, M. The greenfoot programming environment. *ACM Transactions on Computing Education (TOCE) 10*, 4 (2010), 14. 2.1.2

[65] Kölling, M., Brown, N. C., and Altadmri, A. Frame-based editing: Easing the transition from blocks to text-based programming. In *Proceedings of the Workshop in Primary and Secondary Computing Education* (2015), ACM, pp. 29–38. 3.1.2

[66] Koschitz, D., and Rosenbaum, E. Exploring algorithmic geometry with beetle blocks: a graphical programming language for generating 3d forms. In *Proceedings of the 15 th International Conference on Geometry and Graphics* (2012), pp. 380–389. 2.1.2

[67] Kreiser, S. Marrying gui and git: how authoring tools for e-learning can benefit from version control. Master's thesis, University of Twente, 2016. 5.1, 5.1.2

[68] Kuenzi, J. J. Science, technology, engineering, and mathematics (stem) education: Background, federal policy, and legislative action. 1.1

[69] Kumawat, S., and Khunteta, A. A survey on operational transformation algorithms: Challenges, issues and achievements. *International Journal of Computer Applications 3*, 12 (2010), 3038. 4.1

[70] Kyfonidis, C., Moumoutzis, N., and Christodoulakis, S. Block-c: A block-based visual environment for supporting the teaching of c programming language to novices. 3.1.2

[71] Lawrance, J., Jung, S., and Wiseman, C. Git on the cloud in the classroom. In *Proceeding of the 44th ACM technical symposium on Computer science education* (2013), ACM, pp. 639–644. 5.1.1

[72] Letia, M., Preguiça, N., and Shapiro, M. Crdts: Consistency without concurrency control. *arXiv preprint arXiv:0907.0929* (2009). 4.1, 4.2.2

[73] Liang, S. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Professional, 1999. 3.2.1

[74] Lv, X., He, F., Cai, W., and Cheng, Y. An efficient collaborative editing algorithm supporting string-based operations. In *Computer Supported Cooperative Work in Design (CSCWD), 2016 IEEE 20th International Conference on* (2016), IEEE, pp. 45–50. 4.2.2

[75] MacLaurin, M. B. The design of kodu: A tiny visual programming language for children on the xbox 360. In *ACM Sigplan Notices* (2011), vol. 46, ACM, pp. 241–246. 2.1.1, 2.1.2, 2.1.2, 2.1.2

[76] Malan, D. J., and Leitner, H. H. Scratch for budding computer scientists. *ACM SIGCSE Bulletin 39*, 1 (2007), 223–227. 3.2.2, 4.2.1, 6.1.1

[77] Maloney, J. Gp. *URL: https://gpblocks.org* (2018). 3.1.1

[78] Maloney, J., Resnick, M., Rusk, N., Silverman, B., and Eastmond, E. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE) 10*, 4 (2010), 16. 2.1.1, 2.1.2, 2.1.2, 2.1.4, 5.2

[79] Maloney, J. H., Peppler, K., Kafai, Y., Resnick, M., and Rusk, N. *Programming by choice: urban youth learning programming with scratch*, vol. 40. ACM, 2008. 5.2, 6.1.1

[80] Maloney, J. H., and Smith, R. B. Directness and liveness in the morphic user interface construction environment. In *Proceedings of the 8th annual ACM symposium on User interface and software technology* (1995), ACM, pp. 21–28. 4.2.1

[81] Maróti, M., Kecskés, T., Kereskényi, R., Broll, B., Völgyesi, P., Jurácz, L., Levendovszky, T., and Lédeczi, Á. Next generation (meta) modeling: Web-and cloud-based collaborative tool infrastructure. *MPM@ MoDELS 1237* (2014), 41–60. 4.1, 4.2.1, 5.1, 5.1.2

[82] Martin, F., and Resnick, M. Lego/logo and electronic bricks: Creating a science-land for children. In *Advanced educational technologies for mathematics and science*. Springer, 1993, pp. 61–89. 2.1.1, 2.1.2

[83] Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. Learning computer science concepts with scratch. *Computer Science Education 23*, 3 (2013), 239–264. 1.1, 6.1.1

[84] Meerbaum-Salant, O., Armoni, M., and Ben-Ari, M. Learning computer science concepts with scratch. *Computer Science Education 23*, 3 (2013), 239–264. 3.2.2

[85] Milentijevic, I., Ciric, V., and Vojinovic, O. Version control in project-based learning. *Computers & Education 50*, 4 (2008), 1331–1338. 5.1

[86] Minecraft. *URL: http://minecraft.net/, accessed March* (2018). 2.1.2

[87] Moenig, J., and Harvey, B. Snap! *URL: http://snap.berkeley.edu/, accessed Aug* (2017). 2.1.1, 2.1.2, 2.1.2, 2.1.4, 3.1.1

[88] Monig, J., Ohshima, Y., and Maloney, J. Blocks at your fingertips: Blurring the line between blocks and text in gp. In *Blocks and Beyond Workshop (Blocks and Beyond), 2015 IEEE* (2015), IEEE, pp. 51–53. 3.1.2

[89] Morelli, R., De Lanerolle, T., Lake, P., Limardo, N., Tamotsu, E., and Uche, C. Can android app inventor bring computational thinking to k-12. In *Proc. 42nd ACM technical symposium on Computer science education (SIGCSE'11)* (2011), pp. 1–6. 6.1.1

[90] Moreno-León, J., Robles, G., and Román-González, M. Dr. scratch: Automatic analysis of scratch projects to assess and foster computational thinking. *RED. Revista de Educación a Distancia*, 46 (2015), 1–23. 6.1.1

[91] Moro, M. Object oriented programming and development in jxlogo. In *Proceedings of the 10th European Logo conference. Edited by Gregorczyk G. and oth., Warsaw* (2005), pp. 132–143. 2.1.1

[92] Nebraska. *URL: http://wiki.squeak.org/squeak/1356, accessed Oct* (2017). 4.1, 4.2.1

[93] Noss, R. Constructing a conceptual framework for elementary algebra through logo programming. *Educational Studies in Mathematics 17*, 4 (1986), 335–357. 2.1.1

[94] OpenWeatherMap: Weather API. http://openweathermap.org/api/. Cited 2017 Nov 8. 2.1.4, 2.1.4

[95] Oshiba, T., and Tanaka, J. 3d-pp: Visual programming system with three-dimensional representation. In *Proceedings of International Symposium on Future Software Technology* (1999), pp. 61–66. 2.1.2

[96] Oswald, H., Allen, J., and Gough, B. Sharelatex, the online latex editor. 5.1, 5.1.2

[97] Ozik, J., Collier, N. T., Murphy, J. T., and North, M. J. The relogo agent-based modeling language. In *Simulation Conference (WSC), 2013 Winter* (2013), IEEE, pp. 1560–1568. 2.1.1

[98] Papert, S. *Mindstorms: Children, computers, and powerful ideas*. Basic Books, Inc., 1980. 6.1.1

[99] Pea, R. D. Logo programming and problem solving. 2.1.1

[100] Pea, R. D., and Kurland, D. M. Logo programming and the development of planning skills. technical report no. 16. 6.1.1

[101] Perez De Rosso, S., and Jackson, D. What's wrong with git?: a conceptual design analysis. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software* (2013), ACM, pp. 37–52. 5.1, 5.1.2

[102] Perkins, D. N., and Simmons, R. Patterns of misunderstanding: An integrative model for science, math, and programming. *Review of Educational Research 58*, 3 (1988), pp. 303–326. 1.2

[103] Pokress, S. C., and Veiga, J. J. D. MIT app inventor: Enabling personal mobile computing. *arXiv preprint arXiv:1310.2830* (2013). (document), 2.1.2, 12, 13

[104] Pong (cloud data). *URL: https://scratch.mit.edu/projects/148536151/, accessed Oct* (2017). 2.1.4

[105] Qualls, J. A., and Sherrell, L. B. Why computational thinking should be integrated into the curriculum. *Journal of Computing Sciences in Colleges 25*, 5 (2010), 66–71. 6.1.1

[106] Reid, K. L., and Wilson, G. V. Learning by doing: introducing version control as a way to manage student assignments. In *ACM SIGCSE Bulletin* (2005), vol. 37, ACM, pp. 272–276. 5.1, 5.1.1

[107] Resnick, M. Starlogo: An environment for decentralized modeling and decentralized thinking. In *Conference companion on Human factors in computing systems* (1996), ACM, pp. 11–12. (document), 2.1.1, 2.1.1, 2.1.1, 1, 2.1.2

[108] Richards, E., and Terkanian, D. Occupational employment projections to 2022. *Monthly Lab. Rev. 136* (2013), 1. 1.1

[109] Riviere, E. Simplifying hands-on teaching of distributed algorithms with splay. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International* (2012), IEEE, pp. 1311–1316. (document), 2.1.3, 2.1.3, 16

[110] Rizvi, M., Humphries, T., Major, D., Jones, M., and Lauzun, H. A cs0 course using scratch. *Journal of Computing Sciences in Colleges 26*, 3 (2011), 19–27. 6.1.1

[111] Rocco, D., and Lloyd, W. Distributed version control in the classroom. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (2011), ACM, pp. 637–642. 5.1

[112] Romagosa Carrasquer, B. From the turtle to the beetle. Master's thesis, Universitat Oberta de Catalunya, 2016. (document), 7

[113] Rosso, D., Perez, S., et al. Purposes, concepts, misfits, and a redesign of git. In *ACM SIGPLAN Notices* (2016), vol. 51, ACM, pp. 292–310. 5.1.2

[114] Sabitzer, B., Antonitsch, P. K., and Pasterk, S. Informatics concepts for primary education: preparing children for computational thinking. In *Proceedings of the 9th Workshop in Primary and Secondary Computing Education* (2014), ACM, pp. 108–111. 1.1, 6.1.1

[115] Schreiner, W. A java toolkit for teaching distributed algorithms. In *ACM SIGCSE Bulletin* (2002), vol. 34, ACM, pp. 111–115. (document), 2.1.3, 2.1.3, 2.1.3, 17

[116] Scratch statistics - imagine, program, share. *URL: https://scratch.mit.edu/statistics, accessed Oct* (2017). 2.1.2

[117] Screenhero. *URL: http://screenhero.com/, accessed March* (2018). 4.1

[118] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Conflict-free replicated data types. In *Symposium on Self-Stabilizing Systems* (2011), Springer, pp. 386–400. 4.1

[119] Shapiro, M., Preguiça, N., Baquero, C., and Zawirski, M. Convergent and commutative replicated data types. *Bulletin-European Association for Theoretical Computer Science*, 104 (2011), 67–88. 4.1, 4.1

[120] Silva, Y. N., and Chon, J. Dbsnap: Learning database queries by snapping blocks. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (2015), ACM, pp. 179–184. 2.1.2

[121] Singer, L., and Schneider, K. It was a bit of a race: Gamification of version control. In *Games and Software Engineering (GAS), 2012 2nd International Workshop on* (2012), IEEE, pp. 5–8. 5.1, 5.1.1

[122] Sketchware - IDE in your pocket. *URL: http://sketchware.io/, accessed October* (2017). 2.1.2

[123] Slany, W. Tinkering with pocket code, a scratch-like programming app for your smartphone. *Proc. of Constructionism* (2014). 2.1.2

[124] Smith, V. A., and Duncan, I. Biology students building computer simulations using starlogo tng. *Bioscience Education 18*, 1 (2011), 1–9. 1.1

[125] Spohrer, J. C., and Soloway, E. Novice mistakes: are the folk wisdoms correct? *Commun. ACM 29*, 7 (July 1986), 624–632. 1.2

[126] School for Science and Math at Vanderbilt. http://theschool.vanderbilt.edu/. Cited 2018 February 2. 6.3.1

[127] Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., and Ergin, H. Atompm: A web-based modeling environment. In *Joint proceedings of MODELS'13 Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th International Conference on Model Driven Engineering Languages and Systems (MODELS 2013): September 29-October 4, 2013, Miami, USA* (2013), pp. 21–25. 4.1

[128] Theunis, J., Van den Broeck, B., Leys, P., Potemans, J., Van Lil, E., Van de Capelle, A., et al. Opnet in advanced networking education. In *the Proceedings to the International Conference on Networking ICN* (2002), vol. 1. 2.1.3

[129] Thum, C., Schwind, M., and Schader, M. Slima lightweight environment for synchronous collaborative modeling. *Model Driven Engineering Languages and Systems* (2009), 137–151. 4.1, 4.2.1

[130] Tisue, S., and Wilensky, U. Netlogo: A simple environment for modeling complexity. In *International conference on complex systems* (2004), vol. 21, Boston, MA, pp. 16–21. 2.1.1, 2.1.1

[131] Trinket: Code is Your Canvas. https://trinket.io/. Cited 2018 March 1. 3.1.2

[132] Varga, A. Using the omnet++ discrete event simulation system in education. *IEEE Transactions on Education 42*, 4 (1999), 11–pp. 2.1.3

[133] Varga, A., and Hornig, R. An overview of the omnet++ simulation environment. In *Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops* (2008), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), p. 60. (document), 2.1.3, 14

[134] Wang, E. Teaching freshmen design, creativity and programming with legos and labview. In *Frontiers in Education Conference, 2001. 31st Annual* (2001), vol. 3, IEEE, pp. F3G–11. 6.1.1

[135] Wang, K., McCaffrey, C., Wendel, D., and Klopfer, E. 3d game design with programming blocks in starlogo tng. In *Proceedings of the 7th international conference on Learning sciences* (2006), International Society of the Learning Sciences, pp. 1008–1009. 1.1, 2.1.1, 2.1.2

[136] Weber, D., Glaser, J., and Mahlknecht, S. Discrete event simulation framework for power aware wireless sensor networks. In *Industrial Informatics, 2007 5th IEEE International Conference on* (2007), vol. 1, IEEE, pp. 335–340. 2.1.3

[137] Weintrop, D., and Holbert, N. From blocks to text and back: Programming patterns in a dual-modality environment. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education* (2017), ACM, pp. 633–638. 3.1.2

[138] Weiss, S., Urso, P., and Molli, P. Logoot-undo: Distributed collaborative editing system on p2p networks. *IEEE Transactions on Parallel and Distributed Systems 21*, 8 (2010), 1162–1174. 4.1

[139] Wilson, A., Hainey, T., and Connolly, T. M. Using scratch with primary school children: an evaluation of games constructed to gauge understanding of programming concepts. *International Journal of Game-Based Learning (IJGBL) 3*, 1 (2013), 93–109. 1.1, 6.1.1

[140] Wing, J. Computational thinking benefits society. *40th Anniversary Blog of Social Issues in Computing 2014* (2014). 6.1.1

[141] Wing, J. M. Computational thinking. *Communications of the ACM 49*, 3 (2006), 33–35. 6.1.1

[142] Wolber, D. App inventor and real-world motivation. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (2011), ACM, pp. 601–606. 6.1.1

[143] Yu, W. A string-wise crdt for group editing. In *Proceedings of the 17th ACM international conference on Supporting group work* (2012), ACM, pp. 141–144. 4.1

[144] Zorn, C., Wingrave, C. A., Charbonneau, E., and LaViola Jr, J. J. Exploring minecraft as a conduit for increasing interest in programming. In *FDG* (2013), pp. 352–359. (document), 2.1.2, 10, 2.1.2