Sikuli Blocks

Making Computer Automation Tools More Learnable

Adam Leonard ⁕ 6.UAP ⁕ May 17, 2012

Supervisor: Professor Rob Miller

1.0 Introduction

Sikuli Blocks is a block-based environment that allows users without a programming background to easily automate arduous tasks on their computers using screenshots.

Sikuli Blocks is built on top of Sikuli Script [1], a Python-based automation framework that can interact with GUI elements identified by screenshots of that element. Using Sikuli Script, a user can, for example, take a screenshot of a checkbox and use a Python `for` loop to click on all checkboxes in a window. Sikuli Script is powerful; it enables automated keyboard and mouse interactions with almost anything the user can see on screen. Thus, people have used it for a wide variety of applications, from performing regression tests on complex software applications [2] to playing perfect games of Angry Birds [3].

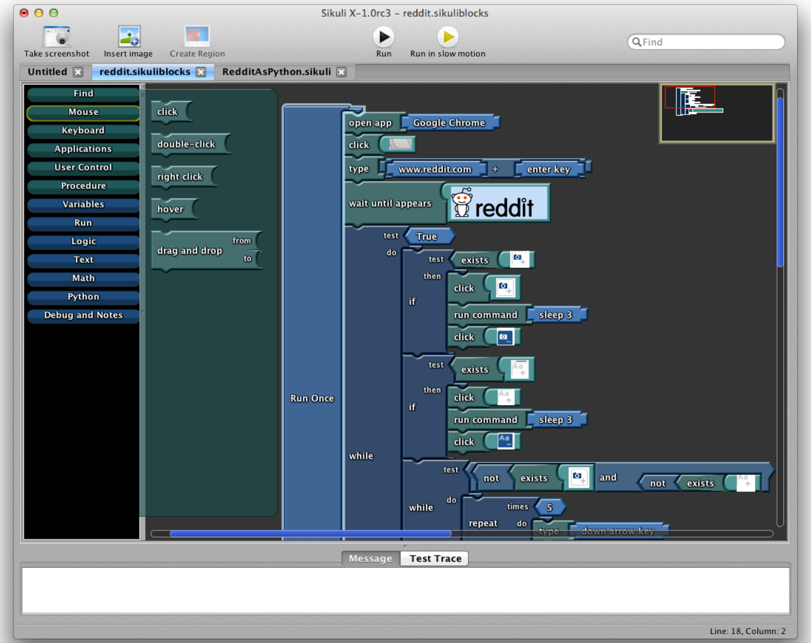


Figure 1: The Sikuli Blocks interface with a user-created block program that automatically surfs reddit.

However, to expose the full power of Sikuli Script, users must know how to program in Python. Users who have some programming experience, but lack experience in Python, need to learn Python syntax and the Python and Sikuli library commands before they can write interesting Sikuli scripts. Users without any programming experience generally cannot write even the most basic Sikuli scripts. Thus, Sikuli is currently restricted to programmers and advanced computer users who are willing to devote a lot of time to learn the tool.

We believe that a large number of intermediate computer users who are unfamiliar with programming would benefit from screenshot-based automation. Indeed, we are not aware of any automation tools that are both easy enough for non-programmers to learn and powerful enough to automate a wide range of tasks in *any* application.

Sikuli Blocks seeks to fill this need. It offers a simple block-based programming environment where, rather than typing Python commands, users can choose from a list of “blocks” that represent commands and connect them together like puzzle pieces. Using visual blocks without any code, users have access to almost all of the features Sikuli offers, including the ability to easily add screenshots, to automate keyboard and mouse interactions with GUI elements represented by those screenshots, and to launch and quit applications. It also exposes many common Python features through blocks, including support for variables, logic, math, conditionals, loops, and functions. If needed, users can inline custom Python code into a block program, or convert the entire block program to an editable Python program. This visual programming approach eliminates the need to write raw Python code for most automation tasks. Thus, we have found that Sikuli Blocks enables non-programmers to accomplish complex automation tasks without spending a lot of time learning the tool; users who faced considerable frustration accomplishing an automation task using Sikuli Script were successful when they switched to Sikuli Blocks.

This report is divided into seven sections. Section 2 analyzes the learnability of existing automation tools, including Sikuli Script. Section 3 introduces the design goals for Sikuli Blocks and describes how the block-based design addresses the learnability issues present in other tools. Section 4 describes the features of Sikuli Blocks in detail and how those features satisfy our design goals. Section 5 summarizes how we implemented Sikuli Blocks. Section 6 presents and analyzes the results of usability tests on Sikuli Blocks. Finally, section 7 presents ideas for future work.

2.0 Learnability Analysis of Existing Automation Tools

The idea of creating tools that allow non-programmers to automate their computers is not new. Dozens of such tools have been developed in the past twenty years, and they have achieved varying degrees of success. In particular, they have all had to make a fundamental tradeoff between the flexibility the tool exposes to the user and the degree of learnability it offers to the non-programmer.

Others have written about the features and implementations of automation tools such as AppleScript, Windows Scripting, DocWizards, Jitbit, and Co-Scripter [1, 4]. Rather than repeating the discussion of the features of these various tools, this section focuses on analyzing the properties that are most significant for this project: learnability and flexibility for non-programmers. Three tools are particularly interesting to study with respect to these properties: Applescript, a classic text-based automation language, Automator, a visual block-based language, and Sikuli Script, a modern text and screenshot based language.

2.1 AppleScript

Applescript is a text-based automation language released by Apple Computer in 1993. It was designed and marketed specifically at power users who had minimal programming experience, and it could automate any application that supported the underlying cross-application communication API, Apple Events. The AppleScript development team believed that non-programmers could easily learn a language that resembled English, so they based the language around nouns, adjectives, and verbs [5].

For example, to skip to the next song in a user's iTunes library, a user could enter the following AppleScript into the included IDE and click the "Run" button:

```
tell application "iTunes" to play (next track)
```

As is typical for most AppleScripts, this script is very easy for non-programmers to read and understand; most users can immediately guess what it does. In a C-based object-oriented automation language, this code might look like:

```
iTunes app = System.getApplication("iTunes");
app.play(app.getNextTrack());
```

which is far harder for non-programmers to understand.

While Applescript succeeded in being a very *readable* language since it looks like natural language, the fact that it does not employ true natural language processing means non-programmers find it difficult to write their own AppleScript or modify an existing Applescript. For example, the following minor modifications to the above iTunes AppleScript result in errors:

```
tell application iTunes to play (next track)
tell application "iTunes" to play (next song)
tell "iTunes" to play (next track)
play (next track) in application "iTunes"
tell application "iTunes" to skip to the next track
```

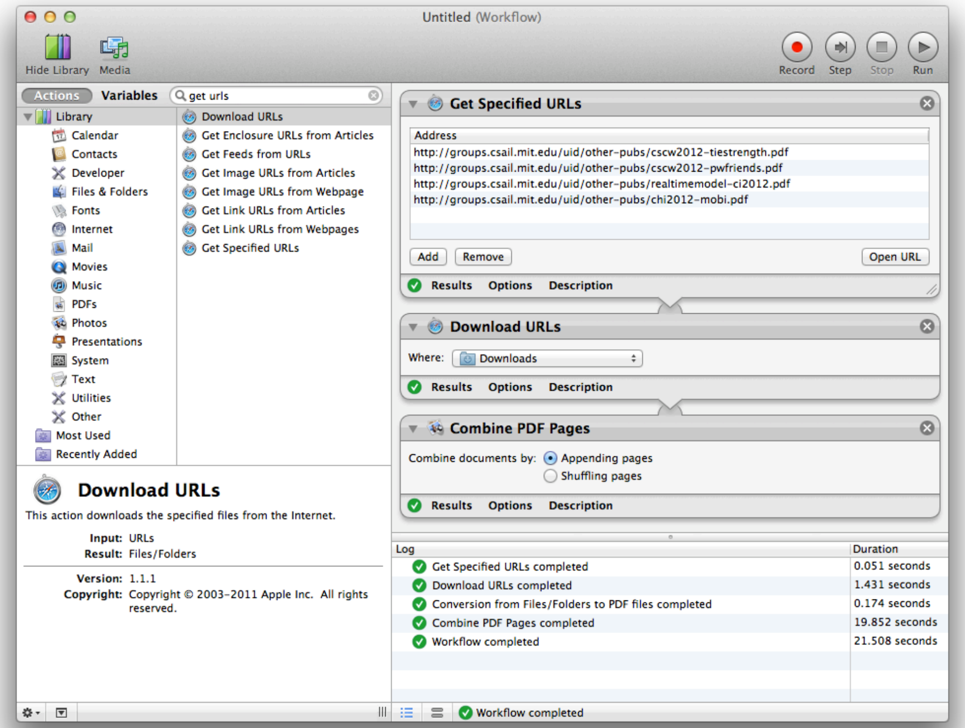
Indeed, William Cook, one of the original developers of AppleScript, acknowledged this learnability flaw in [5]:

> The main problem is that AppleScript only appears to be a natural language. In fact is an artificial language, like any other programming language … Even small changes to the script may introduce subtle syntactic errors which baffle users. It is easy to read AppleScript, but quite hard to write it.

This flaw suggests that AppleScript was not successful in achieving its goal of allowing non-programmers to easily write computer automation scripts.

2.2 Automator

Automator is a visual block-based automation language designed by Apple as a modern alternative to AppleScript. Like AppleScript, Automator is targeted at users without a programming background. However, Automator makes a very different set of tradeoffs with respect to the flexibility vs. learnability balance. Ultimately, Automator offers far less flexibility than AppleScript, but it is much easier to learn.



Get Specified URLs

Address
http://groups.csail.mit.edu/uid/other-pubs/cscw2012-tiestrength.pdf
http://groups.csail.mit.edu/uid/other-pubs/cscw2012-pwfriends.pdf
http://groups.csail.mit.edu/uid/other-pubs/realtimemodel-ci2012.pdf
http://groups.csail.mit.edu/uid/other-pubs/chi2012-mobi.pdf

Download URLs

Where: Downloads

Combine PDF Pages

Combine documents by: [✓ Appending pages] Shuffling pages

Download URLs

This action downloads the specified files from the Internet.

Input: URLs
Result: Files/Folders

Version: 1.1.1
Copyright: Copyright © 2003-2011 Apple Inc. All rights reserved.

Log	Duration
✓ Get Specified URLs completed	0.051 seconds
✓ Download URLs completed	1.431 seconds
✓ Conversion from Files/Folders to PDF files completed	0.174 seconds
✓ Combine PDF Pages completed	19.852 seconds
✓ Workflow completed	21.508 seconds

✓ Workflow completed

Figure 2: The Automator interface with a program to download PDF files and combine them into a single document.

Rather than building a general purpose automation tool, Automator constrains users to automating a relatively small set of high-level, common tasks, such as “Combine PDF Pages,” or “Create Thumbnail Images.” Each task is displayed as a specialized “block” that can be connected to other blocks by dragging it into a “workflow.” By feeding the output of one block into the input of another, users can construct multi-step automation workflows without writing code, as in figure 2.

This construction makes Automator very easy to learn. First, all available blocks are displayed on a panel on the side, so unlike text-based languages, users do not need to memorize commands or syntax. Second, since blocks can be dragged into place as if the user were creating a recipe, Automator workflows are both easy to construct and easy to read. Third, since blocks are self-contained chunks of code, it is very difficult to make a syntax error in Automator.

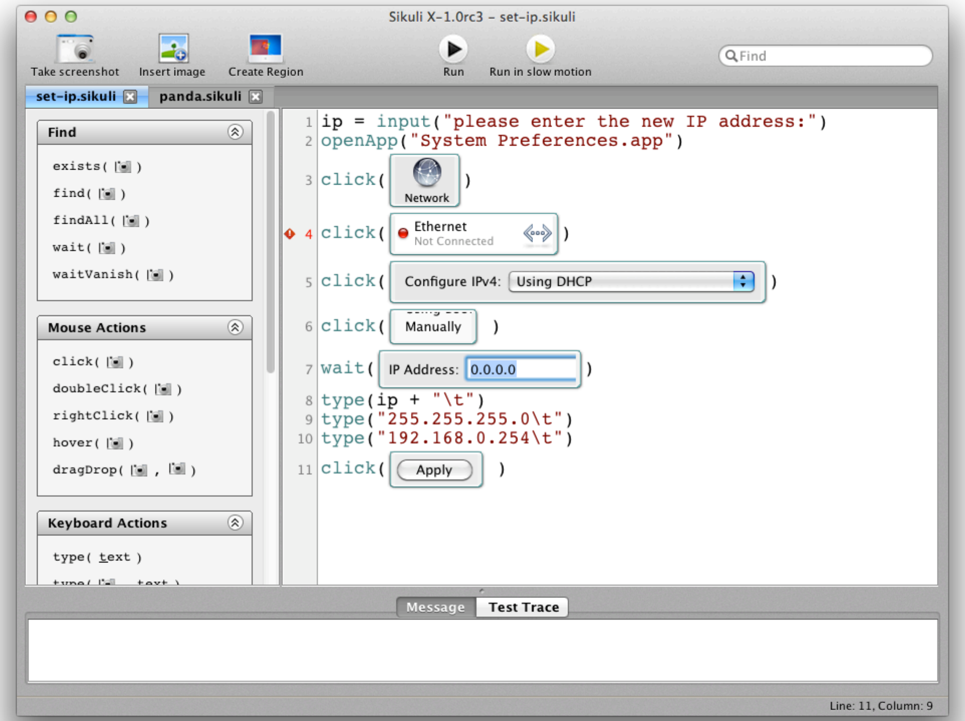
However, since Automator works at a very high-level, many automation tasks are impossible to accomplish. For example, unlike AppleScript, Automator does not support user-defined conditionals, loops, and functions. Even simple mathematical operations, such as addition, cannot be expressed in Automator since it does not offer math blocks. Further, since Automator offers a limited set of blocks, users often want to accomplish tasks that do not exist as blocks. For example, while Automator offers some blocks to control iTunes, such as “Pause iTunes,” it does not ship with a “Play Next Track” block.

To mitigate these problems, Automator offers “escape hatches” that advanced users can use to extend the functionality of Automator workflows. First, Automator offers “Run AppleScript” and “Run Shell Script” blocks that present a text box where custom code can be injected into a workflow. Additionally, programmers can build custom blocks using AppleScript or Objective-C that can be displayed alongside the default set of blocks. These escape hatches are effective for programmers, but by reverting back to a text-based design, they generally do not help non-programmers. Thus, while some automation tasks can be easily completed by non-programmers in Automator, others are impossible to accomplish without learning traditional programming techniques.

2.3 Sikuli Script

Sikuli Script is the text and screenshot based language that was discussed in the introduction. AppleScript and Sikuli Script make a similar set of tradeoffs; Sikuli Script is designed to be very flexible and powerful, but generally does not expose that power to users without a programming background. Specifically, like AppleScript, Sikuli Script can automate a wide range of tasks, and since it is built on top of Python, it offers standard programming language constructs, including variables, conditionals, loops, and functions. Also like AppleScript, simple Sikuli Scripts are fairly easy to read by non-programmers; actions are performed on screenshots so it is easy to visually follow what the script is interacting with based on what screenshots are used.

The IDE that ships with Sikuli Script takes measures that make writing basic Sikuli scripts easier for new users. It offers a sidebar the contains a list of common commands including `click`, `doubleClick`, `type`, etc. When users select one of these commands, it automatically appears in the code editor. From there, users can click on the screenshot icon in the code editor to insert a screenshot of the element with which they want to interact. This feature partially addresses one of the major learnability issues with text-based languages, namely that available commands are not visible, so they must be memorized. This was apparent in the AppleScript example above; the GUI provided no indication that to refer to an item in iTunes, the user should use the word “track” rather than, for example, the word “song.”



```
ip = input("please enter the new IP address:")
openApp("System Preferences.app")
click( Network )
click( Ethernet Not Connected )
click( Configure IPv4: Using DHCP )
click( Manually )
wait( IP Address: 0.0.0.0 )
type(ip + "\t")
type("255.255.255.0\t")
type("192.168.0.254\t")
click( Apply )
```

Figure 3: The Sikuli Script IDE interface with a program to set the user's IP address. The command sidebar is on the left.

However, even when the command sidebar is used, the user must interact with the raw Python code editor, where it is easy to make syntax mistakes. For example, clicking the "type" command in the sidebar inserts `type(text)` into the code editor, and highlights the word `text`. If the user simply starts typing `type(Hello World)` a syntax error is generated at runtime because Python requires quotation marks around strings. Similar syntax errors occur with the command sidebar if the user ignores whitespace, forgets closing parenthesis, etc.

Additionally, the command sidebar only allows users to build simple, linear scripts that only use eighteen of the many included Sikuli commands. It does not help the user enter other Sikuli commands, Python library functions, mathematical operators, variables, conditionals, loops, functions, etc. A subset of those features are generally required to build interesting Sikuli Scripts, but they require knowledge of Python to use. Thus, only the most basic parts of Sikuli Script are learnable by non-programmers; most of the interesting features of SIkuli Script are only accessible to more advanced users.

3.0 Block-Based Automation and Design Goals

The goal of Sikuli Blocks was to create an automation tool that achieves a better balance between flexibility and learnability than existing tools. Specifically, Sikuli Blocks was designed to leverage the wide range of features that Sikuli Script offers, while making it nearly as easy to use as Automator. Through this balance, non-programmers can write complex, interesting automation scripts.

3.1 Introduction to Block-Based Programming

The history of automation tools suggest that it is very difficult to build a text-based automation language that is learnable by users without programming experience. This was particularly evident in AppleScript. However, block-based programming offers a visual alternative where small units of work, called "blocks," are displayed in a visual catalog. Blocks can be dragged out onto a workspace, where they can be arranged and connected together to build a program.

Visual, block-based programming tools are substantially more learnable than text-based programming languages for many reasons, including:

- Users do not need to memorize commands. All commands are accessible visually in the GUI.
- Graphical constraints and feedback can be used to prevent users from making syntactical errors, rather than simply reporting the syntactical error after the fact.
- Blocks offer stronger real-world metaphors than text. For example, blocks look like puzzle pieces, which allows users to understand what blocks can and cannot fit together.

Automator is an excellent example of a block-based language that leverages these properties to create a tool that is very easy to learn by non-programmers. However, Automator's lack of flexibility is not representative of all block-based languages.

Scratch [7], a research project from the MIT Media Lab, uses one of the most comprehensive block based languages, yet it is targeted at children. It allows users to create games without writing code. These games vary widely, which speaks to the flexibility of the language. Its blocks perform small units of work. For example, it offers blocks that move a character a particular number of steps, and blocks that perform basic mathematical operations. This contrasts with Automator's approach, which uses blocks that perform high-level tasks that cannot be broken apart. Additionally, Scratch offers blocks that implement common programming constructs, including variables, conditionals, and loops. As a pure block-based language that leverages the advantages of visual programming, Scratch's advanced features remain learnable by non-programmers.

Google's App Inventor [8] is another example of a block-based language that makes complex programming concepts accessible to users who are unfamiliar with programming; it enables users to build a wide range of full-featured Android applications.

3.2 Block-Based programming in Sikuli Blocks

Sikuli Blocks takes Scratch's approach to block-based programming, rather than Automator's approach. Indeed, Sikuli Blocks uses the same code foundation as Scratch with an open-source offshoot project called Open Blocks [6], also developed in the MIT Media Lab. Open Blocks turns out to be an ideal platform for Sikuli, since it can model most of the common language features Python offers, and thus can be used to implement most of Sikuli's features.

3.3 Design Goals

To achieve a high degree of learnability while maintaining most of the functionality available in the text-based Sikuli Script language, Sikuli Blocks was designed around the following principles:

- Users without a programming background should be able to quickly build automation scripts without spending a lot of time experimenting or reading documentation.
- Sikuli Blocks should offer as much functionality and flexibility as possible without sacrificing learnability. It should be easy to build programs that are nonlinear and interactive in nature.
- Blocks should perform small units of work so they can be used together in an interesting manner. Complex and static Automator-style blocks should be avoided.
- Sikuli Blocks should follow the spirit of Sikuli Script by using screenshots to specify elements with which the program will interact.
- Sikuli Blocks should offer "escape hatches" so functionality available in Sikuli Script, but not yet available with blocks, can be used inline in a block program.
- Sikuli Blocks code should be directly convertible to Sikuli Script code so users can graduate from the learnable model offered by block-based programming to the advanced model offered by text-based programming.
- Sikuli Blocks should actively prevent users from making syntactical errors by providing visual feedback and by inferring the user's intent. It is better to generate valid code that makes guesses for the user than to generate syntactically invalid code. This goal is more important for basic features (e.g., `click` blocks) than for advanced features (e.g., function blocks), since advanced users should be better equipped to interpret error messages.
- Sikuli Blocks should be integrated into the existing Sikuli IDE. Block programs should live alongside text programs and use the same set of commands, buttons, and shortcuts whenever possible.

4.0 User Interface Design

Sikuli Blocks introduces a Scratch-like block-based environment into the existing Sikuli Script IDE. Thus, it is built on top of the graphical user interfaces of the Sikuli IDE, described in [1], and Open Blocks, described in [6]. This section describes the interesting additions and changes Sikuli Blocks makes to those interfaces.

4.1 The Main Window

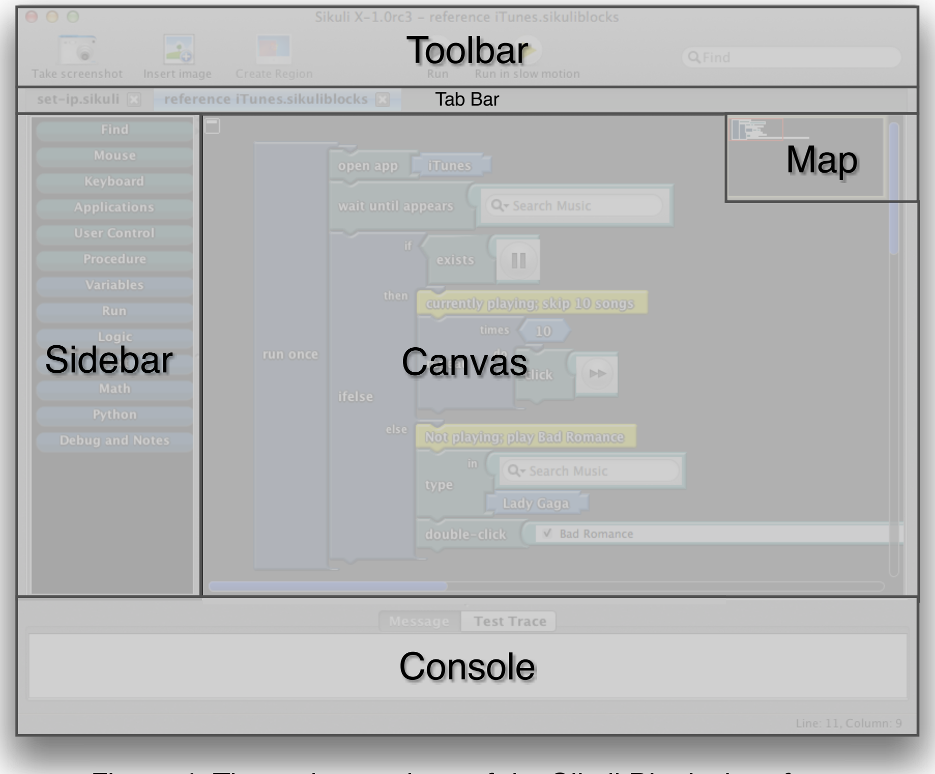


Figure 4: The major sections of the Sikuli Blocks interface.

Sikuli Blocks is offered as an addition, rather than a replacement, to the existing Sikuli Script code editor. Thus, the buttons and menu commands present in the code editor are reused in the blocks environment.

Users can create block programs by choosing “New Blocks Script” from the File menu. Sikuli Blocks creates a new tab for the new file’s block environment. Block-based programs and text-based programs can be open at the same time, and users can switch between them using the tab bar.

In the block environment, a sidebar contains a list of block categories. When a category is clicked, a popover appears offering a visual representation of the blocks in that category. For example, when the “Mouse” category is selected, `click`, `double click`, `drag and drop`, and other mouse-related blocks appear in the popup. Blocks in the popup can be dragged out onto the central portion of the screen, called the “canvas.” In the canvas, blocks can be connected together and configured to specify a complete program. For large block programs, the canvas also offers a “map” that shows a miniaturized version of the entire program. Clicking on a region of the map scrolls the canvas to that position in the program, offering an efficient way to navigate complex programs. This organization closely resembles that found in Scratch.

Standard editor commands found in the code editor are also available in the block environment. Changes to blocks (additions, movement, connections, etc.) can be undone or redone, and blocks can be copied and pasted within an environment. Block programs can be opened and saved. The block environment also respects relevant user preferences set in the Preferences window, including keyboard shortcuts for taking screenshots. Finally, users can build block-based unit tests with Sikuli Test, as described in [2].

The toolbar at the top contains the same commands present in the Sikuli Script code editor. Screenshot blocks can be created with the “Take screenshot” or “Insert image” buttons (see section 4.2), and block programs can be run with the “Run” and “Run in slow motion” buttons. The “Create Region” button and search field are also present in the toolbar, but they are not implemented in the current version of Sikuli Blocks.

The bottom portion of the window contains a console that displays the output of block programs. Syntax errors, runtime errors, and printed messages outputted by the `print` block appear in the console.

Sikuli Blocks supports over 100 built-in blocks, which are described in the following sections.

4.2 Primitive Blocks

Primitive blocks contain the data upon which other blocks operate. They are analogous to primitives found in text-based languages, and can also be thought of as “types.” There are six primitive blocks: `text` blocks represent a string, `number` blocks represent an integer or floating-point number, `boolean` blocks represent “true” or “false,” `variable` blocks represent the name of a variable, `argument` blocks represent the name of an argument to a function, and `screenshot` blocks represent an image or screenshot.

Users can specify data for primitive blocks by editing the block directly, and the block displays the data as its contents. For example, when a user clicks on a `text` block, an inline text field appears where the user can enter the desired string. We use the same inline text field approach for `number`, `variable`, and `argument` blocks.

The image used for a `screenshot` block can be specified in several ways. First, users can specify an image when the block is created by clicking the “Take screenshot” or “Insert image” buttons in the toolbar. Second, if the user drags a screenshot block out from the sidebar, the block displays a placeholder “camera” icon which the user can double-clicked to insert the desired image. Third, users can replace an existing image in a screenshot block by double clicking on the image. Fourth, users can select a screenshot block or the parent of a screenshot block in the canvas and choose “Take screenshot” or “Insert image” from the toolbar to replace the image in the block. Offering multiple ways to add `screenshot` blocks makes the system more learnable, since many different approaches a user may try first will work.

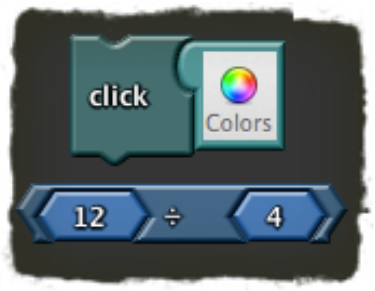


Figure 5: Primitive blocks connected to other blocks as arguments. Notice how the block shapes fit together.

Primitive blocks can be connected to other blocks as “arguments.” For example, a screenshot block can be connected to a `click` block and two number blocks can be connected to a ÷ block. To prevent syntax errors resulting from unsupported connections, basic blocks cannot be connected to blocks that do not have the correct type. For example, only number blocks can be connected to a ÷ block and attempting to connect a screenshot block to a ÷ block will show the screenshot block as visually disconnected. Like a puzzle piece, the type a particular block can take is hinted by its shape. For example, numbers are shaped like a hexagon, and blocks that take numbers have pentagon-shaped holes. Since other blocks have different shapes, users can learn what blocks they can use by visually comparing shapes, rather than resorting to trial and error.

This static type-checking scheme is very useful to prevent a common class of syntax errors, but since Python is dynamically typed, it is desirable to not make this static typing mandatory. Thus, more advanced blocks, such as function blocks, can take any type of block as an argument.

4.3 Expression, Statement, and Control Flow Blocks

The remaining blocks operate on primitive blocks. They do not store data themselves, so they are controlled only by their connections to other blocks and cannot be edited directly. **Expression** blocks return transformations of data that are fed as arguments into other expression blocks, or into **statement** blocks which execute a command.

Control flow blocks contain a sequence of statement blocks. The simplest control flow block is the `run once` block, which executes a sequence of statement blocks when the “Run” button is clicked. The compiler, described in section 5.2, searches for `run once` blocks and recursively compiles their children into the resulting Sikuli Script program. If multiple top-level `run once` blocks exist in a program, they are all compiled and ordered arbitrarily. Conditionals, loops, and functions are also provided as control flow blocks, as described in section 4.8. If a block does not have a control flow block as an ancestor, the compiler ignores it entirely.

When a new block program is created, a `run once` block is automatically added with an attached "Drag blocks here" comment. As described in section 6.1, these initial conditions helped new users figure out how to connect blocks such that they are run correctly.

4.4 Color Scheme

Block colors were chosen to differentiate between types of blocks without compromising the readability and glancibility of a complex block program. Scratch chose to use a saturated color palate where each of the eight block categories was represented by a different color. This results in bright, multicolored programs that look playful, but sacrifice readability. Since Sikuli is designed for a more professional audience, we chose a more tame color palate. Control flow blocks are represented by a dark color, statement and expression blocks by a lighter color, and primitives by a still lighter color. Generally, this makes programs flow smoothly from dark colors on the left to light colors on the right. This smooth flow differentiates the types of blocks and makes it easier to find a section of a program at a glance, but avoids the hard color transitions found in Scratch that make programs hard to read. Additionally, to break apart programs into automation sections and support sections, we color the Sikuli library blocks in shades of green and all other blocks in shades of blue.

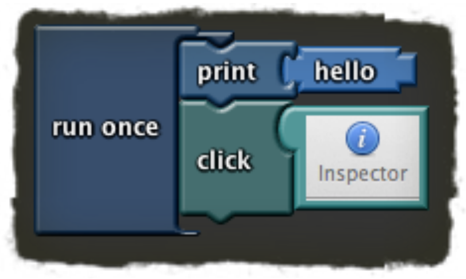


Figure 6: The possible block colors.

4.5 Sikuli Library Blocks

Sikuli library blocks implement the functions provided by Sikuli Script. These blocks provide direct support for automation. Sikuli library blocks fall into several categories:

- Blocks that interact with GUI elements, such as `click`, `double click`, `drag and drop`, and `type`.
- Blocks that find elements on the screen, such as `exists`, which returns `true` if an element resembling a screenshot is on the screen, and `wait until appears`, which blocks until an element resembling a screenshot is visible.
- Blocks that perform a sequence of commands when something on screen changes, such as `on appear`, which runs when a given screenshot is first seen on screen. As in Sikuli Script,

these blocks must be used with a `observe` block that informs the runtime to start looking for changes on screen.

- Blocks that can `open`, `close`, or `switch` to a particular application, specified by a name.
- Blocks that interact with a user at runtime, including `show popup` and `ask question`.

Most of these blocks take screenshots as arguments. Since it is tedious and potentially undiscoverable to drag out screenshot blocks for each Sikuli library block, Sikuli library blocks include placeholder screenshot blocks already attached as default arguments. As described in section 4.2, these default arguments can easily be replaced with actual screenshots by, for example, double clicking on them.

These blocks are natural for non-programmers to use, since they directly refer to the ways in which a user interacts with a computer. They also perform general, system-wide actions, rather than the application and task specific actions provided by Automator. For example, `click` can click on anything the user sees on screen in any application. This generality provides a great deal of flexibility in automation workflows, which satisfies our design goals.

4.6 Python Library Blocks

Python library blocks implement many common functions available in a standard Python distribution, including basic mathematical operators, trigonometric functions, random number generators, and debugging functions such as `print` and `assert`. These blocks provide flexibility to perform small tasks that are not derived directly from automation.

4.7 Variables

As in Python, variables store the result of an expression. They are identified by a special `variable` primitive, which contains a name as a string. The `set` block stores a value into a variable, and the `get` block returns the value of a variable. Variables may be useful to refer to a screenshot in several different `click` blocks, to keep a counter of the number of times an action has been performed, or to remember the answer a user gives to an interactive prompt.

Since Python variables do not have static types, we do not use static types for variables in the block environment; the `set` block takes any type of value, and the `get` block returns any type. This lack of type checking can result in runtime errors. However, since Sikuli Blocks offers few types and it is obvious what types most blocks can take, type errors resulting from variable blocks are uncommon and easy to detect.

4.8 Conditional, Loop, and Function Blocks

Sikuli Blocks derives much of its ability to express complex automation tasks through conditional, loop, and function blocks. They are all control flow blocks, as described in section 4.3.

Conditional blocks include `if` and `if else`, which take a `boolean` argument and run a sequence of commands based on whether the `boolean` argument is true. Conditional blocks can be used to, for example, detect whether a window is already open or needs to be opened before the program can proceed.

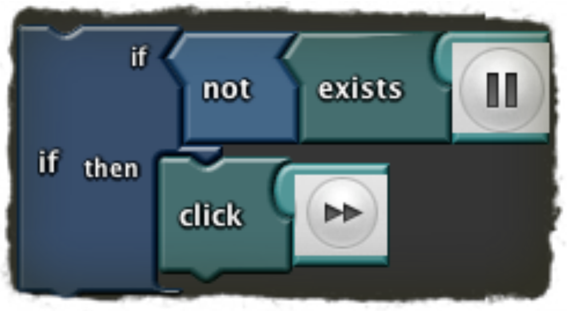


Figure 7: The `if` block used to control whether a button is clicked.

Two loop blocks are offered: `while` blocks take a `boolean` argument and run a sequence of commands while the boolean argument is true. `repeat` blocks perform a sequence of commands a fixed number of times, as specified by a `number` argument. The `break out of loop` block performs the function of Python's `break` statement, and the `continue` block is compiled to Python's `continue` statement. Loop blocks can be used to, for example, click all the checkboxes on screen.

Function blocks allow users to call and define functions. The `define` block takes a function name, zero or more argument names (specified by an `argument` primitive), and a sequence of commands. A function can return a value using the `return` block. The `call` block takes a function name, zero or more arguments, and optionally the module where the function exists. In both cases, the number of spots available for function arguments expands as arguments are added; we maintain the invariant that there is always one empty spot at the bottom where a new argument can be added. That empty spot is specified as "(arg `n`)" where `n` is the number of the argument. It is surrounded in parentheses to indicate that it will not actually be used until it is filled. See figure 8.

Function blocks are considered an advanced feature; they are useful for maintaining readable code in a large Sikuli Blocks program, but they are rarely necessary to accomplish automation tasks. In fact, one of the most common uses of functions in automation scripts is to run a sequence of commands when something changes on screen. These functions can be defined with the much simpler `on appear` family of blocks described in section 4.5. Additionally, non-programmers have trouble understanding the concept of functions.

Thus, we designed function blocks for advanced users, and provide some advanced features that allow a great deal of flexibility. First, like variables, we do not enforce static typing on arguments. Second, a `call` block can call any function available in the Python environment; it does not necessarily have to call a function defined by a `define` block. This allows users to call Python library functions that are not available as blocks without needing to use custom Python blocks. By setting a module that should be imported in a `call` block, users can even call functions in custom Python modules.

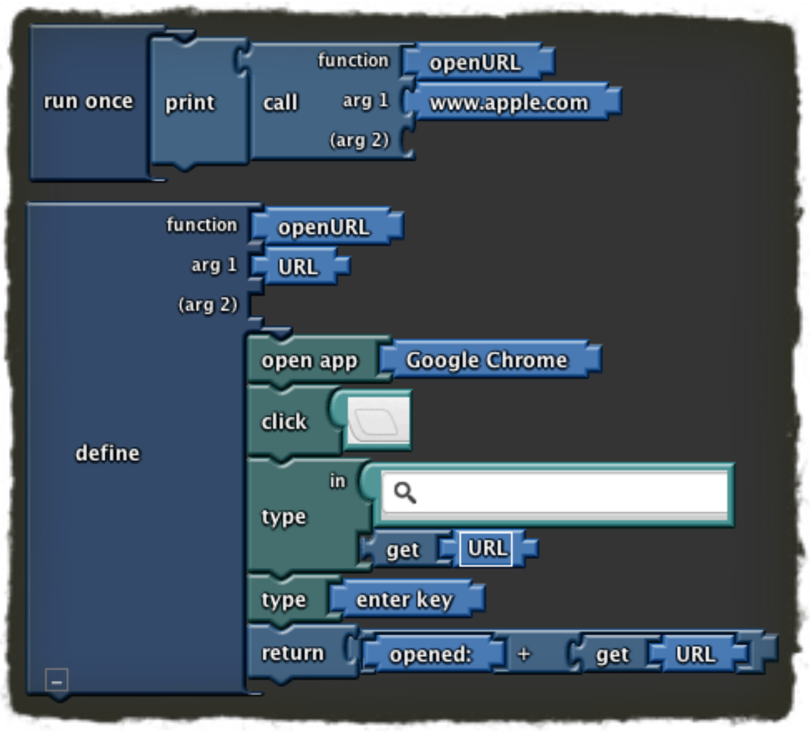


Figure 8: An `openURL` *function defined and called with function blocks. The function opens the provided URL in Chrome and returns a string to indicate success. This function could be useful if a program needs to open many different URLs.*

4.9 Custom Python Blocks

Custom python blocks allow advanced users to write fragments of raw Python code inline in a block program. This satisfies our "escape hatch" design goal; goals that are impossible or tedious to accomplish using blocks, but can be accomplished using Python, can be implemented with custom Python blocks.

There are two types of custom Python blocks: expressions and statements. Like functions, they take zero or more arguments of any block type. Arguments can be referenced from the Python fragment using a placeholder of the form `$n`, where `n` is the argument number. For example the second argument connected to a custom Python block can be referenced as `$2`. Additionally, expressions return any block type, so they can be connected to statement blocks or other expression blocks. Python expression and statement blocks look and operate just like other expression and statement blocks, but users can type raw Python code directly inside them. Thus, they essentially allow users to define custom blocks. Currently, only a single line of Python code can be entered into each block, and we do not offer a custom control flow blocks. These would both be simple extensions to our design.

4.10 Converting Block Programs to Python

We also had the design goal of allowing users to “graduate” from the block environment to the text-based Python environment. If a user starts a block program but wants to finish it in a text editor, we offer a “Convert To Python Script” menu item, which compiles the block program to Python and opens the Python code in a new editor in the IDE. The implementation outputs readable Python code; comments, naming, and order are maintained. However, changes made in the text editor are currently not reflected in the block environment (see Future Work in section 7.1).

5.0 Implementation

Sikuli Blocks is implemented in Java using the Swing GUI toolkit.

Sikuli Blocks was built on top of Sikuli Script, Sikuli IDE, and OpenBlocks. Sikuli Script provides the underlying library that implements all of the automation blocks. Sikuli IDE implements the text-based editor, including all of the surrounding features that were reused in SIkuli Blocks, such as the main window, console, preferences, and Sikuli Test panel. OpenBlocks provides the underlying blocks data model, including serialization and a specification to design a block language, as well as a fully functional Scratch-like GUI to choose and manipulate blocks.

This existing code base, which has been extensively described in [1] and [6] drastically reduced the amount of new code that we had to write for Sikuli Blocks. The primary implementation contributions of Sikuli Blocks consist of the automation block language it defines, the compiler that converts block programs in to Python scripts, the integration support to make the frameworks work together, and the minor additions and fixes to OpenBlocks to support the unique properties of Sikuli automation.

5.1 Block Language

OpenBlocks allows developers to easily define block languages using an XML standard. Each type of block is specified by properties such as its name, its color, its type (primitive, function, statement, or control flow), what it returns, what it arguments it takes, and its category. This standard is described in [6]. Sikuli Blocks includes a custom XML document defining over 100 blocks. OpenBlocks then automatically interprets this XML file to create a fully functional Swing JComponent where the Sikuli blocks can be dragged out and connected.

5.2 Compiler

Sikuli Blocks implements a simple compiler to convert block code into Python that can be run with Sikuli Script. Each of the over 100 included blocks has a corresponding compilation stub that takes an OpenBlocks Block object as input, which contains information about the block and its connections, and outputs a fragment of Python code. We implemented helper functions that perform common compilation tasks, such as compiling a function given the function name as a string and its arguments as OpenBlocks objects. Thus, most compilation stubs are implemented with fewer than ten lines of code.

To avoid many classes of syntax errors which often confuse non-programmers, compilation stubs are written to be lenient with the arguments they take; if an argument is missing, it chooses a sensible default value that allows the script to run without generating a syntax error. For example, if a `while` block does not have an attached `boolean` condition block, the compilation stub compiles the block to: `while True:`.

Compilation stubs are called by adapters that are registered with the compiler to compile a particular block. The compiler provides a `compileBlock` function that takes an OpenBlocks Block as an argument, and calls the adapter whose registered type matches that of the block.

A block is responsible for compiling its children by calling `compileBlock` on its children and using the resulting Python code appropriately. Specifically, blocks with arguments compile their arguments, and statement and control flow blocks compile the next block in the sequence. Thus, to generate a complete block program in Python, the compiler is simply told to compile each top-level control flow block. The compiler also maintains a list of modules that need to be imported. At the end, the compiler inserts `import` statements at the top of the program to ensure the program will run in the default Python environment.

5.3 IDE Modifications

We modified Sikuli IDE to accommodate both the existing text-based editor and the new block editor in the main window. Many functions of the IDE are shared between both editors, including the toolbar buttons, the screenshot capture GUI, the open and save dialogs, and the Sikuli Script bindings. To reuse these functions and to minimize code duplication between the two editors, we established an interface of about twenty methods that an editor can implement to be fully supported in the main window. This interface includes methods such as `insertScreenshot`, which the central IDE controller calls to ask an editor to insert a screenshot at a given path, and `saveFile`, which is called when the user chooses the “Save” menu item. It also includes methods that specify the functionality that an editor supports, such as `supportsRegions`. Both the text-based editor and the block editor implement this interface and thus become first-class citizens in the main IDE window.

To implement the functionality behind this interface, we had to move much of the code out of views and text-based controllers and into central controllers. For example, previously the "Take Screenshot" button class depended directly on the text editor to insert a screenshot at the correct position. Now, the "Take Screenshot" button is managed by a `CaptureController` class that calls the `insertScreenshot` method on the current editor, and the editor class does the actual work to display the screenshot in a manner appropriate for its type. These changes required a significant amount of refactoring, but the result is a much cleaner model-view-controller design that can support future expansion.

5.4. OpenBlocks Modifications

OpenBlocks is a relatively immature framework. We added several minor features and fixed bugs that affected usability. Some of these changes include:

- Added support for dynamic images on blocks to allow screenshot blocks to display a thumbnail of the screenshot.
- Re-implemented undo to work across all serializable changes in the block environment. The new undo implementation also groups actions that occur close together in time so a single invocation of undo reverses all changes that occurred recently.
- Fixed many bugs related to block selection.
- Customized the color scheme.

6.0 Evaluation

We performed multiple rounds of user tests throughout the development process. We focused on our target audience: users who were comfortable performing everyday tasks on their computers, such as surfing the web, but had little or no programming experience. The tests focused on the learnability properties of both the text-based Python environment and the block environment. The tests also revealed limitations and learnability issues with the underlying Sikuli Script implementation, but we consider these issues outside the scope of our project.

6.1 Early Usability Tests

We used an iterative process to develop Sikuli Blocks and performed informal usability tests after each iteration. Many of the usability issues discovered in these early tests were fixed in future iterations. Some of the issues found and fixed in early iterations include:

- In the first iteration, new block programs started off empty. Users did not understand how to connect blocks together and were confused why nothing happened when they did not add a `run once` block. We addressed this issue by starting new block programs with a `run once` block.

- After adding the `run once` block by default, some users did not connect blocks to the `run once` block and thus were still unsuccessful in running their programs. We addressed this issue by adding a comment next to the `run once` block that stated "Drag blocks here." After this comment was added, users quickly figured out how to correctly connect blocks together so they would run.

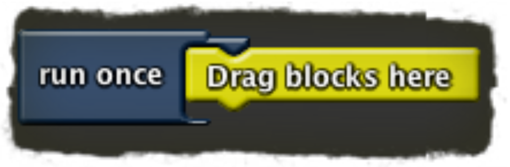


Figure 9: The default canvas for new block programs. In early iterations, new block programs started off empty.

- In early iterations, the only way to delete a block was to drag it off screen to the left. This behavior was not learnable, so we added support for the "delete" key to delete blocks.
- We observed that, rather than deleting blocks, some users preferred to simply move aside blocks that did not want to run. This strategy is less destructive, since the blocks can be dragged back into place if needed. So, we ensured that blocks not connected directly or indirectly to a `run once` or a function definition block are ignored by the compiler, and thus do not result in unexpected errors or runtime behavior.
- Early iterations did not support undo and redo. These features were found to be very important to make users efficient, since new users spent a significant amount of time making each change, and undoing a change manually was very costly. Thus, undo and redo were implemented in later iterations.
- Some of the block categories had confusing names, which made blocks hard to find. We tweaked the naming, arrangement, and contents of each category in response to feedback.

6.2 Final Usability Tests — Method

We conducted a formal round of usability tests with our latest iteration of Sikuli Blocks. We tested four users on a series of three increasingly complex tasks in both the text-based environment and the block-based environment.

We chose testers relevant to our target audience based on two questions:

1. On a scale from 1-10, how comfortable do you feel using your computer for everyday tasks, like surfing the web and playing music?

2. On a scale from 1-10, how comfortable do you feel with programming in any language, where 1 means you have never tried to program?

All of our chosen testers responded with a score of 8 or higher on question 1, and a score of 3 or lower on question 2. Thus, all testers felt very comfortable using their computers for common tasks, but did not feel comfortable programming.

We gave testers a brief, verbal introduction to Sikuli, indicating that it was a tool to help them perform repetitive tasks on their computers. We explained that they could tell Sikuli what they wanted it to do by taking screenshots of things to click on or type into. We did not show example programs.

We then asked each tester to perform three tasks, first using the text-based environment, and then using the block-based environment. If a tester had not made any forward progress for several minutes, we let the tester move on to the next task. If a tester could not complete a task in a particular environment, we did not ask that tester to try a more difficult task in the same environment. For example, if a tester could not complete task 1 in the text-based environment, we did not ask him or her to attempt task 2 in the text-based environment. The three tasks were:

1. Please use Sikuli to automate the following task: open www.apple.com in a new tab in Google Chrome.

2. Please use Sikuli to automate the following task: open iTunes. If iTunes is currently playing, skip to the next song. If nothing is playing, search for "Lady Gaga" and play "Bad Romance."

3. When your automation program is run and iTunes is playing, instead of skipping one song, please modify your program to skip 10 songs.

The first task could be completed with a simple, linear program involving few blocks or statements. The second task required the use of a more diverse set of blocks or statements, including an `if else` block, or a Python `if` statement. The third task was best completed using a `repeat` block or a Python `for` statement. Thus, the tasks increased in complexity, both in terms of the diversity of blocks used and the programming intuition required. Since each user test lasted less than a half hour, we did not user test more advanced Sikuli Blocks features, such as variables, functions, and custom Python blocks.

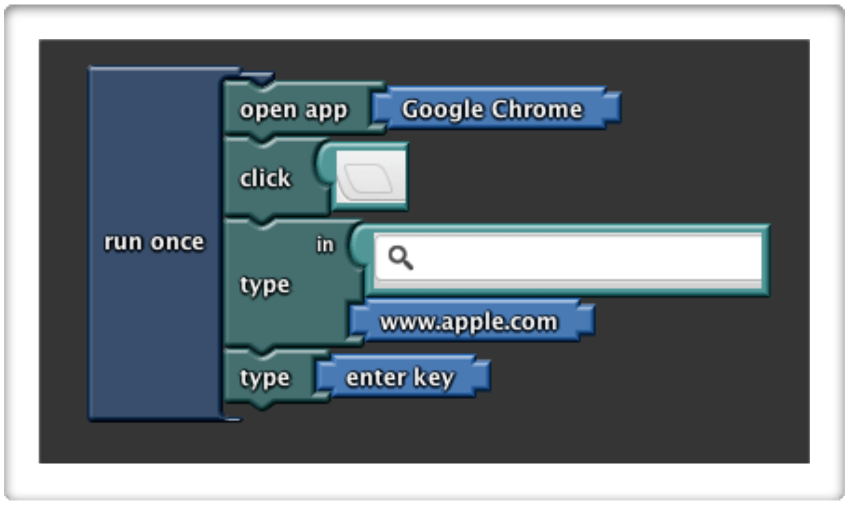


Figure 10: Reference solution to task 1: Open apple.com in a new tab in Google Chrome.

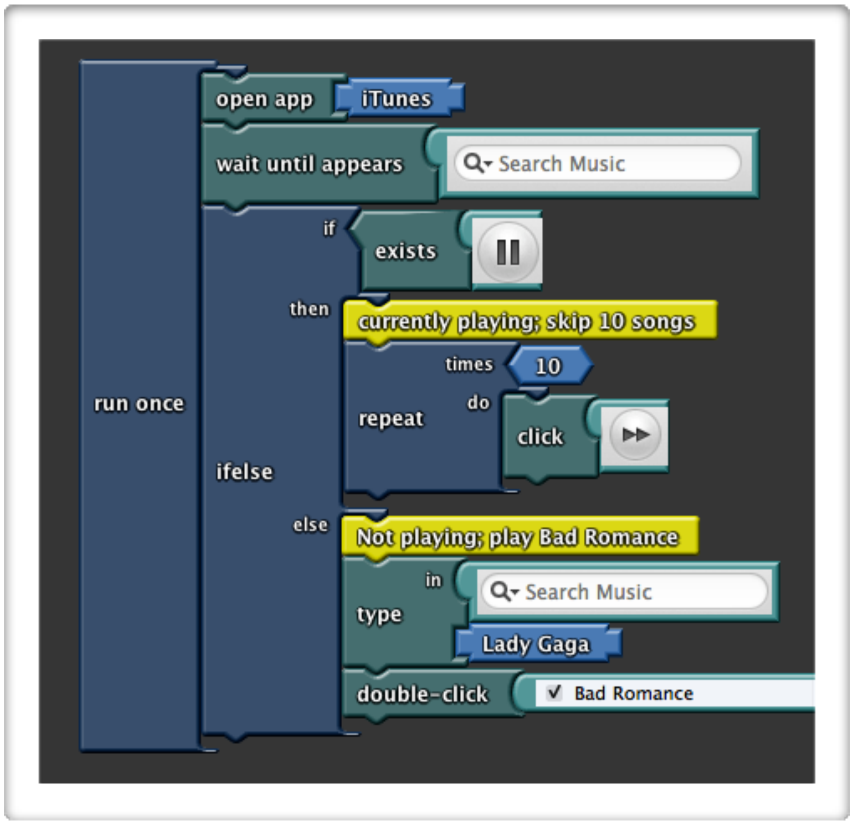


Figure 11: Reference solution to task 3: If iTunes is currently playing, skip ten songs. If nothing is playing, search for "Lady Gaga" and play "Bad Romance." The reference solution to task 2 is the same as the one shown here, except the `repeat` *block is not used.*

6.3 Final Usability Tests — Results

Testers were universally more successful with the block-based environment. Of the four testers, only one was able to complete task 1 in the text-based environment, while all four successfully completed task 1 in the block-based environment. None of the testers could complete tasks 2 and 3 in the text-based environment, while three of the four testers completed all three tasks in the block-based environment. Detailed results for each environment are summarized below.

Text-Based Environment Observations

Testers immediately reported feeling intimidated by the empty text editor. For example, after looking at the interface for about a minute without clicking or typing anything, one tester said "I don't think I know how to do this." This initial reaction suggests that the text-based environment is likely to scare off many users without a programming background before they would even begin to experiment with the tool.

Eventually, all users tried to use the text-based interface. The tester who made the least progress in this environment did not keep any text they entered, and instead simply took a series of screenshots. The tester reported that she did not know how to put the screenshot into the text she entered. Without any commands to accompany the screenshots, nothing happened when the tester clicked "Run." The other three testers made forward progress with the text-based interface by using the command sidebar. After a few minutes, these testers successfully used the `click` command with a screenshot. However, these testers encountered several problems they had difficulty solving.

First, testers only used the limited set of commands available in the command sidebar, and got stuck when a command they needed was not shown. Task 1 was best completed with Sikuli's `openApp` command to launch Chrome, and the `Key.ENTER` constant to navigate to the URL that they entered into Chrome's address bar. Since the command toolbar contained neither of these options, the testers got stuck. As a hint, we suggested that testers check the online documentation, available from the "Help" menu. The testers spent a short amount of time reading, and ultimately none found helpful information. Instead, the testers attempted to work around the missing commands by using commands they knew. For example, instead of using `openApp`, one tester used `click` and `type` statements to search for "Chrome" in Spotlight. Working around the inability to type the enter key proved challenging, and only one tester was successful.

From this issue, we can conclude that users without a programming background are unlikely to use any functionality they cannot see directly on the screen. For example, they will not guess the names of commands to type in. They are also unlikely to read documentation where they could learn new commands that are not visible. Thus, while the eighteen commands in the sidebar help users perform some tasks, it is important for an automation tool targeted at non-programmers to make *every* command the tool offers visible in the GUI in some manner.

Second, testers encountered syntax errors in the text-based environment, which were difficult to fix. For example on the first task, when users tried to type in “www.apple.com” into Chrome’s address bar, all testers initially tried to use the command sidebar to enter: `type(www.apple.com)`. This resulted in a syntax error due to missing quotation marks around the string, “www.apple.com.” Users had trouble understanding the cryptic error message: “NameError: name 'www' is not defined” since it gives no hint that quotation marks were missing. Through experimentation, all users were able to fix that error, but similar errors provided a source of frustration throughout the test. The single tester who completed task 1 in the text-based environment, and thus attempted task 2 in the same environment, tried to use an `if` statement, but did not know the right syntax, which prevented him from completing the task.

From this issue, we can conclude that syntax is a major roadblock for non-programmers. Thus, automation tools should make an active effort to prevent users from making syntax errors. The text-based environment could partially address this problem by fixing errors for the user. For example, it could automatically insert quotation marks inside a `type()` statement. Additionally, the text-based environment could make error messages more readable for non-programmers. However, in general, syntax errors are probably unavoidable in any system that excepts freeform input from the user.

Block-Based Environment Observations

The block-based environment resolved many of the issues users encountered in the text-based environment.

First, testers reported feeling much more comfortable using the block-based environment. Testers started experimenting by dragging blocks within seconds of seeing the interface, and all figured out that they had to connect blocks to perform the task within a couple minutes.

Second, when testers did not know which block to use, they clicked through the categories to explore the blocks they had available. Since all available commands are visible in one of the categories, users generally had much more success using the correct commands. For example, in the first task, three of the four testers found the `open app` block and used it to quickly open Chrome (one tester instead still searched from Chrome in Spotlight). In the second and third tasks, three of the four testers quickly found the `if else` and `while` blocks and used them correctly.

None of the testers encountered syntax errors in the three tasks. In some cases, syntax errors were avoided due to the lenient compiler. For example, the `type` block does not require quotation marks around its string. In other cases, syntax errors were avoided due to type checking. For example, only blocks that return a boolean type can be connected to an `if else` block, so users quickly realized what type of block to use. Additionally, since the block environment constrains the input from the user (i.e., it does not let the user enter freeform text, except in custom Python blocks), users could not provide extraneous inputs that would confuse the compiler.

Testers still had trouble understanding how to use some blocks. For example, while all of the testers quickly found the `enter key` block in the first task, they did not know how to connect it to their program. The `enter key` is a primitive text block with a predefined constant inside, so it was intended to be connected to a `type` block. Testers tried dragging the `enter key` block under their existing program, or beside another `type` block that already had a text block attached, but they noticed that it did not connect, and thus did not expect their program to work. Through experimentation and observation of the `enter key` block's shape, all testers figured out how to use it after a few minutes.

This issue reenforces the need to design blocks to be intuitive. Specifically, blocks should provide the context needed to understand what it does. The issue could be mitigated with documentation, for example by including a tooltip on the `enter key` block that indicates it should be used with a `type` block, or even better, by automatically inserting the needed context around each block, for example by including an attached `type` block when an `enter key` block is dragged out.

Testers also had trouble using Sikuli in the visual manner it expects. For example, in the second task, three testers searched for a block that would explicitly tell them if iTunes is playing, but no such block is available. The intended solution was to use an `exists` block to detect the presence of a particular iTunes GUI element that is only visible when iTunes is playing. For example, testers could have checked whether the pause button is visible on screen (see figure 11). This solution was not intuitive for these testers. After several minutes, we offered the hint, "see what changes visually when you press 'play' in iTunes." Two of the three stuck testers understood this hint and successfully completed the task, while the third tester gave up after several more minutes. Since visual thinking is critical to using Sikuli, we are unsure how to address this issue without major changes to the manner in which Sikuli operates. However, the issue could be mitigated with examples that teach users how to think about screenshot-based automation.

Finally, testers encountered several minor issues and bugs that led to confusion. For example, one tester overlapped two blocks such that the block in the back, which was connected to the program, was completely invisible. This prevented his program from working until he noticed the problem. Sikuli Blocks could solve this issue by moving aside blocks that overlap.

7.0 Conclusion

7.1 Future Work

Sikuli Blocks was designed and developed on a tight schedule. There are many ways it can be improved, including:

- Feature Parity with Sikuli Script: There are still commands and features available in Sikuli Script that we did not have time to design and implement as dedicated blocks. First, Sikuli Blocks currently provides no support for regions, which allow users to constrain interactions to a particular part of the screen. Second, Sikuli Blocks does not currently support the Match and Pattern classes without the use of custom Python blocks, which provide fine-grained control over Sikuli's interactions. Third, Sikuli Blocks currently offers only limited support for lists and iteration. Solutions to these limitations would allow Sikuli Blocks to express more automation programs, and we believe that these solutions could be implemented in a manner consistent with our design that does not sacrifice learnability.

- Sikuli Script to Sikuli Blocks Translator: Users can convert a Sikuli Blocks program to a text-based Sikuli Script program using the "Convert to Python Script" option (see section 4.10) but we did not implement a converter that can translate a text-based Sikuli Script program into blocks. This feature would give users greater freedom to work in whatever manner that made the most sense for their current workflow without feeling locked in to either the block or text paradigm. It could be implemented by parsing a Python script's abstract syntax tree and mapping each node or pattern of nodes in the tree to a particular block. Unsupported nodes could map directly to custom Python blocks.

- Improved Block Organization: Currently, blocks are grouped into categories, which are identified by a text button in the sidebar. As the number of blocks grows, it becomes harder for users to find the blocks they want. This makes the block environment less efficient, and it makes it more likely that users will choose the wrong block for the task they are trying to accomplish. There may be better ways to organize blocks that would mitigate these issues.

- Documentation: We have not yet written user-facing documentation for Sikuli Blocks. One of our design goals was to allow users to immediately start using Sikuli Blocks without having to read documentation, but certain types of documentation may help users who get stuck nonetheless, or users who want to explore more advanced features. For example, tooltips could help explain how to use each block. Additionally, online tutorials, especially video tutorials, could teach users how to think about Sikuli's visual programming approach.

- Other minor features that would enhance the block editing experience, such as search, multiple selection, faster access to blocks, a better visual representation for blocks that overlap, and a more modern visual design.

7.2 Closing Remarks

We set out to explore how automation tools could be improved for users without a programming background. Specifically, we wanted to design a tool that would be easy to learn for non-programmers, but still provide all users a great deal of flexibility in expressing complex and interesting automation programs. The visual paradigm offered by a block-based environment proved to be a natural fit for addressing this goal.

Usability tests proved that Sikuli Blocks is much more learnable and usable by non-programmers than Sikuli Script; users who were unable to accomplish even simple tasks in Sikuli Script were able to accomplish interesting tasks in Sikuli Blocks. Since Sikuli Blocks retains most of the features present in Sikuli Script, the block-environment offers the flexibility to automate a wide range of tasks in any application. Thus, we believe that Sikuli Blocks strikes a better balance between flexibility and learnability than other existing automation tools, such as AppleScript and Automator.

7.3 Acknowledgments

I would like to thank Prof. Rob Miller, Tsung-Hsiang Chang, and Geza Kovacs for their indispensable guidance, feedback, and support throughout the project.

8.0 References

[1] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. "Sikuli: Using GUI Screenshots for Search and Automation." UIST 2009, pp. 183-192. Available HTTP: http://uid.csail.mit.edu/projects/sikuli/sikuli-uist2009.pdf

[2] Tsung-Hsiang Chang, Tom Yeh, and Robert C. Miller. "GUI Testing Using Computer Vision." CHI 2010. Available HTTP: http://groups.csail.mit.edu/uid/projects/sikuli/sikuli-chi2010.pdf

[3] Franck Dernoncourt. “Sikuli Plays Angry Birds on Google Games.” 2011. Available HTTP: http://sikuli.org/blog/2011/08/15/sikuli-plays-angry-birds-on-google-games/

[4] Tom Yeh. “Interacting with computers using images for search and automation.” Ph. D. thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 2009 Available HTTP: http://hdl.handle.net/1721.1/53308

[5] William R. Cook, et. al. “AppleScript.” 2006. Available HTTP: http://www.cs.utexas.edu/~wcook/Drafts/2006/ashopl.pdf

[6] Ricarose Vallarta Roque. “OpenBlocks : an extendable framework for graphical block programming systems.” M.S thesis, Dept. of Electrical Engineering and Computer Science, MIT, Cambridge, MA, 2007 thesis Available HTTP: http://hdl.handle.net/1721.1/41550

[7] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. "The Scratch Programming Language and Environment." ACM Transactions on Computing Education, November 2010. Available HTTP: http://dl.acm.org/citation.cfm?id=1868363

[8] “App Inventor for Android.” Official Google Blog, July 2010. Available HTTP: http://googleblog.blogspot.com/2010/07/app-inventor-for-android.html