University of Bergen
Department of Informatics

Automating User Interfaces for a Multi-way Dataflow Constraint System

Author: Karl Henrik Elg Barlinn
Supervisors: Jaakko Järvi, Mikhail Barash

May, 2022

Abstract

A scriptable User Interface (UI) can be set to record the user's actions into a script, and then play that recorded script back over different data. The purpose is to automate oft-occurring use patterns. Though such automation is useful, especially for advanced users of particular software systems, scriptable UIs are not common. We conjecture that the implementation cost of such features is too high for them to become common.

The project develops a generic approach for scripting where this feature could be packaged into a library, to be reused by different UIs. In this approach, the effort needed to implement scripting is considerably reduced.

The context for this thesis is the use of multi-way dataflow constraint systems in Graphical User Interface (GUI) programming. Such systems can represent the state of a GUI in a concise and well-structured manner. These state representations can be inspected and manipulated programmatically, which is what we exploit for generic scripting too. Concretely, we build scripting support for the HotDrink GUI framework that is based on multi-way dataflow constraint systems and provides a mechanism for structural manipulation of GUI elements.

Acknowledgements

I would like to thank my supervisors, Jaakko Järvi and Mikhail Barash, for their invaluable feedback and discussions.

Secondly, I would also like to thank my fellow students and friends without who writing this thesis would have been a lot less interesting, but perhaps more efficient.

Finally, I extend my sincere thanks to my family who has been supportive and genuinely curious about my work with this thesis.

Karl Henrik Elg Barlinn

Tuesday 31st May, 2022

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

When performing the same action repeatedly, one begins to see patterns in the expected outcome. We are, in a sense, following a series of steps to produce a result from a set of inputs. In other words, we are following an informal *algorithm*. Such repetition of the same steps becomes tedious quite quickly, and we start thinking of ways to automate our algorithm. This process applies to all aspects of life, and can be seen a driving force behind many inventions. For instance, before computers, the grocer had to manually calculate the total cost of an order for each item purchased at the convenience store; while today this process is automated by scanning a machine-readable code on each item and letting the computer do the calculation.

The demand to automate our lives extends to our every-day interaction with computers as well. While in the past we interacted with computers using command line interfaces, today we do most of our interaction with Graphical User Interfaces (GUIs). Thus, we, the users, often wish to automate repetitive interactions within these GUIs. However, developers might not necessarily want to invest in *GUI automation* due to the effort it requires. Implementing automation from scratch will add another layer of complexity over often already complex GUI code, and increases the amount of code that will need maintenance in the future. For any reasonable developer to implement GUI automation, the benefit must outweigh these costs. While this calculation must be done on a per-application basis, the cost of having GUI automation can generally be lowered by using a *framework* compared to creating automation from scratch.

Conveniently, modern GUIs are usually created with the help of frameworks. Developers must in many frameworks manually handle the relationship between the different elements in a GUI. These relationships can be very complex; GUI elements can interact with each other in non-trivial ways. One study found around one third of all GUI code exists simply to validate the user's actions and handle dependencies between elements [40].

HotDrink [35] is a framework which aims to simplify the dependency management of GUI elements with *multi-way dataflow constraint systems*. When using the HotDrink framework, dependencies between elements of the GUI are explicitly defined in *constraints*. These constraints exist within a *constraint system*, which a *constraint solver* will uphold to the best of its ability. The formalizing of relationships between elements frees GUI developers from writing the aforementioned validation code.

However, a shortcoming of HotDrink (and most other frameworks) is its lack of support for GUI automation. The goal of this thesis is to extend the existing HotDrink framework with GUI automation capabilities; the extension is called *HDScript*.

We created GUI automation similarly to how many other standalone GUI automations have been created; that is, by recording the user's actions and by writing scripts. GUI automation scripts are composed of instructions that the GUI interprets and executes. A disadvantage of manually writing scripts is that they require the user to be able to write them in a scripting language. Some automation suites will therefore record what the user does within the GUI, and then create a script from these recorded actions. However, blindly recording exactly what the user does might cause the context of an action to be lost.

The context is of great importance when trying to understand the *intent* behind an action. For instance, if a user were to double the size of an image, the old size is the context of the action, and it is needed to understand why the new size was given. Without the context, it would be impossible to understand why the user gave exactly the new size of the image. We show that in order to create an accurate automation workflow by recording the user's actions, we must capture the essence of the context by asking the user for their intent behind each action. Thus, even users without any programming experience can accurately express their informal algorithms. In all, this thesis demonstrates that GUI automation, script recording, editing, and playback can be implemented with only a very small development effort per GUI, by having GUI automation built into the HotDrink framework.

To give an introductory example of GUI automation, consider a GUI for modifying the dimensions of an image and a user who needs to resize a collection of images using this GUI. The user's informal algorithm is as follows: first, set the width of the image to double of its current value. Then, set the height of the image to the new width of the image. For instance, applying the algorithm to an image with the dimension of 480×240 pixels will result in an image with the dimensions 960×960 pixels. If there are only a couple of images to resize, the algorithm can be performed manually using the GUI. However, this becomes infeasible when the number of images to resize increases significantly, for example, to thousands. It is at this scale that automation becomes appealing. Once an automation workflow has been created, the number of times it must run is insignificant, as it is the computer that does all the tedious work.

The structure of this thesis is as follows. In Chapter 2, we give a detailed background for and context of the thesis, focusing on multi-way dataflow constraint systems and scriptable Graphical User Interfaces. Chapter 3 gives two motivational examples. In Chapter 4, we detail the implementation of HDScript developed in this work and describe problems faced in the implementation. This chapter also discusses how to recognize the user's intent, and some implementation details related to the user's intent. Next, we give an overview of existing and related works in Chapter 5. In Chapter 6, we evaluate HDScript by comparing an application created with the framework to an application with an identical feature set created without any external frameworks. Finally, in Chapter 7, we give a conclusion of the thesis and a glance at possible future work.

Chapter 2

Background

2.1 Scripting Languages

A *scripting programming language* behaves like an interpreted language. In interpreted languages, an interpreter reads the source code[1], at runtime, statement by statement and executes them directly. Many scripting language emphasize development convenience over execution speed. This makes them easier to use for those new to programming and those whose primary job is not programming, as they do not need detailed knowledge of the underlying system. Development within scripting languages is usually more laid-back than compiled languages such as C++ and Java. For instance, in the scripting language PHP one does not need to explicitly declare that a variable exists, instead variables are declared when they are first used [16]. Another distinguishing feature of scripting languages is the fact that the data type of a variable can change simply by assigning a value with a different type to it, such as in Python [20].

There are three primary uses for scripting languages, identified in the book *The World of Scripting Languages* [31].

Glue Applications Scripting to create *glue applications* by combining (or glueing) off-the-shelf components into a new application using a *glue language*. These glue languages are often general-propose scripting languages, while the components glued together can be written in compiled languages. Glue applications might be created for rapid prototyping

[1]A *script* is a program written in a scripting language.

or as the final product. The vital characteristic is the adaptation of the application to changing user requirements, for instance, using shell scripting to create new commands[2].

To Control Programs Scripting to control an existing program programmatically via an Application Programming Interface (API) or by manipulating the program objects directly. With this kind of control, a user can automate repetitive tasks, customise the behaviour, and optimise the user experience. Programmatically controlling an application can be seen as a specialised glue application, as there are existing components (i.e., the application) and a glue language to tie them together. However, the difference lies in how no new features are created. Everything the API can do is already accessible to the user via the GUI. For instance, Selenium [25] can be used to programmatically interact with any webpage.

General Purpose Programming Due to the plenitude of features and ease of use, it might be easier to develop specific programs in a general-purpose scripting programming language than in a compiled language. Developing an application in a scripting language is particularly desired when the script developer's time is more important than execution speed. Interacting with the underlying operating system to do administrative tasks is a prime example of such a case. Examples of generic scripting languages are Python [19], Lua [12], ECMAScript [4] among others.

2.1.1 Automating User Interfaces

Many applications are only intended to be used by a human user through a GUI. Regardless, one often wishes to be able to automate, and write scripts, for such applications too. External automation tools, i.e., software robots, may help in these situations. They work the same way a regular human user would interact with an application, such as using, or emulating, a mouse and keyboard. These external automation tools have the same features as applications with built-in scripting capabilities. However, they often lack the ability to view or make sense of the internal structure of the program they are automating. A common strategy to circumvent this is to use the GUI structure (e.g., HTML Document Object Model) itself to navigate the application. Relying on the layout of an application is fragile, however, as the visual look and feel could change without warning;

[2]As an example, to count the number of words in the PDF `input.pdf` would be `pdftosrc input.pdf - | wc -w`. In this example, `pdftosrc` and `wc` are written in compiled languages.

for example, the screen dimension and size may affect the layout. Often these types of automation are also version-dependent, as it is often the case GUIs change version to version. It should be noted that the problem of version dependency is not exclusive to external automation tools. For example, the SAP GUI Scripting automation interface [22] (which has internal scripting capabilities) is unable to play scripts written for an older version of the software [33]. The underlying problem is the reliance on the user interface's visual structure.

In contrast to external automation, i.e., applications that can only automate other programs, there are programs that have ingrained scripting capabilities. Automation of these programs is not limited to what the user can see and do in the GUI. Instead, it can leverage an internal API of a program, which increases the ease of use, efficiency, and coverage. To differentiate between the above two kinds of programmatic control of applications, we say an application with built-in scripting support has *internal scripting capabilities*, and programs that help automate other programs have *wraparound scripting capabilities*.

While it is possible for each application to add its own unique flavour of scripting internally, it would be very wasteful, in terms of programming work, to "reinvent the wheel" that way. Such a scripting feature is demanding to implement correctly and laborious to keep up to date as the application evolves. A more sustainable solution is to move the scripting away from the visual layer of the application and into the logical layer. Preferably, this should be done generically in a framework to facilitate scripting in multiple applications. More concretely, scripting should be implemented at the view-model level and not at the view level.

2.2 Model–View–View-Model Pattern

The Model–View–View-Model (MVVM) pattern [37] divides GUIs into three distinct parts: model, view, and view-model, showcased in Figure 2.1. The *view* is what the user sees and interacts with, e.g., buttons, input fields, keyboard shortcuts, graphics, and other controls over the GUI. The *model* controls the application's business logic, which can be complex parsing of the data sent by the user, or it can be as simple as writing what it receives into a database. The model is often written in a different programming language and unaware of the GUI. The view is only responsible for the interface, and the model is only responsible for the business logic. Thus, the view should not be doing complex

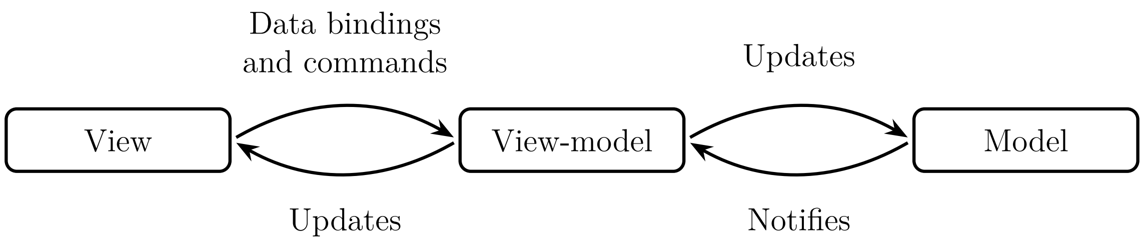


Figure 2.1: How the different parts of MVVM interacts with each other.

calculations, and the model should not handle the interface. Instead, a third layer will be the middleman, called the *view-model.* The view-model layer acts as an adapter between the view and the model—translating what the view is showing into data the model can understand and handling view states such as disabled elements and selection [37].

2.3 Multi-way Dataflow Constraint Systems

The *dataflow* of an application describes how the values *flow* between *variables.* A variable holds a value which may change during execution. Consider a *method* which, when executed, sets the value of a variable to double the value of another variable, e.g., `f := c * 2`. We say the value of `f` flows from the value of `c`. When we add another method, `c := f / 2`, the relationship between `f` and `c` is expressed comprehensively. The two methods together with the `f` and `c` variables are called a *constraint.* A constraint enforces the relationships between variables with a non-empty set of methods and the variables to enforce. Each method assign new values to a subset of the constraint's variables, such that the constraint becomes satisfied. All methods in a constraint must use all the variables of the constraint as either input or output [35]. That is to say, a method is a function that takes some of its constraint's variables and returns a separate set of the constraint's variables. Constraints are a part of a *constraint system*, which is a tuple of a variable set and constraint set. The constraint system uses a *constraint solver* to keep the constraints satisfied when the system's variables are assigned new values from outside the system. Some older implementations of constraint solvers [36] only allow *one-way* constraints, i.e., exactly one method per constraint [41]. When there are multiple methods in a constraint, we say the constraint system is *multi-way.*

As an example, the statement `f = c * 2` can be considered as a dataflow constraint. The constraint consists of the variables `f` and `c`, and the methods `f := 2 * c` and `c := f / 2`. Consider the (unsatisfied) constraint `f = 2 * c` with the variables `f = 10` and `c = 0`. The constraint solver may execute either of the constraint's methods to satisfy the constraint system. As an example, assume the `c := f / 2` method is arbitrarily chosen. The method calculates the variable `c` to be `5` (`c := 10 / 2`). The constraint is now considered satisfied, or enforced.

2.3.1 HotDrink Framework

This thesis builds on the *HotDrink* [39] framework, which is a multi-way dataflow constraint system [35]. In HotDrink's representation of constraint systems, the top-most structure is called a *component*. Components have one or more variables and zero or more constraints. These components hold all possible variables a constraint within it can use. Constraints in HotDrink denote implicitly in each method signature which variables it uses.

To provide more flexibility in constructing constraint systems, HotDrink deviates from the idealistic description of a multi-way dataflow constraint system above by adding *variable references*. In HotDrink, a component does not directly hold variables but rather variable references. These variable references might *reference* a variable. If they do not reference a variable, they are called *dangling variable references*. A dangling variable reference can be linked to the variable of another reference variable. In so doing, the variable reference will no longer be dangling. The separation of variables and variable references is needed to allow components to reference a variable in a different component.

We give an example of a constraint system, in Listing 2.1, to showcase HotDrink. The purpose of this constraint system is to convert temperature values between Fahrenheit and Celsius. Without going into too much detail, we can observe the component defined in Listing 2.1 has two variables, `celsius` and `fahrenheit`, and a single constraint with two methods `c2f` and `f2c`. The first method convert Celsius into Fahrenheit, while the second does the inverse, converting Fahrenheit into Celsius. The resulting component is visualized in Figure 2.2.

```
component my_component {
    var celsius, fahrenheit;
    constraint my_constraint {
        c2f(celsius -> fahrenheit) => celsius * (9 / 5) + 32;
        f2c(fahrenheit -> celsius) => (fahrenheit - 32) * (5 / 9);
    }
}
```

Listing 2.1: Temperature converter component between Celsius and Fahrenheit implemented in HotDrink.

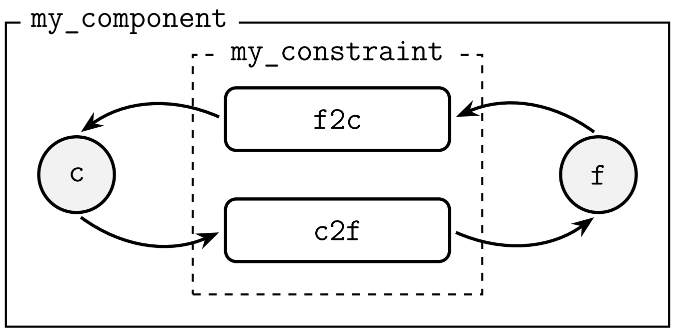


Figure 2.2: Graph view corresponding to Listing 2.1. The variables `celsius` and `fahrenheit` are represented as `c` and `f`, respectively. Circles are variables, rounded boxes are methods, the dashed box is a constraint, and the solid box is a component.

Listing 2.2 extends the above example with a new constraint (line 7 onward) which will convert temperatures between Kelvin and Celsius. There is no need to explicitly convert between Fahrenheit and Kelvin, as a change to one of the variables will cause the other variable to be updated indirectly by the `celsius` constraints. In Listing 2.2 the `celsius` variable is given the initial value `100`. This will cause the system to be solved during the initial loading, which will update `fahrenheit` and `kelvin` to their correct values with regard to `celsius`. Since `celsius` is assigned a value, the solver selects methods that do not overwrite this value. As can be observed in Figure 2.3, the `c2f` and `c2k` methods will be selected for execution.

```
 1 var celsius = 100, fahrenheit, kelvin;
 2
 3 constraint f_and_c {
 4     c2f(celsius -> fahrenheit) => celsius * (9/5) + 32;
 5     f2c(fahrenheit -> celsius) => (fahrenheit - 32) * (5/9);
 6 }
 7 constraint c_and_k {
 8     c2k(celsius -> kelvin) => 273.15 + celsius;
 9     k2c(kelvin -> celsius) => kelvin - 273.15;
10 }
```

Listing 2.2: Temperature converter component between Celsius, Fahrenheit, and Kelvin implemented in HotDrink.

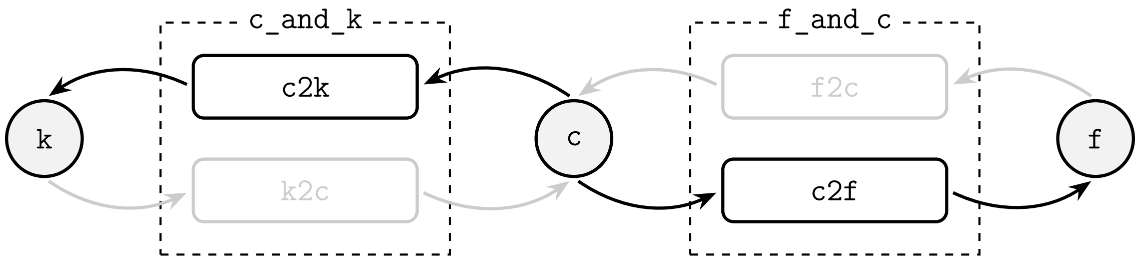


Figure 2.3: Graph view over which methods are activated (i.e. non-greyed out elements) when calculating the initial state of Listing 2.2. The variables `celsius`, `fahrenheit` and `kelvin` are represented as `c`, `f`, and `k`, respectively. The optional component name is omitted.

2.3.2 Binding the View and View-model in HotDrink

HotDrink fills the view-model role [35] in MVVM. The view and HotDrink interact with each other via binders. Each binder is an arrow pair between the view and view-model in Figure 2.1. The binding is shown in more detail in Figure 2.4. The view notifies the binder of user events (i.e., user-entered values), which in turn updates the corresponding HotDrink variable. If a variable in HotDrink is updated (though never if the binder updated it), HotDrink will send an event to the binder, and then the binder will update the view.

The binder concept allows for views to be interchangeable. A view only needs to send changes to the binder and update the relevant widgets once an update is received from the binder.

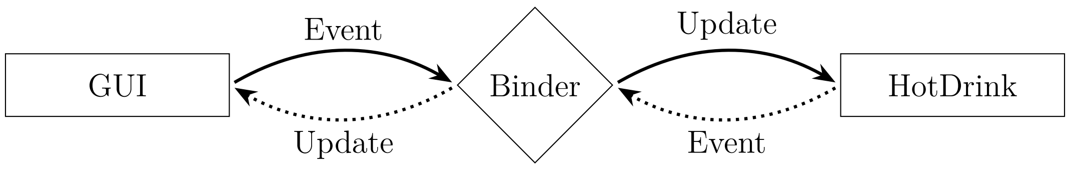


Figure 2.4: Binding between the view on the left and view-model on the right.

2.3.3 Asynchronous Methods

HotDrink's methods are executed asynchronously, so that the system can accept new requests for updating its variables, while it is being solved. HotDrink ensures that methods are executed in a consistent order. This also means that HotDrink variables cannot be directly assigned to from outside; there is a special API function, schedule command, for requesting such changes to variables' values. By scheduling a command, the view is essentially using a one-shot binder to do work in the view-model. In order for the view to update multiple variables at the same time or when the (correct) current value of a variable is needed to update a variable, it must be done by scheduling a command.

Consider a GUI with a single button, such as the one given in Listing 2.3. When the button is clicked, a counter will increase by one after some arbitrary time. Now consider a user clicking the button multiple times and very quickly. If the counter took the current value of the variable when it was clicked, the new value might be wrong because multiple clicks could occur before the variable had time to be updated. A solution to this is using a schedule command that takes the counter variable both as an input and an output. By deferring the value lookup to when the command is being executed, we can be sure of the correctness of the program.

```
const system = new ConstraintSystem();
const comp = component(["var count = 0;"]);

system.addComponent(comp);
system.update();

innerTextBinder(document.getElementById("count"), comp.vs.count);

const vars = [comp.vs.count.value]
document.getElementById("button").addEventListener("click", () => {
    system.scheduleCommand(vars, vars, (count) => ++count, true);
});
```

Listing 2.3: Schedule command counter example with a HTML view [32].

Chapter 3

HDScript: Scripting in Graphical User Interfaces

Programs with wraparound scripting capabilities have the advantage of enabling scripting for applications that lack internal scripting capabilities. However, the disadvantage of using external automation tools is the lack of knowledge of the internal structure of the program they automate. In contrast, applications with internal scripting capabilities have the opposite problems. They have the advantage that they can use internal API calls and are not limited to what the user can see and do, but this is restricting too: program-specific implementations cannot easily be reused in multiple GUIs. Moving scripting capabilities from applications to a framework will leverage the advantages and minimize the disadvantages: multiple applications are able to have internal scripting capabilities and knowledge of the internal structure. The thesis conjectures that scripting built into a framework is the only pragmatic solution to achieving a more widespread adaptation of scriptable GUIs.

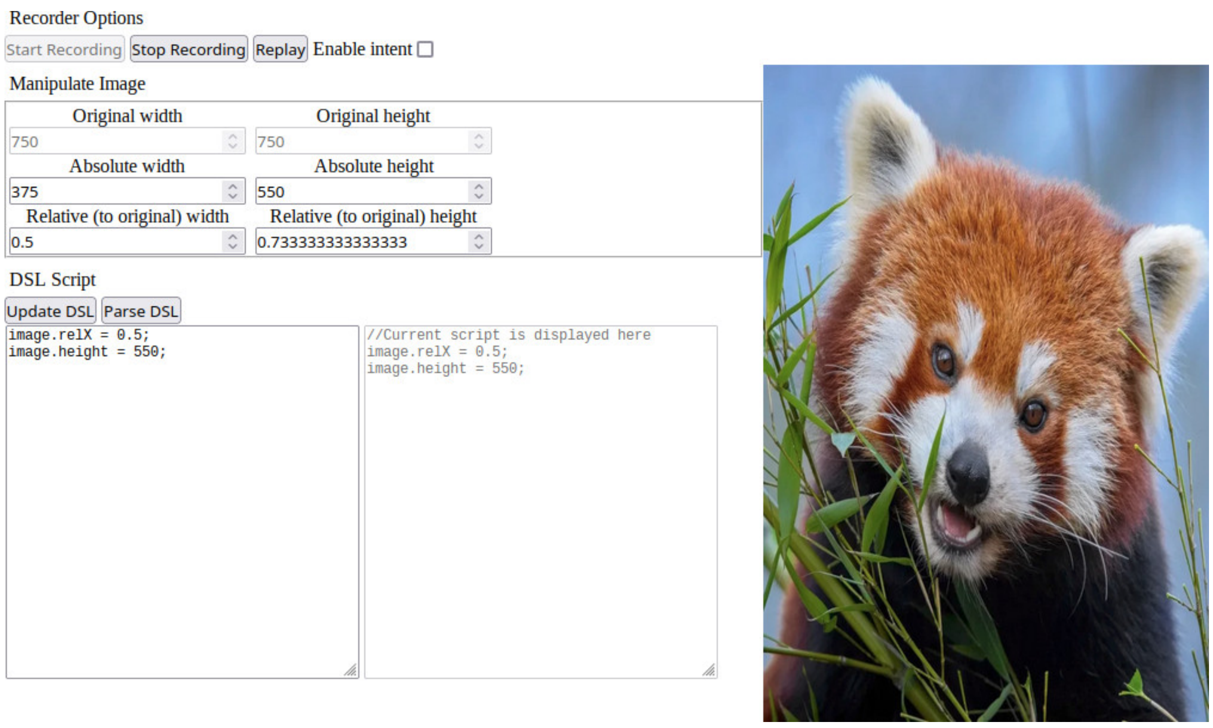
Capturing Users' Intent

When the user interacts with the GUI, HDScript records all changes sent to the view-model. However, mindlessly recording everything the user does might result in the context of a change being lost. The context is critical for understanding the intent behind a change. For example, if a user doubles the size of an image, the old size serves as the backdrop for the adjustment and is required to recognize the new size. Without context, it would be difficult to ascertain why the user specified the new image size. We show that in order to define an accurate automation workflow by recording user actions, we must understand the the action by asking the user for their intent behind each change. As a result, even people with little programming knowledge may write correct scripts.

We are now ready to showcase the main contribution of this thesis, *HDScript*, and how it has extended the HotDrink framework to provide internal scripting capabilities for all application created with the framework. We utilize the clear separation between the view and the view-model in HotDrink application; which, for automation purposes, makes it clear whether a change should be included in an automation workflow.

Two sample applications will be described: one for image resizing and another for (simple) spreadsheet manipulation. We explain the machinery of our solution through their implementation.

3.1 Image Resize Application



Original width: 750
Original height: 750
Absolute width: 375
Absolute height: 550
Relative (to original) width: 0.5
Relative (to original) height: 0.7333333333333333

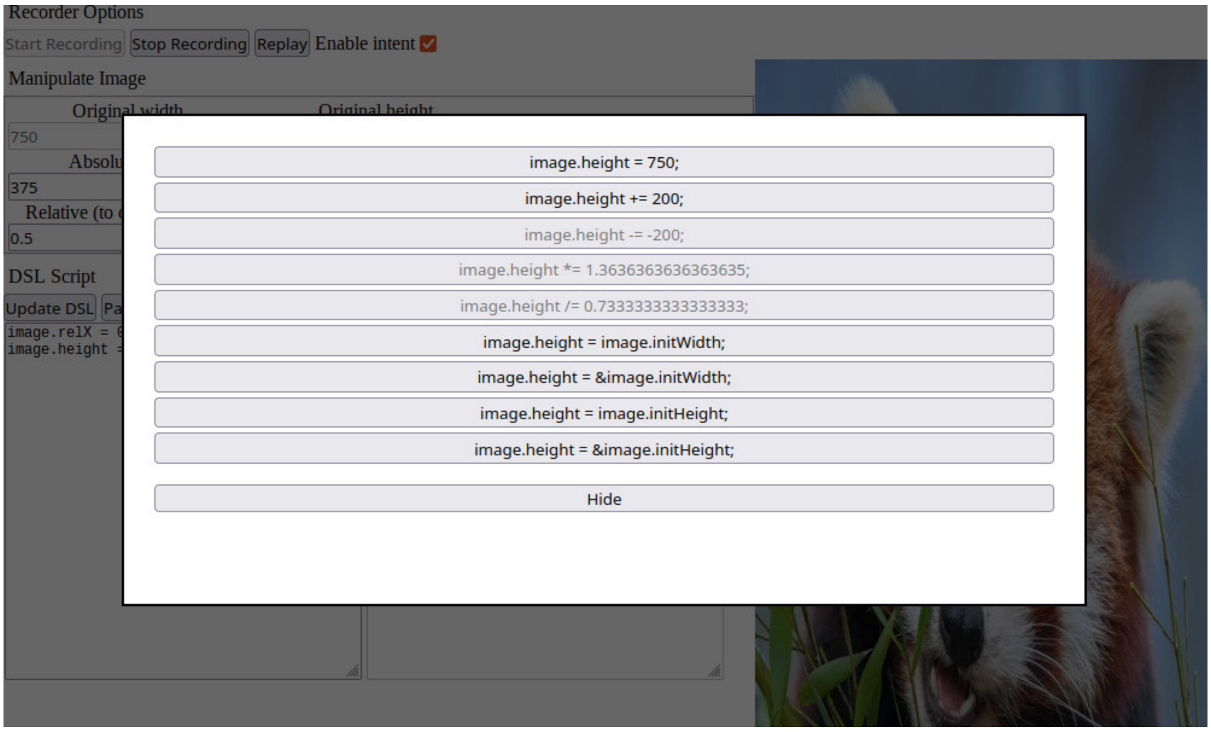
```
image.relX = 0.5;
image.height = 550;
```

```
//Current script is displayed here
image.relX = 0.5;
image.height = 550;
```

Figure 3.1: The image resize application after running the script specified in the text area at the lower left pane of the page.

The image resize application, Figure 3.1, allows the user to resize an image either to an absolute pixel size or to a size relative to the original image. The image which is resized can be seen rightmost in the GUI. There are input fields for changing the absolute width and height, and relative width and height of the image. The resize dialogue's scripting tools appear above and below these resize controls. The user can start, stop, and replay the current *action recorder*. Action recorders, discussed in detail in Section 4.1, present a way to record changes caused by a user interacting with the GUI, i.e., *actions* (described in Section 4.3). The lower right pane displays the state of the current recorder, as a list of HDScript Domain Specific Language (DSL) statements. In the lower left text pane; HDScript DSL *scripts* can be entered by a user; the DSL is discussed in Section 4.2.

Figure 3.2 shows a dialogue box that is displayed when a variable has been updated, which asks the user to specify their intention. For an in-depth description of how we aim to capture the user's intent, see Section 4.4.



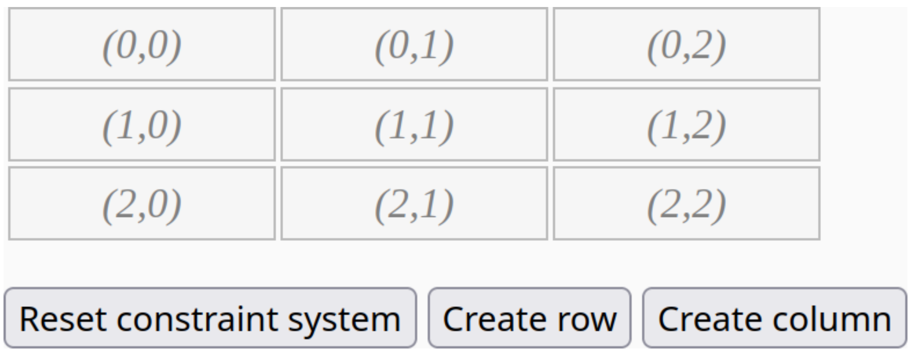
```
image.relX = 
image.height =
```

```
image.height = 750;
image.height += 200;
image.height -= -200;
image.height *= 1.3636363636363635;
image.height /= 0.7333333333333333;
image.height = image.initWidth;
image.height = &image.initWidth;
image.height = image.initHeight;
image.height = &image.initHeight;
```

Figure 3.2: The image resize with the intent dialogue box active. The change made by the user was updating the absolute height of the image from 550 to 750.

In the image resize application, the underlying constraint system is static; no new variables or constraints are added while the user interacts with the GUI. In this simple case, the application automatically supports HDScript from the outset. More advanced constraint systems, where constraints are added dynamically, might not be supported out-of-the-box, as the design of the view is critical to automating the application properly. Section 4.5 gives a more in-depth explanation and possible solutions to the dynamic constraint system problem.

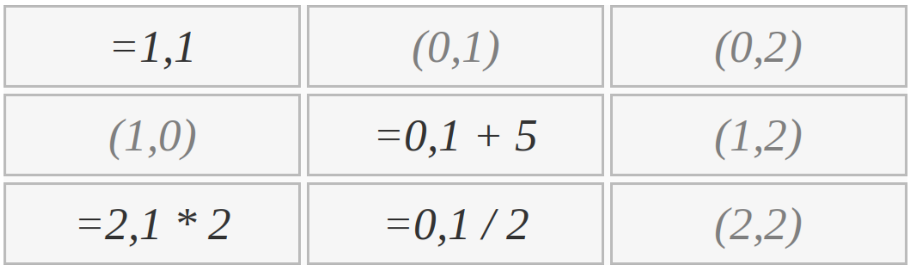
3.2 Spreadsheet Application



(0,0)	(0,1)	(0,2)
(1,0)	(1,1)	(1,2)
(2,0)	(2,1)	(2,2)

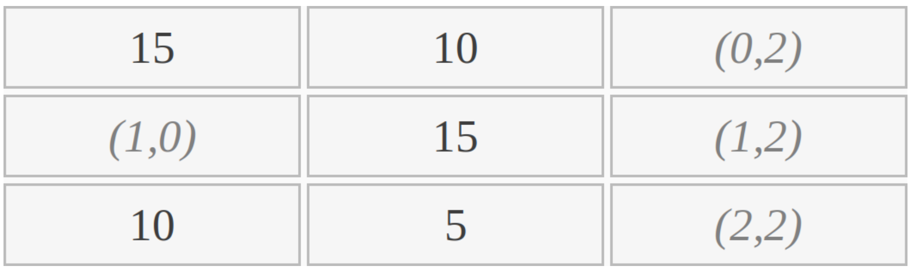
Figure 3.3: The spreadsheet application with interaction buttons.

Our spreadsheet example application, shown in Figure 3.3, has only the most basic functionalities expected of a spreadsheet application. There are rows and columns of cells to which a user can assign any value. A user can also write simple *functions* when prefixing the value of a cell with an equals sign. Consider, for example, the function `=1,1 + 10`; it takes the value of cell `1,1` and adds the numerical value `10`. The result of the function will be the value of the cell in which we write the function, see Figure 3.4 and Figure 3.5. There are also buttons to create new rows and columns, and a button to reset the state of the application.



=1,1	(0,1)	(0,2)
(1,0)	=0,1 + 5	(1,2)
=2,1 * 2	=0,1 / 2	(2,2)

Figure 3.4: The spreadsheet application with functions shown. The functions are displayed when there are no values in the referenced cell.

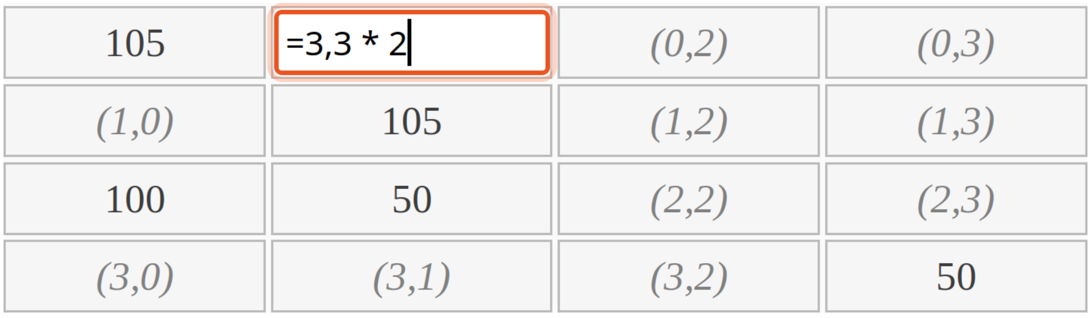


15	10	(0,2)
(1,0)	15	(1,2)
10	5	(2,2)

Figure 3.5: The spreadsheet from Figure 3.4 where the value 10 was assigned to cell `(0,1)`.

The spreadsheet cells are implemented as variables in a single HotDrink component and a function in this application is implemented as a component that is added to the constraint system. Generally, a limited dynamic constraint system is achievable by following a few guidelines defined in Section 4.5.1. These guidelines are designed to prevent the view from doing anything other than displaying the values of the constraint system.

To create a fully dynamic view using a constraint system with scripting, *custom actions* are often required. This is detailed in Section 4.5.2. For instance, when either the create row or the create column button is clicked, a custom action will be recorded. In Figure 3.6 we have used custom actions to be able to modify the view; these custom actions have created a new row and a new column of cells. The DSL script given in Listing 3.1 was the script executed.



105	=3,3 * 2	(0,2)	(0,3)
(1,0)	105	(1,2)	(1,3)
100	50	(2,2)	(2,3)
(3,0)	(3,1)	(3,2)	50

Figure 3.6: The spreadsheet from Figure 3.4 with new cells created by two custom actions. The function in cell `(0,1)` is now `=3,3 * 2`.

```
#custom create_row;
#custom create_column;
vars.v3_3 = "50";
```

Listing 3.1: Example of using custom actions.

Chapter 4

Implementing HDScript

HDScript consists of: (i) *script actions*, (ii) an *action recorder* to record script actions, (iii) a *DSL* to create actions from scripts, and (iv) an *intent recognition subsystem* to simplify creating complex scripts.

The basic building block of HDScript are the *script actions*[1], which can be seen as a change waiting to be applied to a constraint system. In essence, the action recorder, the DSL and its parser, and the intent recognition subsystem exist for and revolve around script actions. Figure 4.1, for instance, illustrates how the action recorder and the HDScript DSL parser are both considered *action sources*; that is, they produce script actions. The action recorder may divert from its route to *refine* its input before creating each script action, whereas the DSL parser creates exactly the actions stated in a script. By refine, we mean recognizing the intent of an action and allowing the user to select a more appropriate action than the default "set the variable's value to a constant value".

An action is said to be *replayed*, when the change it represents is applied to a constraint system[2]. When an action is replayed, it interacts with the constraint system in the same way a user operating a GUI interacts with the constraint system. In other words, there is no way for HotDrink to distinguish between a modification triggered by a human user interacting with a GUI and an action being replayed. This fact guarantees that HDScript is able to do everything a human user can do with a GUI.

Due to how HotDrink powered applications are separated by the MVVM pattern, script actions exist exclusively in the view-model. They do not have any concept of what

[1]There are multiple different *types* of script actions, all explained in Section 4.3.

[2]It should be noted that an action is rarely replayed by itself, normally a series of actions are replayed in succession by an action recorder; this will be discussed in full in Section 4.1.

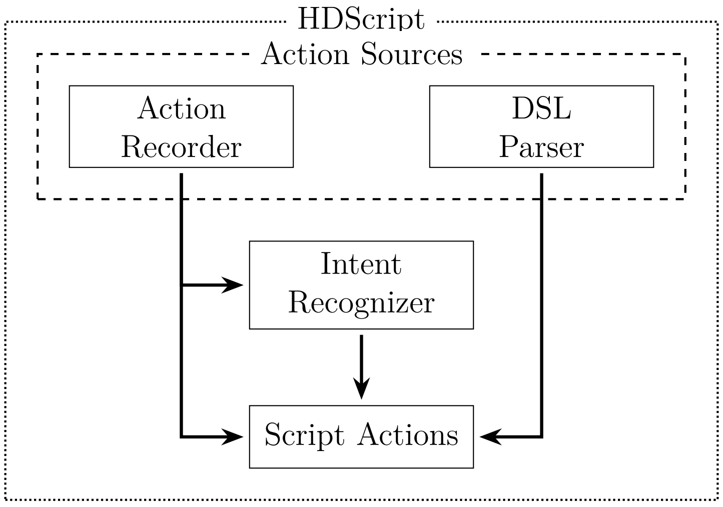


Figure 4.1: A high-level visualisation of how actions are created in HDScript.

the view is, nor any way to view or interact with the view. Additionally, the HotDrink framework has no notion of which technology the view is created with, thus it would be impossible to know how actions should interact with the view. The only exception is the *custom action*, which is created in the view and thus has complete access to it.

In this chapter, we explore the machinery of HDScript. In addition, we describe the difficulties of developing applications in which the structure of the constraint system changes at runtime. A detailed overview of HDScript can be seen in Figure 4.2.

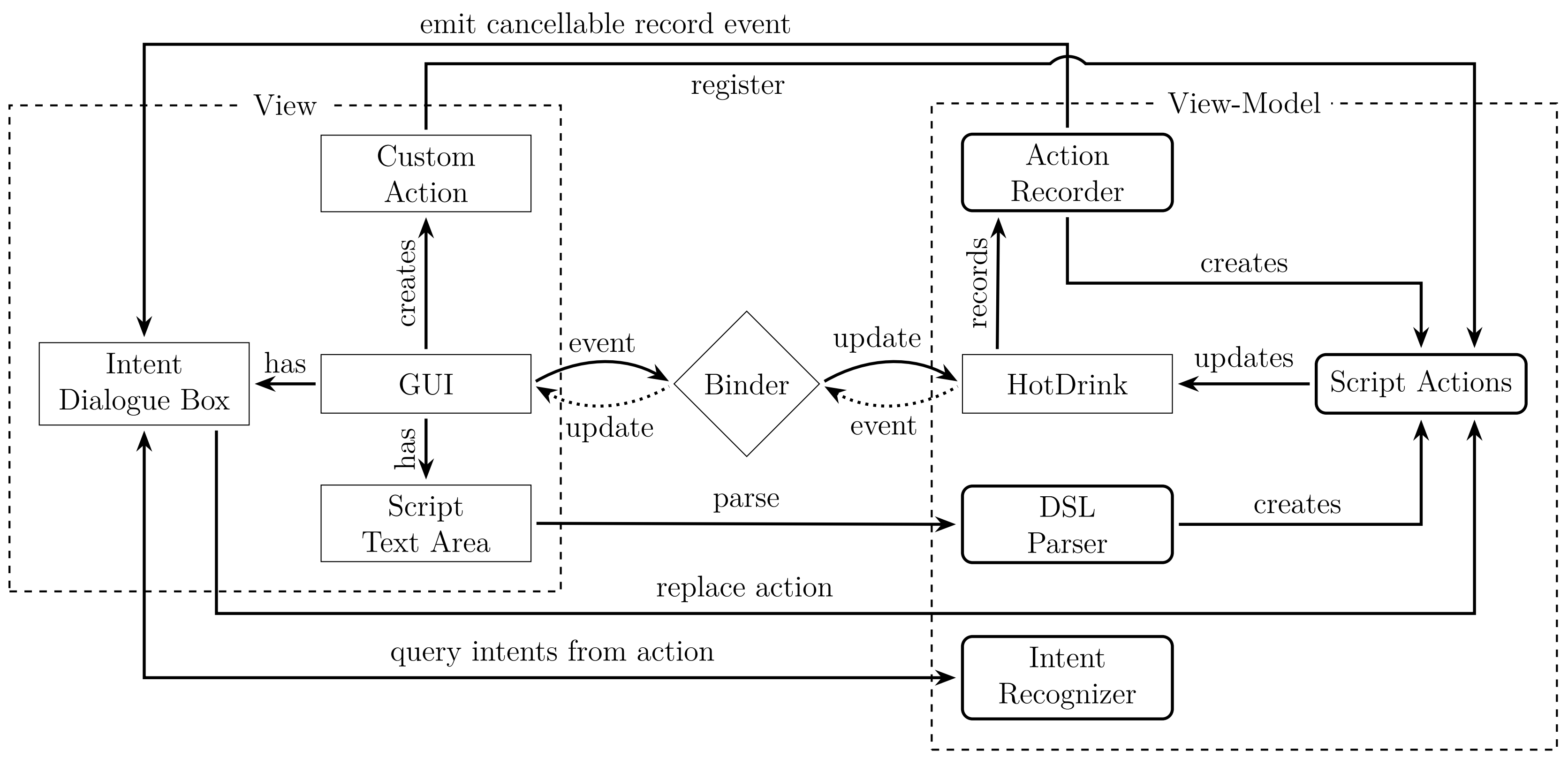


Figure 4.2: Overview of HDScript and its different subsystems (rectangles with rounded corners) are connected with each other, HotDrink, and the view.

4.1 Action Recorder

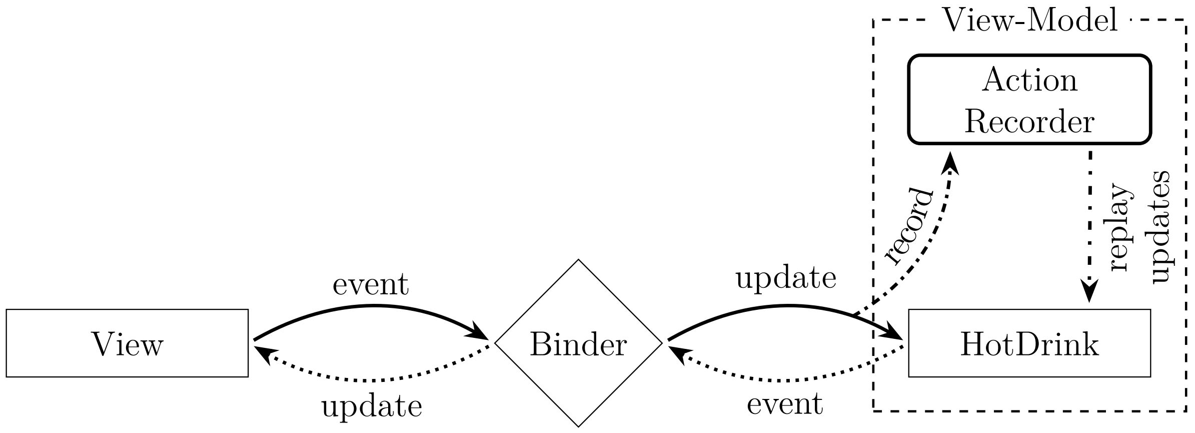


Figure 4.3: The relationship between binding the view on the left and the view-model on the right.

The *action recorder* is one of two ways to produce script actions. Action recorders are analogous to voice recorders, which first capture audio and then later replay it. Likewise, an action recorder will capture and store changes as actions to later *replay* them. Each constraint system have an associated action recorder, called the *system recorder*. The system recorder is initially not recording, see Section 4.1.1. It must be manually started by the view. When recording, the system recorder will capture all changes in the constraint system that originate outside the view-model. Implementation-wise, a list of actions is always represented by an action recorder, even if those actions were attained as the result of parsing a DSL script. Thus, when discussing replaying script actions, we in-fact mean replaying an action recorder.

To record script actions, HDScript takes advantage of how the view updates the constraint system, illustrated in Figure 4.3. A single update from the GUI can cascade through the entire constraint system potentially changing every variable. Apart from the initial update, these changes are irrelevant for recording, as they will always be executed automatically (assuming the same constraints) when replaying. Thus, when recording, we must be careful to only record the initial change as an action. To this end, we leverage a set of predefined methods that the view's binders can use to update the constraint system. By recording calls to these methods as actions, it is possible to distinguish between when the view and when constraints are changing the constraint system.

To give an example of how recording works in practice, consider the Celsius-Fahrenheit-conversion example outlined in Listing 2.1. When a user changes the value of the `celcius` variable, the `fahrenheit` variable will be updated as per the constraint on line 4 in Listing 2.1. The initial change of `celsius` will be recorded as a script action, while the following change to `fahrenheit` will not be recorded, as it was caused by the constraint system automatically being solved. In this example, only the change to the `celcius` variable is needed to fully recreate all changes made to the constraint system.

Action Recorder Events

An action recorder event is emitted when a change is about to happen in a recorder. Every type of event can be cancelled to stop the change from occurring. These events exist so that the view may intercept changes in the system recorder. For example, whenever an action is recorded, the view can cancel the event and then use the intent recognition subsystem to refine the event action. As mentioned in the introduction to Chapter 4, this is how actions may be refined.

Changing Target Component

A niche feature of the action recorder is the ability to change which component each action will act upon. This enables recordings to be replayed over multiple structurally similar components. For example, it could be useful in a GUI with several images where each image is represented by a different component. A single recording could be made for one of the images, then replayed for each image in the application by changing which component the recording targets.

4.1.1 Automatically Starting the System Recorder

A decision we had to make is when, if ever, to automatically start the system recorder. Ideally, we want to start the recorder automatically, but allow the programmer to opt-out to simplify creating scriptable GUIs. However, starting to record too early might result in unwanted actions being recorded. The problem is the lack of distinction between initial actions in the application and the user interacting with the GUI.

Consider the initialization of an application where the user is able to scale the width and height of an image. This application has a constraint system with a `width` and a `height` variable where the initial values depend on the width and height of the image to scale. Thus, the dimensions of the image must be fetched during the initialization of the application, i.e., they cannot be constant values. This initial change of the `width` and `height` variables is impossible to distinguish for the system recorder from a user making a similar change. However, this initial change should ideally not be recorded as the system performed it, not a user. It is up to the developer to make this distinction, as action recorders cannot distinguish an initial change from a real action.

Now, consider a user who wants to use the application described above to create a recording that adds 10 pixels to each dimension of an image. If the initial change were to be recorded, such as setting the image's width and height to 500 by 500 pixels, the final recording would be as follows: set both the `width` and `height` of the image to 500 pixels, then add 10 pixels to both the `width` and `height`. In other words, replaying the recording would always result in an image with a size of 510 by 510 pixels, which is not what the user intended.

Due to the problem described, the system recorder is turned off by default, which forces the view developers to enable the recording manually.

4.2 HDScript DSL

The HDScript DSL is used to control existing applications by turning statements in the DSL into script actions. The DSL consists of statements terminated by a semicolon. Scripts are interpreted linearly, there are no control structures in the DSL. We can do this because all HDScript actions can be transformed into a statement in the HDScript DSL, and all statements can be transformed back into their original actions. In other words, we can convert between the HDScript DSL and HDScript actions without losing any information. Turning an action recorder into a script serves two purposes: it preserves the recording and allows users to change it. These scripts can also be shared among users and utilised by others who are less technical proficient.

Each statement in the DSL is a serialised action. The actions are responsible for serialising itself into a valid DSL statement. Turning an action recorder into a script is therefore trivial. Deserialising a script into an action recorder, however, is not as simple. To accomplish this the HDScript DSL parser first translates the script into an Abstract Syntax Tree (AST). Then it performs a number of checks on the AST to ensure the correctness of the script. Of course the checks will all pass if the script was serialised from actions, but they may fail if it was written or modified by a user. Finally, an action recorder is built from the AST.

Script Validation

The checks performed on a script's AST exists to make sure the script is well-formed. For example, despite being a scripting language HDScript DSL disallows referencing variables, on the right-hand side, in components that do not exist. Conversely, as long as a variable is defined on the left-hand side of a previous statement the script is valid, as is demonstrated in Listing 4.1.

```
//assuming the only component is
// component comp {
//    var v;
// }
comp.v = comp.non_existing; //this would throw an error
comp.non_existing = 0;
comp.v = comp.non_existing; //this would be allowed because of the
    ↪ line above
```

Listing 4.1: Validation of a script.

4.3 Script Actions Types

In this section we give an in-depth explanation of each script action type, except the custom action which is described in Section 4.5.2.

4.3.1 Assignment Actions

```
//assuming the only component is as follows
// component comp {
//    var x = 1, y = 2;
// }
comp.x = 5; //assign-to-constant
comp.y = comp.x; // assign to value of variable
//Final values of the variables:
//comp.x = 5, comp.y = 5
```

Listing 4.2: Example of assignment actions as DSL statements.

The most straightforward types of actions are assignment actions. Implementation-wise they are separated into *assign-to-constant* and *assign-to-value-of-variable.* The former action takes a given constant and sets the variable's value to this constant. While the latter behaves similarly to the former, instead of a constant value, it dynamically retrieves the value of another variable and then sets the variable's value to the retrieved value. As can be seen in Listing 4.2, the DSL syntax for both actions is similar to assignments in any `C`-like programming language. An equality sign separates the left-hand side from the right-hand side. The left-hand side defines the *qualified name* of the variable to be updated, while the right-hand side specifies the new value. A qualified name uniquely identifies a variable within a component.

The assign-to-constant action is one of only two actions which is recorded[3]. Furthermore, the assign-to-constant action is unique in that it is the only action type which uses the intent recognizer tool, to be discussed in Section 4.4.

4.3.2 Modification Actions

Manipulating values of a variable with regard to its old values is possible with modification actions. Most assignment operators the in the JavaScript language[4] are supported. These actions follow the same syntax as the assignment statements in the DSL: a qualified name followed by an operator, then a value. Listing 4.3 gives an example script where some of these actions are used. A complete list of the possible modification actions can be seen in Table 4.1.

Operator	Assignment Name	Explanation
`*=`	Multiplication	Multiply current value by the right-hand side
`**=`	Exponentiation	Raise the current value to the power of the right-hand side
`/=`	Division	Divide current value by the right-hand side
`%=`	Remainder	Find the remainder of the right-hand side
`+=`	Addition	Add the right-hand side to the current value
`-=`	Subtraction	Subtract the right-hand side from the current value
`<<=`	Left shift	Shift the bits of the current by the right-hand side to the left
`>>=`	Right shift	Shift the bits of the current by the right-hand side to the right
`>>>=`	Unsigned right shift	Shift the bits of the current by the right-hand side to the right unsigned
`&=`	Bitwise AND	Do a bitwise AND between the current and the right-hand side
`^=`	Bitwise XOR	Do a bitwise XOR between the current and the right-hand side
`|=`	Bitwise OR	Do a bitwise OR between the current and the right-hand side
`&&=`	Logical AND	Set the value to the right-hand side if the current value is truthy
`||=`	Logical OR	Set the value to the right-hand side if the current value is falsy
`??=`	Logical nullish	Set the value to the right-hand side if the current value is nullish[5]

Table 4.1: Every modification action possible.

[3]The other is the schedule command action, discussed in Section 4.3.4.

[4]Other than the destructuring, assignment every assignment operator is possible. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators#assignment_operators.

[5]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Logical_nullish_assignment.

```
//assuming the only component is
// component comp {
//    var x = 1;
// }
comp.x *= 2; // Double the value to 2
comp.x <<= 1; // Left shift the value once to 4
comp.x %= 3; // Take of remainder of 4 which is 1
comp.x += 1; // Add one for a total of 2
comp.x **= 4; // Calculate the exponent of 2^4 which is 16
// The final value of comp.x is 16
```

Listing 4.3: Examples of modification actions as DSL statements.

4.3.3 Linking Action

Linking a dangling variable reference (i.e., one pointing at no variable) to a variable can be performed with the linking action. The linking action redirects from which variable a variable reference is reading. Linking will cause the former dangling variable reference to report having the same value as the variable reference it is linked to, effectively making the two variable references aliases to the same variable. We cannot link a variable reference to a dangling variable reference, as both the action and the parser of the DSL requires the linked-to variable reference to have a variable. An example script can be seen in Listing 4.4 and what the script does is visualized in Figure 4.4.

```
//assuming the only component is
// component comp {
//    var x = 1, &y;
// }
comp.y =& comp.x; // link comp.y to comp.x
comp.y = 10; //comp.x == 10 && comp.y == 10
```

Listing 4.4: Example of linking action as DSL statements.

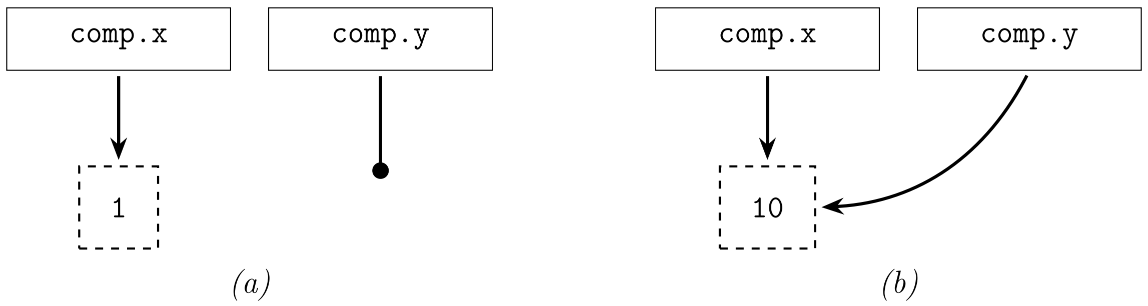


Figure 4.4: Variable and variable references in the constraint system (a) before and (b) after running the script in Listing 4.4. Dashed boxes are variables, while the solid boxes are variable references.

4.3.4 Schedule Command Action

As described in Section 2.3.3, the schedule command utility executes arbitrary code to update HotDrink variables. This utility function has its own dedicated action, the *schedule command* action. Unique to this action is how it has two *modes—record-as-value* and *record-as-function*—which can be specified when calling the schedule command function. When the record-as-function mode is used, the scheduled function is saved in a schedule command action to be later executed whenever the action is replayed. In contrast, the record-as-value mode waits for the command to complete its execution, and then takes the new values of the output variables and records each as a separate assign-to-constant action.

When a function is serialised and then deserialised in JavaScript, all variables in its scope will be lost [27]. When the schedule command action is using the record-as-function mode, it serialises the scheduled function. The record-as-function mode should therefore only be used when all variables accessed are in the set of input variables given to the schedule command; if this is not the case, the record-as-value mode must be used. However, whenever possible, record-as-function should be preferred over record-as-value because it exactly replicates what the original schedule command did.

To give an example of the schedule command action, consider the GUI with a single button described in Section 2.3.3. When the button is clicked, a counter will increase by one after an arbitrary period of time. To create this GUI in HotDrink, we use the schedule command utility and give the counter variable both as an input and an output. Depending on which mode we use, we get a very differently behaving script. If the record-as-value mode is used, it is by design the same as setting the value of the counter variable to a given number, regardless of what the previous value was. In contrast, when using the record-as-function mode, the previous value will be taken into consideration. In this example, we only access the counter variable in the scheduled command, thus the record-as-function mode can be used.

4.3.5 Component Actions

The structure of constraint systems can be modified by component actions. The *add component* action is used to create new components, whereas the *remove component* action is used to delete them. While the remove component action can easily be defined

```
// Add the following component with the add component action.

//component v1_1 {
//var &v0_0, &v1_1, code;
//    constraint {
//        (v0_0 -> v1_1) => v0_0;
//    }
//}

//Note that the JSON have been beautified to make it readable and all
    ↪ \" have been replaced with ".
#AddComponent {
  "name": "v1_1",
  "variables": [
    {"owner": "vars", "name": "v0_0", "value": ""},
    {"owner": "vars", "name": "v1_1", "value": ""},
    {"owner": "v1_1", "name": "code", "value": "=0,0"}
  ],
  "constraints": [{
      "name": "c_0",
      "constraintSpec": {
        "methods": [{
            "name": "",
            "nvars": 2,
            "ins": [0],
            "outs": [1],
            "code": "function anonymous(v0_0\n) {\nreturn v0_0;\n}",
            "promiseMask": [0]
          }]
      },
      "optional": false
    }]
};

// Now remove the added component with the remove component action
#RemoveComponent v1_1;
```

Listing 4.5: Example of adding then removing a component.

by the user, the add component action is challenging to create by hand. Listing 4.5 demonstrates both actions as DSL statements; even though the component is simple, this script would have been very difficult for a user to develop from scratch. The issue in creating an add component action is in understanding how a component is defined as a JSON object and being able to define components that the view can understand and use. We do not expect users to be able to create an add component actions from scratch. Rather, power-users can alter the generated DSL statements of add component actions if they need to make minor changes to the added components.

4.4 Recognizing User's Intent

When a user interacts with a GUI, there is an intent behind each change. However, the intent is not always clear from the interaction. Consider the script in Listing 4.6. Assume it is the initial script created after recording an update of a field from four to sixteen. Imagine the intent of the user in this example is to set the field to be four times as large as it originally was, no matter the original value. The generated script does not express this behaviour; it will always set the value of the field to be sixteen. The user could edit the initial script to be `comp.a *= 4`, which perfectly states the user's intent. However, we cannot assume every user will know what to write to communicate their intention. An advanced user is able to use the action recorder to record an initial script and then modify it to express their intent. A less advanced user cannot modify scripts without learning the scripting language and environment of the application.

```
// Assume comp.a = 4
comp.a = 16;
```

Listing 4.6: Changing a variable value from four to a constant value sixteen.

To enable these less advanced users to express themselves accurately, HDScript includes the *intent recognition subsystem*. It works by taking the old and a new value of a variable together with the variable itself, and then using intent recognizers to find possible intents. Each intent recognizer looks for intents of a specific data-type. This modular approach allows the view developers to add custom intent recognizers and even overwrite the included ones. Included by default are intent recognizers for JavaScript primitives [10], variables, and constant assignment.

All intent recognizers return a set of *intent results*. An intent result consists of an action as a HDScript DSL statement, an explanation of the action, and whether it is recommended. The DSL statement is required for HDScript to convert the intent result into a script action. In the sample applications, DSL statements are presented as the main piece of information when the user clarify their intent. The statement tells the user precisely what they will see in the generated script, and might contribute to the user learning the HDScript DSL. It is possible that the user is ignorant of the statement's meaning, therefore an explanation in plain English is included in each intent result to help the user choose the correct option. Without the explanation, the intent subsystem would require the user to understand the DSL, thus perhaps creating a barrier for non-programmers as users. Finally, each intent result can also be either recommended or not. Intent results that are valid but perhaps not very likely to be the user's intent are still presented to the user, but as "not recommended".

When to Recommend an Action

To recommend an intent result, some heuristics must be used. For example, we assume numbers with fewer digits are preferred over those with more. How few digits a number must have to be considered fit for recommendation is arbitrary. Consider trying to recognize the intent of a changing a number from 17 to 19. In this case, we conjecture it is more likely the user indented addition (`+= 2`) over multiplication (`*= 1.1176470588235294`)[6] or division (`/= 0.8947368421052632`)[7]. There are other cases where either multiplication or division, but not both, have a reasonable chance of being the user's intent. Consider trying to recognize the intent of a changing a number from 8 to 9. In this case, multiplication ($8 * 1.125 = 9$) seems reasonable, while division ($8/0.88\overline{8} = 9$)[8] does not. A third example is the intent results $x - 5$ and $x + -5$, which are equivalent for real numbers. It is reasonable to expect the user to prefer the first intent, as it is less verbose. It is not to say the latter option is invalid, but instead the less likely to be chosen of the two. While the given examples are all from the number intent recognizer, this line of reasoning can be extended to other intent recognizers.

Presenting Intent Results

There are at least two ways to present the intent results—*continuous presentation* and *delayed batch presentation*. The former asks the user constantly for their intention, i.e., after each time an action is recorded. The latter waits for the user to stop the script recorder, and then asks for their intent at each action. The continuous presentation has the advantage that the user will remember their intention. However, it has the disadvantage of continuously asking which might become tiresome. If the user is prepared to provide their intent at each step, it should be perceived as less nagging. For instance, the view can allow the user to opt-out of the intent recognition subsystem temporarily when they do not want to clarify their intent. The delayed batch presentation is less intruding than the continuous presentation, but runs the risk of the user not remembering their intent of all actions. In the end it is up to the view how intent results should be presented.

[6]The factor approximates $\frac{19}{17}$.
[7]The divisor approximates $\frac{17}{19}$.
[8]Assuming the field of real numbers.

4.5 Dynamic Constraint Systems

Until now, all constraint systems given as examples have been *static*. A static constraint system does not change structurally after initialisation; only the values of variables are allowed to change. Disallowing the constraint system to structurally change makes it possible to support scripting without any measures from the view-developers. However, the rigidity severely limits and complicates creating functional GUIs. For instance, an application with a static constraint system must create all possible mutations of components during initialization, which for a given feature in the application might be impossible. In contrast to static constraint systems, *dynamic* constraint systems allow the structure of the constraint system to be modified at runtime. These modifications include adding or removing variables, methods, constraints, and components after the initial solving of the constraint system. Modifying the structure of a constraint system is often desired in complex GUIs. However, using a dynamic constraint system increases the complexity of creating GUIs which are scriptable. For a GUI to be scriptable, a script could be created in one instance and then run in another instance of the application without any issues.

The underlying problem of scripting in dynamic constraint systems is the desired malleability of the constraint system. Consider whether a HotDrink variable exists in a constraint system. Within static constraint systems, neither new variables can be added, nor existing variables removed. Thus, if a variable exists, it is always guaranteed to exist; conversely, if a variable does not exist, it never will. Importantly, the guarantee entails restarts of the application. In dynamic constraint systems, there is no such guarantee. New variables can be added and existing variables can be removed, either because the owning component was removed or because the variable has not been added. Therefore, in order to properly automate a GUI with a dynamic constraint system, the addition and removal of structures must be a part of the automation. There is no point in editing a variable which does not exist, as it would not affect the application in any meaningful way. In HDScript, we record structural changes as component actions, explained earlier in Section 4.3.5.

Both the view and the view-model must be built to accommodate structural changes. That is, the view must also consider structural changes in the constraint system in order to be a scriptable GUI. As was discussed in Section 2.2, the view must not store any valuable data (e.g., field input). Storing data in the view is problematic as it will be lost when the application restarts. Notably, for dynamic constraint systems, the data lost includes all components added at runtime. Thus, when a structural change in the constraint system occurs, the view must be able to reflect it.

The principal problem for applications using dynamic constraint system, is how to recreate the bindings between the GUI and the constraint system. When adding a component to the constraint system, bindings between a component's variables and GUI widgets (e.g., HTML elements) are defined by the view. HDScript is only capable of recording and replaying actions in the constraint system and not in the view nor in the bindings. Any created binders would not be preserved when the application restarts. To solve the binding's problem, an application must recreate the binders when a structural action, such as the add component action, is replayed. However, as the constraint system cannot in any way interact with the view other than with binders, this is not possible unless an action can directly interact with the view.

With the underlying issues explained, we will now explain how to create a scriptable GUI, when these GUIs have dynamic constraint system.

4.5.1 Guidelines for a Scriptable Dynamic Constraint System

To resolve the outlined issues, both the view-model and view must be carefully designed to allow for scripting in applications using dynamic constraint systems. Scripting is supported out-of-the-box when no components are modified at runtime, i.e., for a static constraint system. To facilitate scripting in dynamic constraint systems, we have identified a set of guidelines which shall be followed to allow for scripting.

1. **Changes in the view-model must automatically be reflected in the view.** If the view-model is changed (e.g., by replaying a recording), the view should be updated with the new values without any user interaction. This guideline aims to improve the user experience, as none of the values displayed is outdated. Enforcing this guideline indicates the view-model is the owner of the data, thus the view behaves as defined in the MVVM pattern.

2. **Storing valuable user input in the view should be avoided.** When a user enters data into the application (e.g., types text into a text field), the data should be persisted in the view-model. We cannot guarantee a view will persist user input between restarts, therefore all user-entered data is considered discarded by HotDrink on a restart. Any data valuable enough to be persisted must be stored in the constraint system to prevent it from being lost. Data exclusively relevant to the view can be discarded along with the view (e.g., scroll position).

3. **No new bindings are to be created, nor the view modified after the initial update of the constraint system.** This guideline prevents actions from referencing widgets in the view which might not exist when replaying the recording.

The third guideline is optional, as it restricts the view to be constructed quite drastically. The resulting applications cannot be considered fully dynamic, but rather *semi-dynamic*, as only the constraint system can change and not the view.

Consider a spreadsheet application in which the third guideline is enforced. In this application, all cells must, by the third guideline, be created during initialization. The number of rows and columns in the spreadsheet is fixed, and thus more rows and columns cannot be added at run time. The constraint system can still change, for example, by adding components with constraints between existing cells. Were cells to be added dynamically, a script might reference a cell which does not exist after a restart. This would break with the notion that a script created in one instance of an application can always be replayed in another. By restricting the view to not create new bindings after initialization, we can guarantee that all variables always will exist, thus this application is scriptable.

4.5.2 Custom Actions

It is possible to ignore guideline 3, and instead display an error if a script fails to execute. By doing so, the flexibility the guideline removes is restored, while still indicating that something went wrong to the user. In the spreadsheet example above, new columns and rows can therefore be added at runtime. However, ignoring guideline 3 runs the risk of the user perceiving the application as buggy, as in their mind a perfectly fine script from another instance does not work in a new instance. A better solution is to use *custom actions* which are able to execute view-side code to create bindings; thus solving the problem of dynamic constraint systems not being able to directly interact with the view.

In essence, custom actions are view-side functions with a unique identifier. The most important benefit of using custom actions is the possibility to serialize bindings indirectly by re-creating any binding in a custom action function. Custom actions allow for a truly dynamic constraint system without any of the drawbacks of enforcing the third guideline. While the benefit of enforcing the third guideline is the simplicity of maintenance, it restricts the view from changing at runtime, and conversely ignoring guideline 3 with custom actions imposes no such restrictions. However, there are drawbacks to using custom actions; the custom actions themselves must be maintained, and they might become deprecated as the application changes and old features are removed. These drawbacks are not present when following the third guideline.

Custom actions are created by the view before the initialization of the constraint system to execute view-side code when replaying actions in the constraint system. Custom actions only have to be registered once by the view for them to become available to all action recorders and to the DSL parser. While HDScript does not differentiate between custom actions and other actions, the view must manually tell the system recorder to play a custom action.

Consider a spreadsheet application which has created some custom actions, and does not follow guideline 3. These custom actions are: (i) adding a new row to the spreadsheet and (ii) adding a new column to the spreadsheet. Two buttons are wired to fire each of the custom actions when pressed. When a user clicks one of the buttons, the view must notify the constraint system to play and record the wired custom action. Consider a user clicking one of the buttons. The custom action is recorded and the action recorder is saved as a script. After a restart of the application, the script with the custom action runs. Both times exactly the same change is applied to the view, i.e., there is no difference whether it was a user interacting with the GUI or if a script was executed.

4.5.3 Other Attempted Solutions

The initial idea to allow for dynamic constraint systems was to use the schedule command function as an *ad-hoc* custom action. The main problem with this approach is how deserialised functions only have global variables in scope, e.g., `console` and `document` for a typical HTML view. The loss of variables makes it impossible to interact with the view in any meaningful way. To overcome this problem, the view must create global objects for the command to interact with. However, the usage of global variables is generally deprecated in modern applications [21]. Another issue is the fact that the schedule command function was designed to update HotDrink variables, and not to run view code. This forces the view developer to supply at least one output variable, even when no variables should be changed. These problems make schedule command impractical to use as a way to create a dynamic view.

Another attempted solution to allow dynamic constraint systems is to serialize the actual binders, either when bound or during the standard component serialization. However, the problem with this is how the view can be written in any arbitrary framework and programming language. There is no way we can guarantee the view can serialize its binding. For instance, we could create a view which explicitly disallows serializing the binders.

4.5.4 Example of a Dynamic Constraint system

We created the simple spreadsheet application given as an example in Section 3.2. For the application to be more than a multi-cell calculator, the end-user should be allowed to reference cells from other cells and do simple calculations. Assume that the following two formulas are available: (i) get the value of another cell and (ii) add the value of another cell to a constant number. We do not know which formulas will be used at compile time. Therefore, the application must dynamically change the HotDrink constraint system during the execution.

If no custom actions were created for this application, the size of the spreadsheet must be constant. If any more cells are added later, it is not possible to create a scriptable view without custom actions. Essentially, no view code can be injected dynamically, as the recording playback cannot create view bindings.

However, in this implementation of the spreadsheet application, we have created two custom actions. These custom actions are as described in Section 4.5.2: (i) add a new row to the spreadsheet and (ii) add a new column to the spreadsheet. With these custom actions, it is trivial to extend the spreadsheet, as can be seen in Listing 4.7.

To make a grid of cells into a usable spreadsheet, we need to be able to create some formulas. These formulas are defined by dynamically added components in the constraint system. For instance, Listing 4.8 is an example of a formula where the cell `v0_0` (the cell found in the first column, in the first row) will be constrained to the value of cell `v1_1` (cell in the second row, second column) multiplied by two. The only variable the dynamic component in Listing 4.8 owns is the `code` variable, which holds the raw user input which resulted in the component. The `code` variable is needed for the user to be able to edit the current constraint without having to rewrite the whole formula each time; it is especially needed after a restart. The ability to edit the formula, even after a restart, demonstrates the necessity of guideline 2.

```
// Register custom the actions
ScriptRecorder.setCustomAction(
  new CustomRecordedAction("create_row", async () => {
    await createRow(varsComp, row++, column);
    system.updateDirty();
  })
);
ScriptRecorder.setCustomAction(
  new CustomRecordedAction("create_column", async () => {
    await createColumn(varsComp, row, column++);
    system.updateDirty();
  })
);

// When a user click on one of the buttons, play corresponding custom
   ↪ action
document.getElementById("createRow").onclick = () => {
  system.recorder.playCustomAction("create_row");
};
document.getElementById("createCol").onclick = () => {
  system.recorder.playCustomAction("create_column");
};
```

Listing 4.7: Register custom actions and setup listeners to perform the custom actions.

```
component v0_0 {
    var &v1_1, &v0_0, code;
    constraint {
        (v1_1 -> v0_0) => v1_1 * 2;
    }
}
```

Listing 4.8: An example dynamic component in the spreadsheet application. Variable `v0_0` holds the value of the cell found in the first row, first column, and `v1_1` does the same for the cell found in the second row, second column.

In HotDrink, multiple variable references can point to the same variable. The *context component* of a constraint system is a single component which owns variables used by multiple components. A context component, such as Listing 4.9, does not have any constraints as its sole purpose is to be a centralized place to access shared variables. To work in a dynamic constraint system, the context component must be created at initialization. In the spreadsheet application, the context component stores the values of the cells. Any dynamically added components can update the value of any cell by referencing its corresponding variable in the context component. The system must be robust enough to detect changes in the constraint system and reflect them in the view. With a context component, we can simply bind a variable in the component to a view widget, thus gaining this detection almost for free.

```
component vars {
  var v0_0, v0_1, v0_2;
  var v1_0, v1_1, v1_2;
  var v2_0, v2_1, v2_2;
}
```

Listing 4.9: The context component of the spreadsheet application after initialization for a 3×3 spreadsheet.

Chapter 5

Related Work

This chapter consists of two sections. The first discusses native scripting capabilities in GUIs, while the second discusses external automation, such as Robotic Process Automation (RPA).

5.1 Scriptable GUIs

In this section we give some examples of applications with internal scripting capabilities. These capabilities usually manifest themselves as either scripts written by the user or recordings of user actions; a few provide both.

5.1.1 GNU Image Manipulation Program

GNU Image Manipulation Program (GIMP) is a widely used program for manipulating images. It does not support automatically recording users' actions, but it has several other methods to automate workflows. Out-of-the-box GIMP supports Script-Fu, which is a Scheme-based scripting environment [7]. Script-Fu is designed to automate actions the user frequently performs or tasks that are hard to remember.

Script-Fu supports calling other custom scripts, which allows for complex behaviour to be expressed with minimal code. Realistically, however, only those who can program in Scheme are able to effectively create their own scripts. More widely used programming languages [15] are supported as plugins, including Python [6], Perl [38], and C# [8].

One can observe that scripting in Script-Fu is targeted for expert users with programming experience. Once a script has been created, however, anyone can use it. Listing 5.1 gives an example Script-Fu script. When the script is ran, it will first display a configuration GUI, Figure 5.1, before performing the rest of the script. A result of running the script can be seen in Figure 5.2.

```
(script-fu-register
  "script-fu-remove-bg"                         ;func name
  "Remove Background and Crop"                  ;menu label
  "Remove background and crop"                  ;description
  "kheb"                                        ;author
  "The Unlicense"                               ;copyright notice
  "26. Aug. 2021"                               ;date created
  ""                                            ;image type that the script works on
  SF-IMAGE         "Image"     0                ;image to work on
  SF-DRAWABLE      "Layer"     0                ;layer to work with
  SF-COLOR         "Color"     '(255 255 255)   ;color variable
)
(script-fu-menu-register "script-fu-remove-bg" "<Image>/Image")
(define (script-fu-remove-bg image drawable color)
(let*
    (
      (width (car (gimp-drawable-width drawable)))
      (height (car (gimp-drawable-height drawable)))
    )
    (gimp-undo-push-group-start image)
    (gimp-layer-add-alpha drawable) ;Add alpha channel (otherwise the deleting of pixels is set to bg color)

    (gimp-image-select-color image CHANNEL-OP-REPLACE drawable color) ;Select all pixels of the given color
    (gimp-drawable-edit-clear drawable) ; Delete (i.e. make transparent) selected pixels


    (gimp-drawable-update drawable 0 0 width height) ; Update all pixels
    (gimp-selection-none image) ;Deselect all

    (plug-in-autocrop RUN-NONINTERACTIVE image drawable) ; Crop to content

    (gimp-undo-push-group-end image)
  )
)
```

Listing 5.1: A Script-Fu script that replaces the background of an image with transparent color, and then auto-crops the image to remove excess borders.

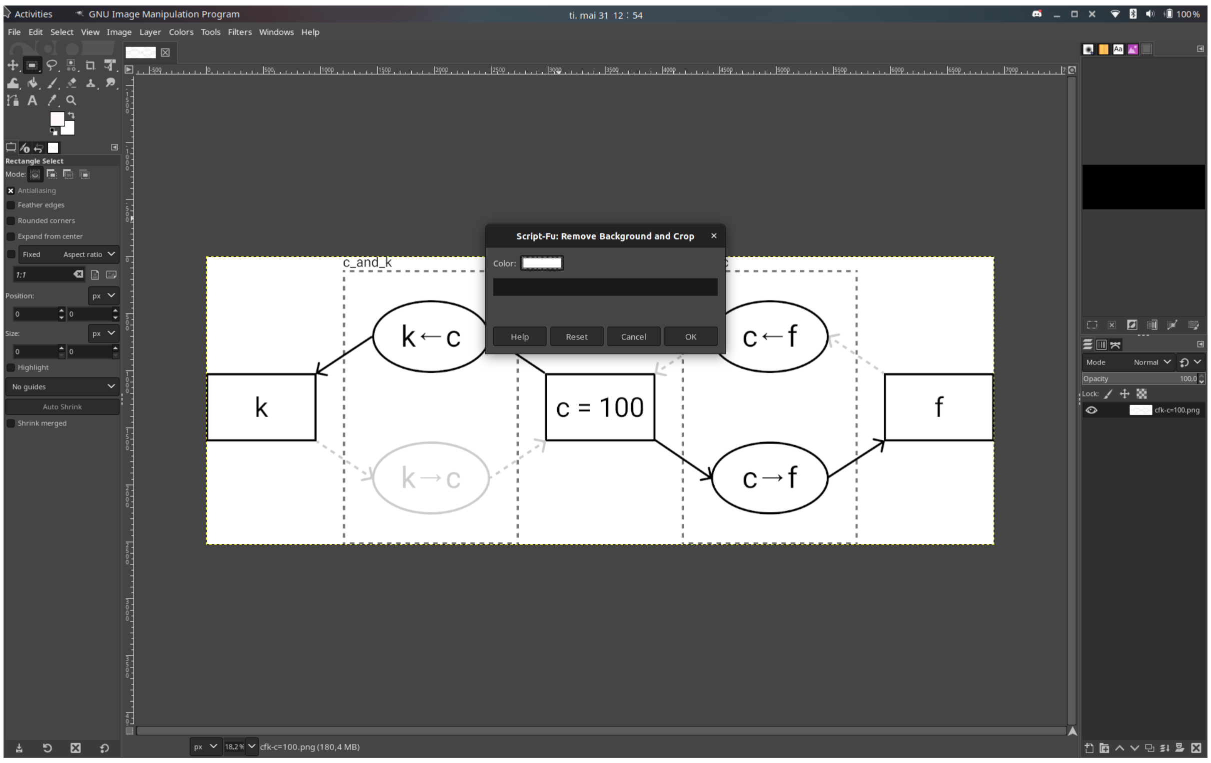


Figure 5.1: Script-Fu dialogue box to select background colour to remove.

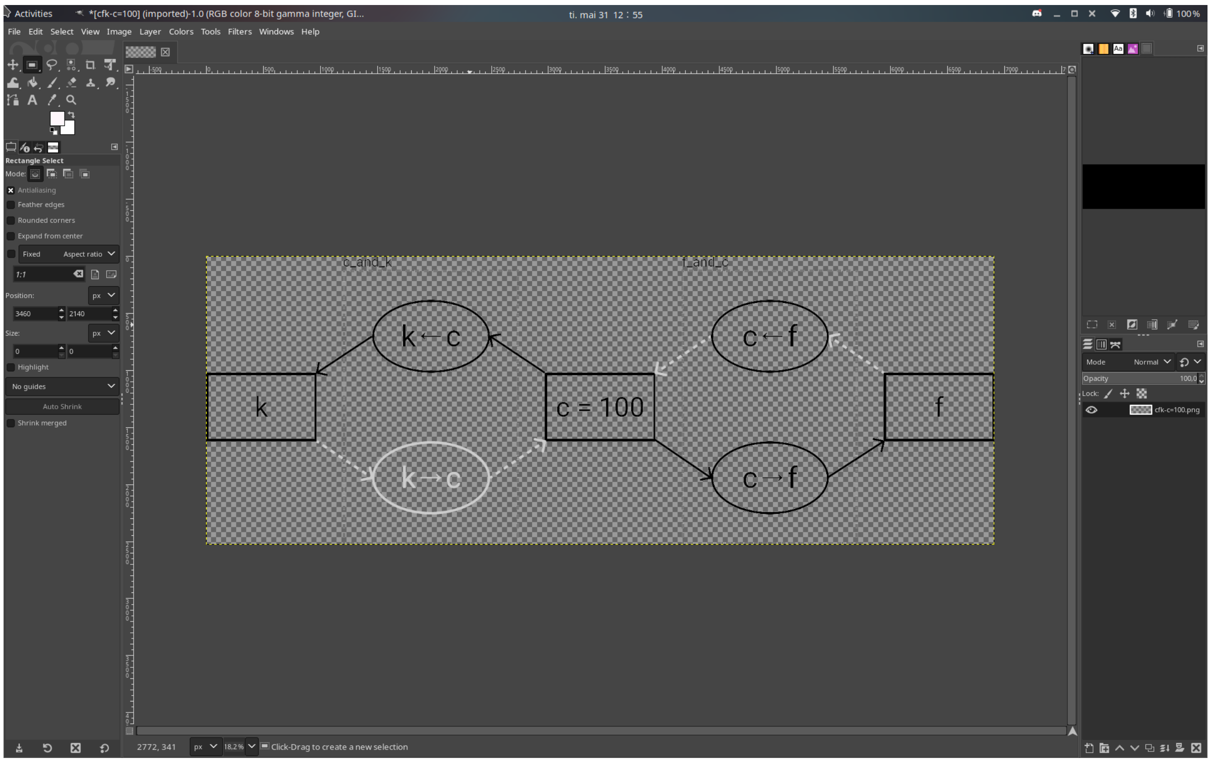
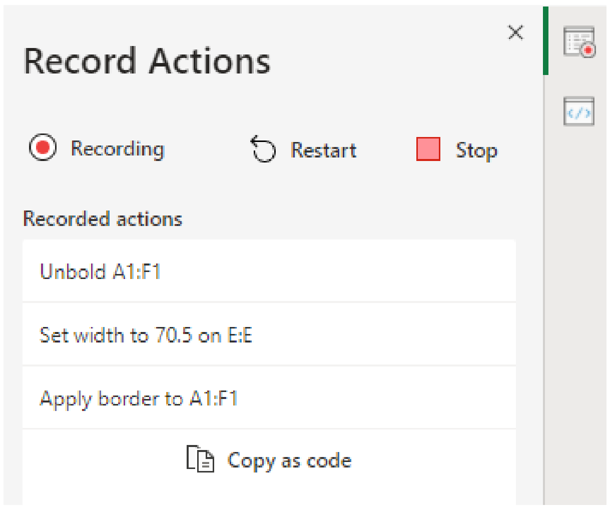


Figure 5.2: Result of running the script from Listing 5.1.

5.1.2 Microsoft Word and Excel

Office Script in Microsoft Excel is an example of a user-friendly scripting platform. An example of its user-friendliness is the option to add a shortcut button to run a recording quickly. Microsoft Excel allows users to record their actions with a built-in “Action Recorder”. There is no need to write code to create and use Office Script; however, more advanced users may edit the result of a recording as a TypeScript [28] function in the built-in code editor [5]. A “Record Action” side-panel can be seen in Figure 5.3. It displays the current recorded actions and has widgets to control the action recorder. Office Script lacks support for any platform other than Microsoft Excel on the web.

For desktop, Microsoft products support Visual Basic for Applications (VBA) macros. This macro system is similar in design to Office Script: allowing non-technical users to record actions they have performed, then play them back over different data. Feature-wise, both Office Script and VBA macros are largely identical. The major exception is that VBA macros can run when an event is triggered, while Office Script in Microsoft Excel programs must be started manually [13]. Figure 5.4 shows how to define a “hello world”-style macro that will be triggered when a command-button is clicked by a user.



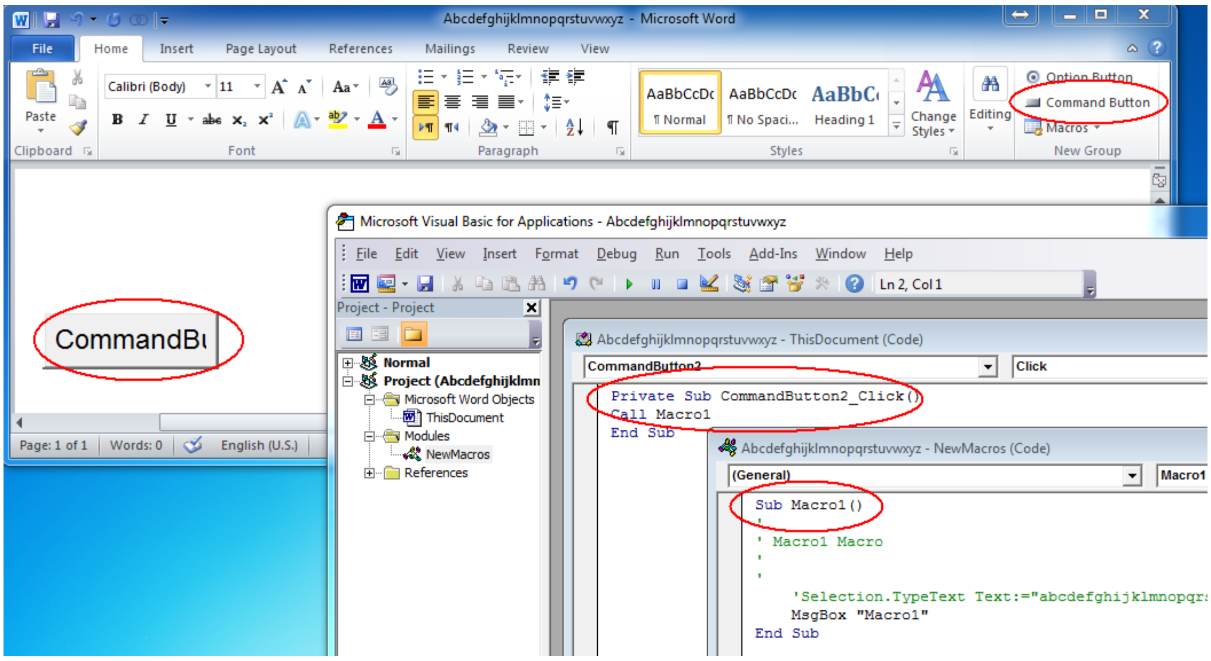
Recorded actions

Unbold A1:F1

Set width to 70.5 on E:E

Apply border to A1:F1

Figure 5.3: Action recorder from Office Script in Microsoft Excel [5].



Abcdefghijklmnopqrstuvwxyz - ThisDocument (Code)

CommandButton2 | Click

```
Private Sub CommandButton2_Click()
Call Macro1
End Sub
```

Abcdefghijklmnopqrstuvwxyz - NewMacros (Code)

(General) | Macro1

```
Sub Macro1()
'
' Macro1 Macro
'
'
    'Selection.TypeText Text:="abcdefghijklmnopqr
    MsgBox "Macro1"
End Sub
```

Figure 5.4: Example of executing a VBA macro triggered by a graphical button click [29].

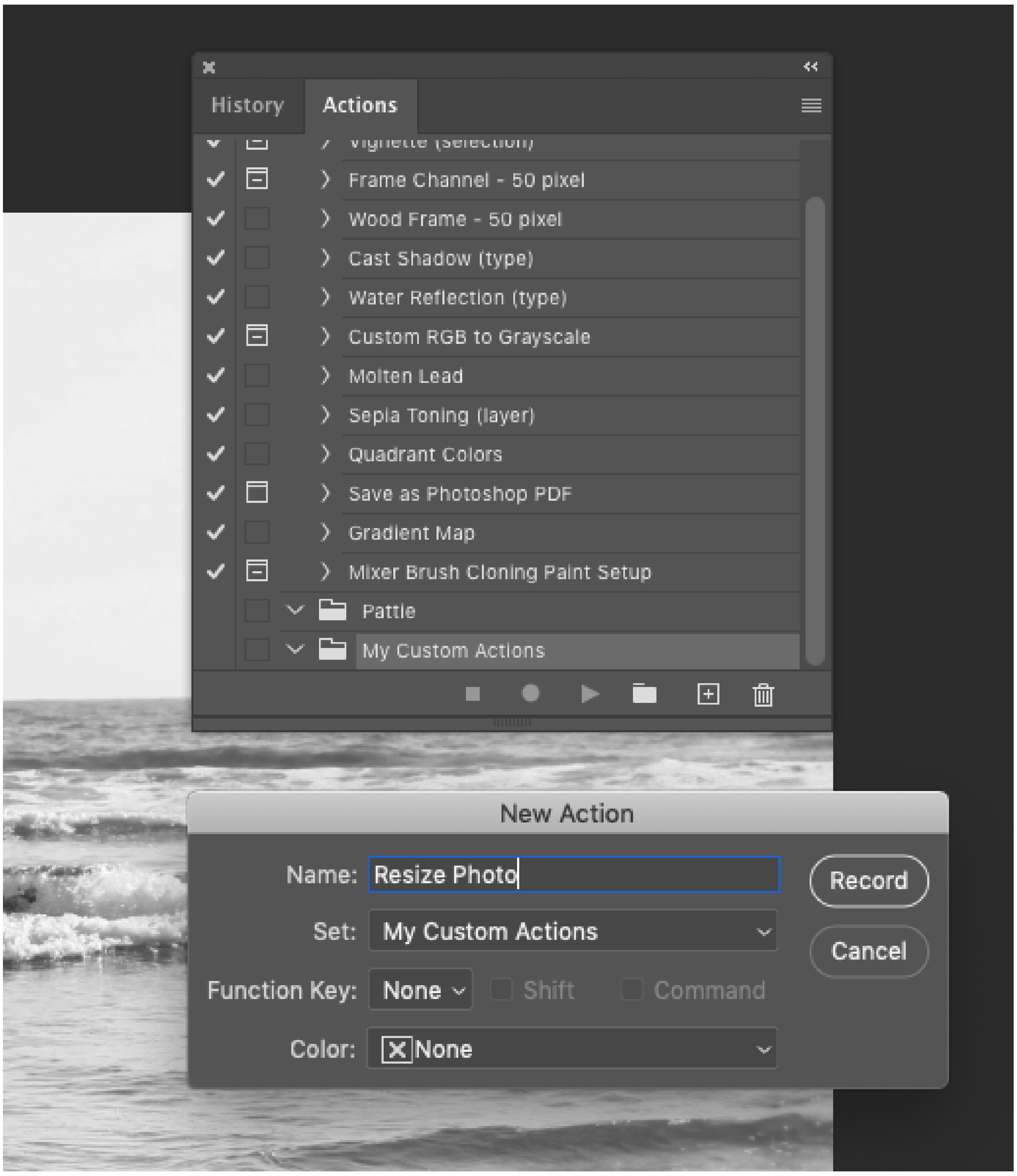
5.1.3 Adobe Suite

Adobe Photoshop offers two ways to automate tasks: the basic *Photoshop Action* and a more feature-rich *Photoshop Scripting*. Photoshop Action allows the user to repeat a set of actions with a built-in recorder, requiring no programming to create a script. The GUI for this recorder is shown in Figure 5.5. An action script may also include actions which cannot be recorded, such as conditional statements, giving the user a decent range of actions, yet with little programming needed. The drawback of Photoshop Action is that it cannot handle files (e.g., reading, writing, renaming) nor can action scripts be transferred to other computers.

Photoshop Scripting can do everything Photoshop Actions can do and more. However, scripts must be programmed and not recorded. Creating scripts requires potential users to be somewhat proficient in programming. Photoshop Scripting supports multiple languages: AppleScript on macOS, Visual Basic on Windows, and JavaScript on both Windows and macOS. An example script from the documentation [17] of Photoshop Scripting is given in Listing 5.2. Scripts are easily shared between different computers, and they work on other Adobe applications too, such as Adobe Illustrator.

```
// Create 2 documents
var docRef = app.documents.add(4, 4);
var otherDocRef = app.documents.add(4, 6);
//make docRef the active document
app.activeDocument = docRef;
//here you would include command statements
//that perform actions on the active document. Then, you could
//make a different document the active document
//use the activeDocument property of the Application object to
//bring otherDocRef front-most as the new active document
app.activeDocument = otherDocRef;
```

Listing 5.2: An example of Photoshop Scripting in JavaScript which demonstrates how to set the active document [17].



Name: Resize Photo
Set: My Custom Actions
Function Key: None ☐ Shift ☐ Command
Color: ☒ None

Figure 5.5: Create new Photoshop Action dialogue [30].

5.1.4 SAP GUI Scripting

SAP GUI is a user interface created for interacting with the various SAP applications; there are three main clients, SAP GUI Windows, SAP GUI Java, and SAP GUI HTML [22]. SAP GUI Scripting emulates the user interacting with the GUI. That is, a script can only do what the user would be able to do, just faster. It can be used as a simple macro recording and playback tool, used for testing, or as a part of a more advanced automation suite.

A developer is able to access individual fields in the GUI by specifying its *field-id* (i.e., path to a unique field). Scripts are often version dependent because a simple change in the GUI might have changed the field-id of an object. As shown in Figure 5.6, there are various utility tools bundled with the GUI to help users automate their workflow, such as finding the field-id an object. Another bundled utility tool is the script recorder, which is able to create scripts. Scripts can also be created by writing them manually in VBA, JScript, JavaScript, C#, or C++. The recorded scripts are serialized to one of these languages, depending on which GUI is used. The scripting is also the basis of all other automation suites, such as performance monitoring, load testing, and functional testing.

By default, scripting is disabled on the server-side; this feature must be explicitly enabled. If scripting is enabled, the system-admins have fine-grained control on how scripts behave, such as only allowing certain groups to have access to scripts or disallowing scripts to change any values in the system. There also appears to have been a marketplace for selling scripts or script services, but it is now gone from the web and not archived [33].

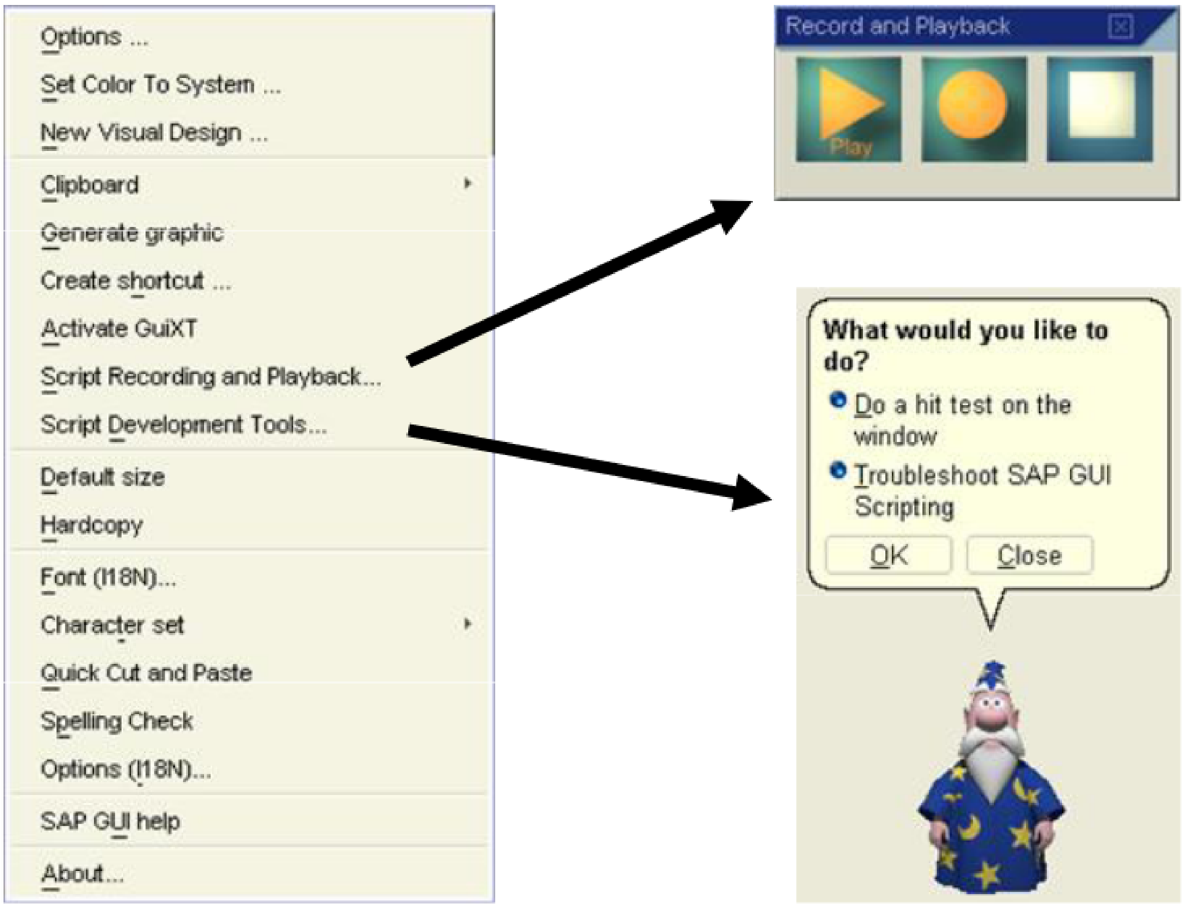


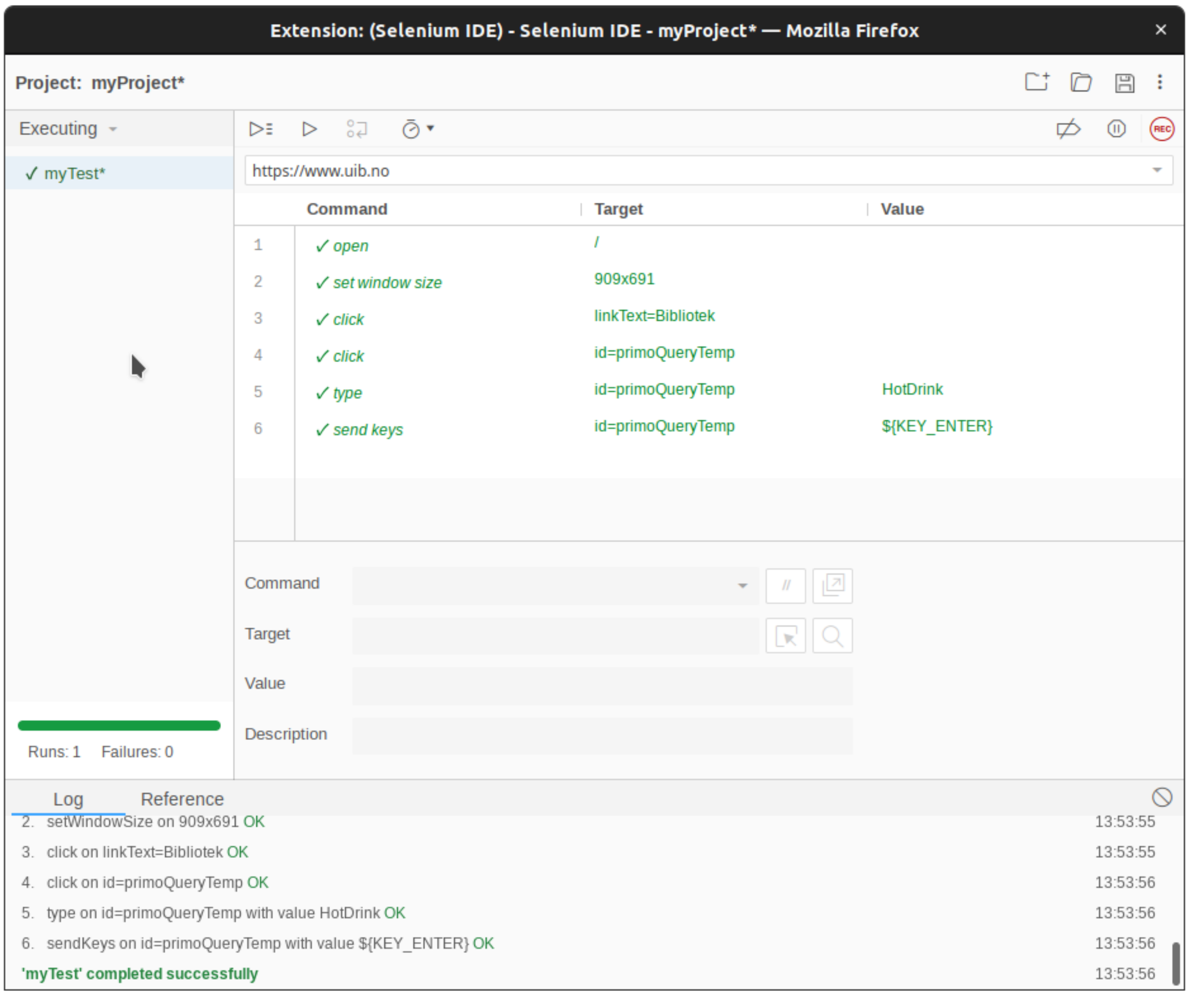
Figure 5.6: Built-in Scripting Utilities in SAP GUI [22].

5.2 External Automation

GUIs with internal scripting capability is not the only way to enable task automation. Another common approach is to programmatically emulate how a human interacts with a GUI. This approach is called Robotic Process Automation (RPA) [43]. RPAs can be used to create automated tests of a GUI or automate repetitive tasks. Compared to internal scripting, the benefit of RPA is universal availability, as the application does not need to support scripting itself [42].

5.2.1 Selenium

An example of RPA is Selenium [24]. It enables automation of all the major web browsers, both programmatically [26] and by recording user input [25]. The Selenium Integrated Development Environment (IDE) has an advanced interface for users to edit their recorded actions, showcased in Figure 5.7. Unlike most other scripting suites, there is no actual programming involved to edit the recorded action, thus it is easier for a novice user to automate a task. The recorded actions can be exported as a unit test to multiple different testing frameworks, e.g., Java JUnit [11], Python pytest [18], and JavaScript Mocha [14].


Extension: (Selenium IDE) - Selenium IDE - myProject* — Mozilla Firefox
Project: myProject*
Executing
✓ myTest*
Runs: 1 Failures: 0
Log
Reference

https://www.uib.no

	Command	Target	Value
1	✓ open	/	
2	✓ set window size	909x691	
3	✓ click	linkText=Bibliotek	
4	✓ click	id=primoQueryTemp	
5	✓ type	id=primoQueryTemp	HotDrink
6	✓ send keys	id=primoQueryTemp	${KEY_ENTER}

Command
Target
Value
Description

```
2.  setWindowSize on 909x691 OK                                          13:53:55
3.  click on linkText=Bibliotek OK                                       13:53:55
4.  click on id=primoQueryTemp OK                                        13:53:56
5.  type on id=primoQueryTemp with value HotDrink OK                     13:53:56
6.  sendKeys on id=primoQueryTemp with value ${KEY_ENTER} OK             13:53:56
'myTest' completed successfully                                          13:53:56
```

Figure 5.7: The Selenium IDE with a test to search for HotDrink in the University of Bergen Library.

5.2.2 AppleScript

In Apple's macOS, there is a built-in scripting language with an English-like syntax called AppleScript [34]. Most notably, AppleScript allows scripts to be created for any application in macOS. An example AppleScript script is given in Listing 5.3. It counts the number of files in a folder. Application developers can respond to various "Apple events" [1] triggered by user scripts. Handling these events allows end-users to automate their workflow with any application using AppleScript. However, applications must be programmed to respond to these events. Even if there is no-to-little support for Apple events within an application, users are still able to create certain types of scripts, called GUI scripts. This scripting method is based on the macOS accessibility framework, which allows a script to send system events. These system events allow the user to interact with an application as if they were using their mouse and keyboard, i.e., pressing buttons, accessing menus, and entering text into text fields [3]. The official AppleScript editor [2] has the ability to record the actions of a user to AppleScript instructions. It simplifies the creation of an initial draft, making it easier to adjust the recording to the intended automation workflow.

```
tell application "Finder"
    if folder "Applications" of startup disk exists then
        return count files in folder "Applications" of startup disk
    else
        return 0
    end if
end tell
```

Listing 5.3: A script for counting the number files in the Applications folder [23].

5.3 Summary

As can be seen by inspecting commonly used commercial applications, some form of scripting is available to end-users. The examples above, however, are cherry-picked for their scripting capabilities.

There are two main approaches to automate a task in an application, as can be seen in the provided examples. The first approach is to write a script that interacts with the application directly, which requires some programming knowledge from the end-user. Script-Fu is an excellent example of this approach. The second approach is to record a user's actions, making automation accessible to non-technical end-users. An example of this is Photoshop Action. This second approach might not necessarily allow the user to edit the recorded actions, but most scriptable software seems to allow it. Exactly how the actions are presented to the user vary from application to application, but most commonly this is as a text based script. Another way to display recorded actions to the user is via a GUI, which Selenium does.

Chapter 6

Evaluation

We implemented the image resize application described in Section 3.1 both with and without HotDrink and HDScript. First we describe how the version without HotDrink was implemented, then we compare the two implementations.

6.1 HotDrink-less Image Resize Application

An application with the identical feature set as the image resize application, detailed in Section 3.1, was created to evaluate how an alternative approach could replicate scripting on a per-application basis. This new application is written in plain JavaScript, without HotDrink or other external frameworks. We call it the *HotDrink-less image resize* evaluation application. It is pictured in Figure 6.1. The HotDrink-less image resize application, like the version using HotDrink, has four *system variables* that define the image size: absolute width, absolute height, relative (to the initial width) width, and relative (to the initial height) height. As we do not have HotDrink to handle the relationships between variables in the application, the relationships must manually be updated each time any of the system variables change.

The evaluation application has internal automation capabilities with record and playback functionality. This, reasonably, a developer can be expected to readily create, as in Listing 6.1. A recorder, at its simplest, is a list of functions, states, or other structures capable of acting as a proxy to the user interacting with the application. In the case of Listing 6.1, the recorder is implemented as an ordered list of functions paired with their

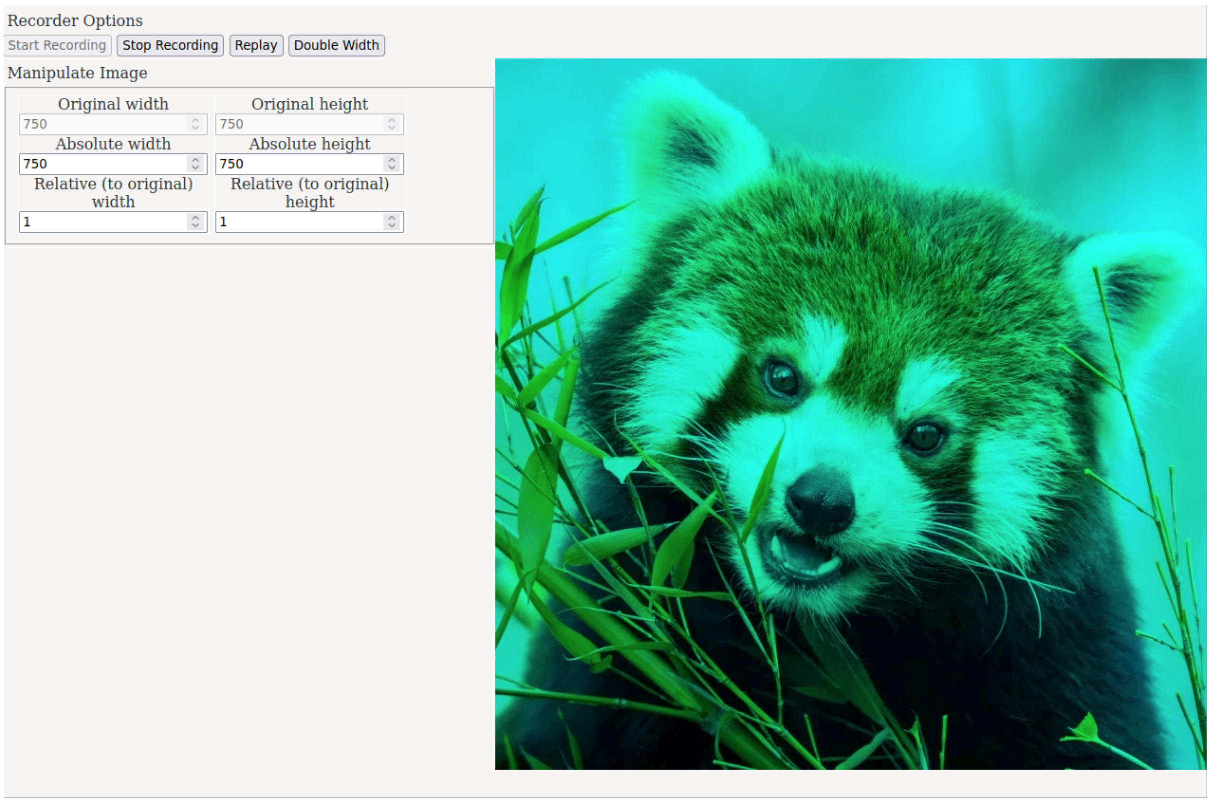


Figure 6.1: The HotDrink-less image resize evaluation application is visually similar to the version with HotDrink described in Section 3.1.

arguments. That is, to record an action, we call the *record* function with the function to record and its arguments. When the recorder is replayed, the application iterates over the recorded functions and executes each with their stored arguments.

In the HotDrink-less image resize application we record an action when any of the system variables change. A system variable changes, for example, when a user updates one of the input fields in the GUI. As listed in Listing 6.2, each action consists of two steps: first the relevant system variable's value is updated, then the values of all other system variable which depend on the relevant system variable are recalculated. These two steps are recorded as a single action in the record function from Listing 6.1.

One should note that script recording cannot be implemented by just storing the snapshots of the full application state. If we were to do so, we would not have achieved automation, but rather persistence. There would then be no need to store more than the latest state of the application, as the older states would be overwritten by newer states. In other words, both steps of the recorded actions, setting the new value and updating dependent variables, are required to enforce invariants of the system when an action is replayed.

```
const funcs = [];
let recording = true;

function startRecording() { recording = true; }
function stopRecording() { recording = false; }

/** Record (if recording) and call the function with the given arguments **/
function record(func, ...args) {
  if (recording) {
    funcs.push({ func, args });
  }
  func(...args);
}

/** Replay recorded actions **/
function replay() {
  let wasRecording = recording;
  stopRecording();
  for (let action of funcs) {
    action.func(...action.args);
  }
  recording = wasRecording;
}
```

Listing 6.1: Generic record and replay functionality without HDScript.

```
record((relX) => {
  // First step of the action: update the value of the variable
  setRelWidth(relX);

  // Second step: update dependent variables
  //  I.e., when the relative width is updated, so should the absolute width
  setAbsWidth(getInitWidth() * relX);
}, getRelWidth()); // take snapshot of new value
```

Listing 6.2: Using the generic record function from Listing 6.1 to record an action in the HotDrink-less image resize application.

6.2 Comparison

```
record(() => {
  const relX = 2 * getRelWidth();
  const absX = 2 * getAbsWidth();
  setAbsWidth(getInitWidth() * relX);
  setRelWidth(absX / getInitHeight());
});
```
Listing 6.3: A modification action that doubles the width of the image of the evaluation application.

The evaluation application falls short compared to the same application written with HDScript when more advanced features than record and replay functionality are desired. For example, in order to edit a recording, each action within the recorder must be transparent enough to be modifiable. In the version without HotDrink, actions are opaque, i.e., JavaScript functions, thus infeasible for a user to alter. The only way to increase the number of alternative actions is to create and record a new function, as is done in Listing 6.3. An alternative solution to edit actions is to serialize the recordings as a script, similar to our approach in HDScript. However, in that case, the view developers would either need to design, implement, and maintain a DSL—which is not an entirely trivial task compared to as creating a recorder—or use an existing scripting language. No matter the option chosen, the development cost will likely be significant. In addition, the developers must create a system to interpret scripts, and apply each statement in the script to the application's current state, i.e., the system must act as if interpreting a script is replaying a set of recorded actions. Even the simple GUI evaluation application shows that implementing scripting without proper support from a framework is practically a non-starter.

Chapter 7

Conclusion and Future Work

As part of this thesis work, we explored several applications with varying degree of GUI automation capabilities. These applications are proof that GUI automation is clearly useful. All of the automation features in the applications we studied were quite different, built specifically for one application only. This thesis questions whether a more general approach to GUI automation would be feasible. In a generic approach, the underlying automation code can be reused in multiple different applications. This should make the benefit of GUI automation outweigh its costs more broadly, and lower the barrier to add GUI automation features to more applications.

We demonstrated how automation can indeed be built as a part of a GUI framework. The HDScript extension of the HotDrink framework enables any application created with it to have internal GUI automation capabilities. We showed that the cost of implementing automation is reduced, and in some cases eliminated entirely, when applications are created with HDScript. In particular, we created a recorder tool for inexperienced users and a DSL for more experienced users to automate any application created with HotDrink. Additionally, we explored how users without any prior programming experience are able to express their intent when using the action recorder. With the extended framework, we implemented several case studies to evaluate changes needed to support automation and scripting in applications. While the tests were rudimentary, they showed that a generic approach to GUI automation can be made to work out-of-the-box, sometimes without any modification to the applications themselves. This is an early step in the line of GUI automation research, there is plenty of work to make scripting a standard feature of run-of-the-mill GUIs. We see immediate avenues to improve usability for end-users. Below, we include several suggestions for improvement.

Additional intent recognizers For the intent recognition subsystem to be effective it should be able to operate on as many data types as possible. Currently, the subsystem can only recognize intents of JavaScript's primitive data types. The overall quality of the intent recognition subsystem will continually improve as intent recognizers for new data types are implemented. To this end, a future objective could be to construct intent recognizers for all data types in the standard JavaScript library where an intent recognizer is suitable. For example, an intent recognizer for Date [9] could suggest the current time or date.

IDE for editing scripts It is currently up to the view to implement where the user writes their scripts. This is not ideal and can be improved by providing a standardised IDE for writing scripts in the view. To HotDrink, however, views are interchangeable. Thus, we must potentially write an IDE for each platform that different views are using (e.g., web, spreadsheet). The Selenium IDE [25] could be a starting point to provide inspiration for what features users might expect.

More structural actions Currently only HotDrink components can be created or removed with HDScript. The DSL could be expanded, and actions created, to more easily alter the internal structure (i.e., constraints, methods, or variables) of components. Listing 7.1, Listing 7.2, and Listing 7.3 give suggestions for additional actions as DSL statements.

```
// Add a new method called ‘‘method_id’’ to the constraint
//  ‘‘constraint_id’’ in the component ‘‘comp_id’’.
#addMethod comp_id constraint_id method_id <<method>>;

// Remove the method defined above
#removeMethod comp_id constraint_id method_id;
```

Listing 7.1: Potential syntax to manipulate methods. The «method» expression should define input and output variables of the method, together with its actual function. The syntax for «method» could be borrowed from HotDrink's DSL e.g., `(a -> b) => a * 2`.

```
//Remove the constraint ‘‘constraint_id’’ in the component ‘‘comp_id’’
#removeConstraint comp_id constraint_id;
```

Listing 7.2: Suggested syntax to remove a constraint.

```
//Add a new variable with the name ‘‘variable_id’’ in the component
    ↪ ‘‘comp_id’’
#addVariable comp_id variable_id;

//Remove the variable ‘‘variable_id’’ in the component ‘‘comp_id’’
#removeVariable comp_id variable_id;
```

Listing 7.3: Suggested syntax to add and remove variables in a component.

Glossary

API Application Programming Interface.

AST Abstract Syntax Tree.

DSL Domain Specific Language.

GIMP GNU Image Manipulation Program.

GUI Graphical User Interface.

IDE Integrated Development Environment.

MVVM Model–View–View-Model.

RPA Robotic Process Automation.

UI User Interface.

VBA Visual Basic for Applications.

Bibliography

[1] About AppleScript.
URL: https://developer.apple.com/library/archive/documentation/AppleScript/Conceptual/AppleScriptX/Concepts/ScriptingOnOSX.html#//apple_ref/doc/uid/20000032-BABEBGCF. [Accessed on *2021-11-30*].

[2] Mac Automation Scripting Guide: Getting to Know Script Editor.
URL: https://developer.apple.com/library/archive/documentation/LanguagesUtilities/Conceptual/MacAutomationScriptingGuide/GettoKnowScriptEditor.html. [Accessed on *2021-11-30*].

[3] AppleScript: Graphic User Interface (GUI) Scripting.
URL: https://www.macosxautomation.com/applescript/uiscripting/index.html. [Accessed on *2021-11-30*].

[4] ECMAScript—ECMA-262.
URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-262/. [Accessed on *2022-05-11*].

[5] Office Scripts in Excel on the web—Office Scripts.
URL: https://docs.microsoft.com/en-us/office/dev/scripts/overview/excel. [Accessed on *2021-11-26*].

[6] The "Python-Fu" Submenu.
URL: https://docs.gimp.org/en/gimp-filters-python-fu.html. [Accessed on *2021-12-01*].

[7] Using Script-Fu Scripts.
URL: https://docs.gimp.org/en/gimp-concepts-script-fu.html. [Accessed on *2021-11-26*].

[8] GIMP #.
URL: http://gimp-sharp.sourceforge.net/index.html. [Accessed on *2021-11-26*].

[9] Date—JavaScript | MDN.
URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Date. [Accessed on *2022-05-20*].

[10] Primitive—MDN Web Docs Glossary: Definitions of Web-related terms | MDN.
URL: https://developer.mozilla.org/en-US/docs/Glossary/Primitive. [Accessed on *2022-05-20*].

[11] JUnit 4.
URL: https://junit.org/junit4/. [Accessed on *2022-01-21*].

[12] The Programming Language Lua.
URL: https://www.lua.org/. [Accessed on *2022-05-11*].

[13] Differences between Office Scripts and VBA macros—Office Scripts.
URL: https://docs.microsoft.com/en-us/office/dev/scripts/resources/vba-differences. [Accessed on *2021-12-01*].

[14] Mocha.
URL: https://mochajs.org/. [Accessed on *2022-01-21*].

[15] Stack Overflow Developer Survey 2021.
URL: https://insights.stackoverflow.com/survey/2021#most-popular-technologies-language. [Accessed on *2021-11-25*].

[16] Language variables basics.
URL: https://www.php.net/manual/en/language.variables.basics.php.

[17] Adobe Photoshop Scripting Guide.
URL: https://github.com/Adobe-CEP/CEP-Resources/blob/71f7620966141ac144f7e85f70d3f27a84e1f7fb/Documentation/Product%20specific%20Documentation/Photoshop%20Scripting/photoshop-scripting-guide-2020.pdf. [Accessed on *2021-11-25*].

[18] Pytest.
URL: https://docs.pytest.org/en/6.2.x/. [Accessed on *2022-01-21*].

[19] Python.
URL: https://www.python.org/. [Accessed on *2022-05-11*].

[20] General Python FAQ.
URL: https://docs.python.org/3/faq/general.html#id4. [Accessed on *2022-05-13*].

[21] Design Principles – React.
URL: https://reactjs.org/docs/design-principles.html. [Accessed on *2022-03-13*].

[22] SAP GUI Scripting.
URL: https://wiki.scn.sap.com/wiki/display/ATopics/SAP+GUI+Scripting. [Accessed on *2022-01-27*].

[23] Scripting with AppleScript.
URL: https://developer.apple.com/library/archive/documentation/AppleScript/Conceptual/AppleScriptX/Concepts/work_with_as.html#%2F%2Fapple_ref%2Fdoc%2Fuid%2FTP40001568=. [Accessed on *2022-04-23*].

[24] Selenium.
URL: https://www.selenium.dev/. [Accessed on *2021-11-25*].

[25] Selenium IDE.
URL: https://selenium.dev/selenium-ide/index.html. [Accessed on *2021-11-25*].

[26] Selenium WebDriver.
URL: https://www.selenium.dev/documentation/webdriver/. [Accessed on *2021-11-25*].

[27] Javascript—Getting All Variables In Scope—Stack Overflow.
URL: https://stackoverflow.com/questions/2051678/getting-all-variables-in-scope. [Accessed on *2021-12-13*].

[28] TypeScript.
URL: https://www.typescriptlang.org/. [Accessed on *2021-11-26*].

[29] Ms Office—Word Template : Executing VBA macro on a graphical button click.
URL: https://stackoverflow.com/questions/18174819/word-template-executing-vba-macro-on-a-graph
[Accessed on *2022-04-23*].

[30] Quick Steps! Create an action now to save time.
URL: https://community.adobe.com/t5/photoshop-ecosystem-discussions/quick-steps-create-an-action-now-to-save-time/m-p/10844134#M358390. [Accessed on *2022-04-23*].

[31] David William Barron. *The World of Scripting Languages*. The Worldwide series in Computer Science. John Wiley & Sons Inc, 1 edition, August 2000. ISBN 9780471998860.

[32] Daniel Berge. Seven tasks implemented using HotDrink.
URL: https://github.com/DanielBerge/7-tasks-hotdrink/blob/f8fb13d27c4feeecf6d200a01d2aec863920b7b6/counter/counter.js. [Accessed on *2022-04-27*].

[33] Christian Cohrs and Gisbert Loff. The SAP GUI Scripting API How to Automate User Interaction—Technology, Examples and Integration.
URL: http://www.sdn.sap.com/irj/scn/go/portal/prtroot/docs/library/uuid/80aaac18-2dfe-2a10-bbb1-ec9b3760ea4c.

[34] William R. Cook. AppleScript. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pages 1–1–1–21, New York, NY, USA, June 2007. Association for Computing Machinery. ISBN 9781595937667. doi: 10.1145/1238844.1238845.
URL: https://doi.org/10.1145/1238844.1238845.

[35] John Freeman, Jaakko Järvi, and Gabriel Foust. HotDrink: a library for web user interfaces. In *Proceedings of the 11th International Conference on Generative Programming and Component Engineering—GPCE '12*, page 80, Dresden, Germany, 2012. ACM Press. ISBN 9781450311298. doi: 10.1145/2371401.2371413.
URL: http://dl.acm.org/citation.cfm?doid=2371401.2371413.

[36] Bjorn N. Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990. ISSN 0001-0782. doi: 10.1145/76372.77531.
URL: https://doi.org/10.1145/76372.77531. [Accessed on *2022-03-25*].

[37] John Gossman. Introduction to Model/View/ViewModel pattern for building WPF apps—Tales from the Smart Client—Site Home—MSDN Blogs.
URL: https://web.archive.org/web/20100601093702/http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx. [Accessed on *2022-01-19*]. Archived from the original http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[38] Ed J. GIMP—A Tutorial for GIMP-Perl Users.
URL: https://www.gimp.org/tutorials/Basic_Perl/. [Accessed on *2021-12-01*].

[39] Jaakko Järvi. HotDrink git repository.
URL: https://git.app.uib.no/Jaakko.Jarvi/hd4. [Accessed on *2022-05-20*].

[40] Jaakko Järvi, Mat Marcus, Sean Parent, John Freeman, and Jacob N. Smith. Property models: from incidental algorithms to reusable components. In *Proceedings of the 7th international conference on Generative programming and component engineering*, GPCE '08, pages 89–98, New York, NY, USA, October 2008. Association for Computing Machinery. ISBN 9781605582672. doi: 10.1145/1449913.1449927.
URL: https://doi.org/10.1145/1449913.1449927.

[41] Michael Sannella. Skyblue: a multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th annual ACM symposium on User interface software and technology*, UIST '94, pages 137–146, New York, NY, USA, November 1994. Association for Computing Machinery. ISBN 9780897916578. doi: 10.1145/192426.192485.
URL: https://doi.org/10.1145/192426.192485.

[42] UiPath. The Impact of RPA on Employee Experience.
URL: https://www.uipath.com/resources/automation-analyst-reports/forrester-employee-experience-rpa.

[43] Wil M. P. van der Aalst, Martin Bichler, and Armin Heinzl. Robotic Process Automation. *Business & Information Systems Engineering*, 60(4):269–272, August 2018. ISSN 2363-7005, 1867-0202. doi: 10.1007/s12599-018-0542-4.
URL: http://link.springer.com/10.1007/s12599-018-0542-4. [Accessed on *2021-11-25*].

Appendix A

Formal HDScript DSL Syntax

We define the formal syntax of HDScript's DSL in Extended Backus-Naur Form (EBNF) with Listing A.1.

```
digit =  "0".."9" ;
letter =  "a".."z" | "A".."Z" ;
letter_plus =  letter | "_" | "$" ;
dig_let =  digit | letter_plus ;
whitespace =  " " | "\t" | "\n" | "\r" ;
identifier =  letter_plus { dig_let } ;
var_ref =  { whitespace } identifier "." identifier { whitespace } ;
var_ref_list =  var_ref { "," { whitespace } var_ref } ;

# For js code essentially allow all ASCII characters
js_code =  { whitespace } { " ".."|" } { whitespace } ;

assign_constant =  var_ref "=" js_code ;
assign_non_constant =  var_ref ( "-" | "+" | "*" | "/" | "|" | "&" |
   ↪ "%" | "^" "||" | "&&" | ">>>" | ">>" | "<<" | "**" | "??" ) "="
   ↪ js_code ;
assign_left_to_right_value =  var_ref "=" var_ref ;
assign_left_to_right_variable =  var_ref "=&" var_ref ;
add_component =  { whitespace } "#AddComponent" js_code ;
remove_component =  { whitespace } "#RemoveComponent" js_code ;
schedule_command =  { whitespace } "(" var_ref_list "->" var_ref_list
   ↪ ")" js_code ;
custom_action = { whitespace } "#custom" { whitespace } identifier {
   ↪ whitespace };

line =  assign_constant | assign_non_constant |
   ↪ assign_left_to_right_value | assign_left_to_right_variable |
   ↪ add_component | remove_component | schedule_command |
   ↪ custom_action ";" ;
script =  { line } ;
```

Listing A.1: Formal syntax of HDScript's DSL in EBNF.