ClickScript

a visual programming language in the browser

Master Thesis

Systems Group

June 8, 2009 - October 5, 2009

Lukas Naef
ETH Zurich
lnaef@ethz.ch

Supervised by:
Prof. Dr. Donald Kossmann

Abstract

Software is omnipresent in today's world, but only computer scientists have a deeper understanding of how to build software. There are many different applications to introduce someone to the world of programming, but either they are complex or limited in their functional range. The goal of this thesis is to create an easy to use programming environment that motivates people to delve into the realm of software by developing their own programs.

In this thesis, we have developed a visual programming language called ClickScript, which is a data flow programming language running entirely in a web browser. To develop applications with ClickScript, we have created an IDE also running in a web browser. The base functionality of ClickScript can be extended by adding new component libraries written in JavaScript.

Our tests with volunteers have shown that it takes only little time for new users to grasp the concepts of ClickScript until they are able to create their own programs. In addition, we did a case study to show how ClickScript can be extended by integrating temperature sensors and fan controls over a network.

Acknowledgments

This thesis would not have been possible without the support of many people. Namely, I would like to express my gratitude to my supervisor Prof. Donald Kossmann for his continuous support, and for giving me the chance to work on such an interesting topic.

I owe my deepest gratitude to Julian Tschannen, Louis Woods and other colleagues for the interesting discussions and the time spent to do textual corrections of this thesis.

Many thanks to all the volunteers who spent their time playing around with ClickScript — especially to Ferdi as the youngest participant among them.

Special thanks to my sister Rahel for her support in the graphical design, and also to the rest of my family as well as my girlfriend Luzia supported me during my whole time at ETH.

Contents

Chapter 1

Introduction

We live in a world where computers are present everywhere and in everybody's life. Whether people browse the Internet, get a ticket from a ticket machine or use a mobile phone, they are confronted with computers and applications running on them. Only a few people, in general computer scientists, have a deeper understanding of how these applications work internally. Why is it, that only specialized people can realize their crazy ideas in this world of computer programs? We believe this is the case, because programming is too complex and the average person does not dare to touch it.

The goal of this thesis is to motivate everybody to create their own programs using a new visual programming language. The simple construction of the language should make the program and in particular its structure visual to the user like no text-based programming language can. To make the language accessible to as many people as possible, it can be used in a standard web browser.

We have developed a visual programming language called ClickScript [1]. ClickScript is an interpreted programming language implemented in JavaScript. The language consists of components representing data sources or actions, and connections representing data flow between the components. The language is fully managed in the browser, meaning programming and execution run in the browser. We also focused on the extensibility of the language, so that a JavaScript programmer can easily extend ClickScript with his own fancy components.

We conducted test with volunteers to see how quickly they understood the programming concepts of ClickScript. The tests have shown, that it takes the participants only a few hours until they were able to continue on their own. It was also interesting to see, that the testers were motivated by this experience to continue playing with the system on their own. This shows that you can unleash intrinsic motivation of non-programmers. In addition, we did a case study where we extended ClickScript by *Web of Things* [2] components to control sensors and fan controls over a network. This allowed us to create a ClickScript program to cool down a room once the temperature got too high.

The thesis is structured as follows: In the second chapter we introduce the language and execution model. The third chapter explains the structure of ClickScript, its different layers, the typing system and the architecture of the execution model. In the fourth chapter, we focus on how to extend ClickScript by custom components. Two case studies shown in the fifth chapter describe how ClickScript can be advantageous compared to other programming languages in terms of simplicity and extensibility. We finish the thesis with the conclusion, presenting a few ClickScript related programming languages and proposing future work for further development of ClickScript.

Chapter 2

ClickScript Language

This chapter describes ClickScript as a visual scripting language. After this chapter you are able to use ClickScript, create your own scripts and understand how they are executed.

2.1 Programming ClickScript

As an introduction we start with a first example of a clickscript[1]. Figure 2.1 shows a small ClickScript program. Basically this program will take two strings, concatenate these two strings, and put the concatenation to an output box as shown in Figure 2.2.

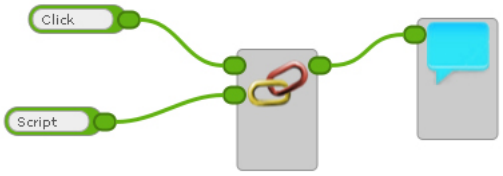


Figure 2.1: Small ClickScript Program

Figure 2.2: Result of ClickScript in Figure 2.1

As you can see, a clickscript consists mainly of components and connections,

[1] 'ClickScript' always means the language and 'clickscript' (lowercase) refers to a ClickScript script.

which we will refer to as wires, between them. An overview of the most common components and functionalities are listed in Appendix A.

2.1.1 Components

Components are the building blocks of ClickScript. A component may consist of *sockets*, *fields* and a *smart picture* to visualize its functionality. The colored rectangles on the left side of the component are called *input sockets* and the ones on the right side are called *output sockets*. Sockets are colored depending on their required types. In Section A.4 the most important types are listed. *Fields* are also some kind of local input of a component. The core of each component is its *Execution Function* which defines the functionality of the component. Basically the execution function reads the values from the input sockets and fields, process them and write the results to the output sockets of the component.

We distinguish three different kinds of components in ClickScript: *Primitives, Modules and Statements.*

Primitive The simplest components are called *Primitives* (see Figure 2.3). Primitives always consist of one field and one output socket. They are used to add static values of a certain type to the script. Therefore, the primitives execution function is straightforward, it just takes the field value, converts it to the requested type and writes it to the output socket. Like its output socket, the primitive itself is colored in the color of the type of the requested value in its field.

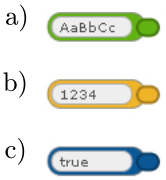


Figure 2.3: Primitives for Types (a)String, (b)Number and (c)Boolean

Module The *Modules* allow to add different functionalities to a clickscript. Unlike the primitive components, there is no limitation to the number of fields, output or input sockets. The modules themselves are gray and always have a small picture indicating their functionality. Figure 2.4 shows three examples of modules: POPUP to display the message coming from the input socket in an alert box, SWITCH to return a boolean value based on the state of the switch, and as a last example the STRING REPEAT component, which repeats the string from the input socket as many times as the number written to its field ('5' in Figure 2.4 (c))indicates and writes this new string to its output socket.

Statement In ClickScript there is no possibility to manage the control flow by wiring. To force a special order of execution we can use so called *statements.*

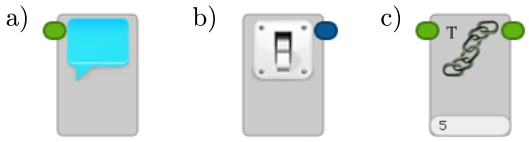


Figure 2.4: Modules (a)POPUP, (b)SWITCH and (c)STRING-REPEAT

Along with the properties of a module, statements also have at least one or more *blocks*. A component-developer has the possibility to control the execution order of these blocks in the execution function of the statement. A block itself is resizable and has its own name. Figure 2.5 shows two typical statements. First, we have the IF/ELSE statement which will execute the block named TRUE in case of a *Boolean true* on the input socket or the execution block named FALSE in the case of a *Boolean false* on the input socket. The second statement is a typical FOR-LOOP. As input it will take a number n and executes its BODY n times. Output sockets of a statement have a slightly different functionality to the output socket of a module. The outputs can be changed after each execution of a block. Therefore, in the case of a FOR-LOOP, we are able to write the counter value to the output socket. Further important statements are the FOR-EACH LOOP, which loops through all elements of a collection, as well as the SEQUENCE statement, which simply allows us to execute components in the FIRST block before components in the SECOND block.

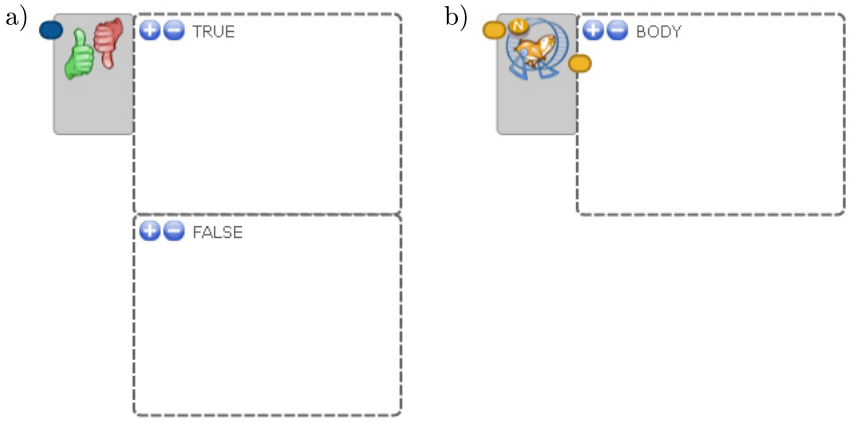


Figure 2.5: Statements (a)IF/ELSE, (b)FOR-LOOP

Something not mention so far is the so called *Execution View*, which is not mandatory for a component and will be explained in Section 2.2.2.

2.1.2 Wiring

Of course we need the possibility to send the resulting values from one component to another. This is the task of the *wires*, which always connect two sockets of different components. To do valid wiring and therefore guarantee a successful execution of the program you have to fulfill the following four simple rules:

1. Only connect sockets with the same types.
2. Always connect an input socket to an output socket.
3. Each input socket has exact one connected wire.*
4. Cycles are not allowed.

If the programmer makes invalid connections, the affected wires get black and dashed. Section 3.2.1 discusses the wiring rules in more detail.

2.1.3 Types

To encourage the goal of ClickScript being used by almost everyone we increased usability by introducing different colors for socket and wire types. The three basic types are:

String Text (color: green)

Number Integer or floating point number (color: yellow)

Boolean Logical values true/false (color: blue)

Have a look at Section A.4 to get an overview of all existing types.

Together with the color we also have the width of the wires and the bordercolor of the sockets which indicates a type specific property. Namely, the property whether the requested type is a collection or not. Sockets which require a collection type are black bordered and their connected wires are fat. The component on the right side, in Figure 2.6, is an ADD TO COLLECTION component which adds the number 4 to an existing collection. The first input socket of this component is colored yellow and is black bordered. Therefore, it requests a collection of type Number. Unlike the second input socket, which represents only a single element of type Number.

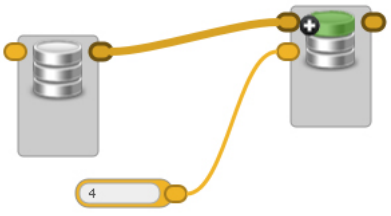


Figure 2.6: Example with Collection Type

2.2 Execution Principle

2.2.1 Execution Strategy

ClickScript is an interpreted visual programming language. The execution strategy of ClickScript is quite simple. As soon as a component has all its inputs ready, the component gets executed and writes the results to its output sockets. Therefore, when starting a program, all components with no input sockets get executed first. After this, the algorithm goes on looking for components which are ready to be executed, meaning components on which all incoming connections have a value, and runs them. This procedure is repeated until every component got executed. To understand programming ClickScript, this is almost all that you have to know about the execution of a script. These things are not quite as simple and how this execution strategy is realized behind the scenes will be explained in Section 3.2.

We have already seen an example of a clickscript at the very beginning of this chapter (Figure 2.1). In in that example the algorithm finds the two primitives with no inputs, i.e. they are ready to be executed. The two primitives get executed and write the field values *'Click'* and *'script'* to the primitives, output sockets. In the next step the CONCATENATION module is ready (has all its input sockets ready) and can be executed. After the execution of this module the output socket of the module will contain the value *'ClickScript'*. Therefore, the POPUP module becomes ready to get executed. Its execution function will call an alert box with the value of the input (Figure 2.2). Because there is no other unfinished component, the whole program terminates.

In the next example (Figure 2.7) the focus is on the control flow managed by the statements. Again, we start with a primitive component, which will write *'3'* to the output socket. Now the FOR-LOOP is ready to be executed, it will write the iterator value *'1'* to its output socket and run its block BODY the first time. Blocks gets executed similar to the start procedure of a clickscript. The execution starts with all components that are ready. Inside the BODY block there is a sequence with no input at all. It gets executed which means that it will run first block FIRST and afterwards block SECOND. In the FIRST block we will wait for five seconds and in the SECOND block the iterator of the for-loop will be shown by an alert box. Now all components in the block BODY have been executed and we go to the second iteration of the for-loop, and so on. The output of this clickscript is: message'1', break for five seconds, message'2', break for five seconds, message'1' and another break for five seconds.

There is also the possibility to run a clickscript repeatedly, meaning after finishing the execution of a clickscript the script restarts automatically. This mode of execution is called *Repeated Run*. An example of this mode will be discussed in Section 2.2.2.

2.2.2 Execution View

An additional but also optional property of a component is its *Execution View*. The Execution View of a component is an additional view which is linked to the

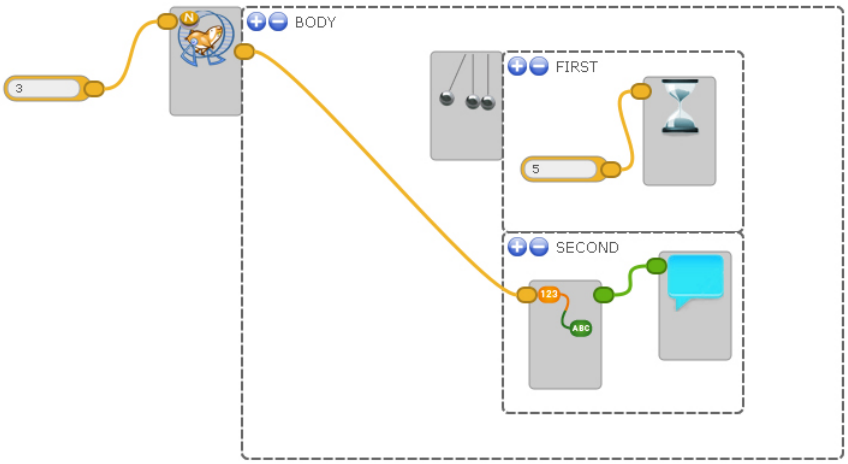


Figure 2.7: ClickScript with Loop- and Sequence-Statements

component. This view allows us to get information about the current state of a component during execution and it gives us the ability to interact with the component. In the lower half of Figure 2.8 we find two components, a TEXTFIELD component (left) and a DISPLAY component (right). In the upper half, the blue area, we see their Execution Views. In this example, the programmer already has entered manually *'test'* into the Execution View of the TEXTFIELD component. As soon as we execute the program, the TEXTFIELD component will take the input from its Execution View and will write it to its output socket. When ready to execute, the DISPLAY component gets executed and shows the value on its Execution-View-display (see Figure 2.9).

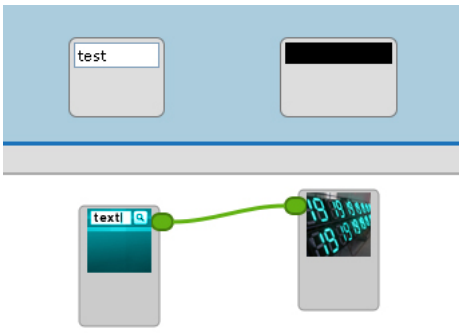


Figure 2.8: ClickScript Example with Programming and Execution View

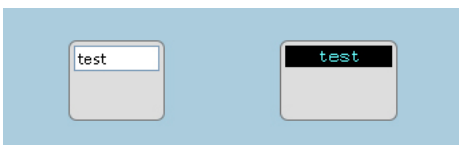


Figure 2.9: Execution View of Figure 2.8 after Run

To get an impression of the application area of the Execution View, we give another example in Figure 2.10, this time with a SWITCH and a LIGHT module. When we start the clickscript, the SWITCH component checks whether the switch in the Execution View is on or off. Based on this value the light component will turn the light on or off. It is possible to label these views individually, for example as in Figure 2.10 with labels 'switch bath' and 'light'. This feature makes sense if there are multiple instances of a component for example many switches, so they can easily be distinguished.

This is also a good example to illustrate the use of the *Repeated Run* facility mentioned in Section 2.2.1. If the script gets run just one time, the switch gets checked once and the light is on or off. But if we activate *Repeated Run* instead of a single *Run*, the script gets executed repeatedly in a loop and restarts after every execution. Therefore, we can switch on and off the light in 'real time'.

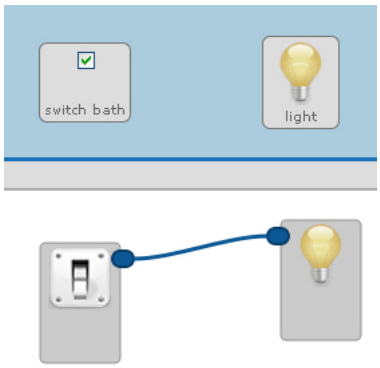


Figure 2.10: Typical Example for Repeated Run

2.3 Integrated Development Environment

2.3.1 Development

To develop and execute clickscripts we have built the ClickScript IDE which is roughly structured in the following parts:

- Menu Bar
- Library
- Execution View
- Programming View
- Console
- Tutorial
- Exercises

Figure 2.11 shows the first four parts of the IDE.

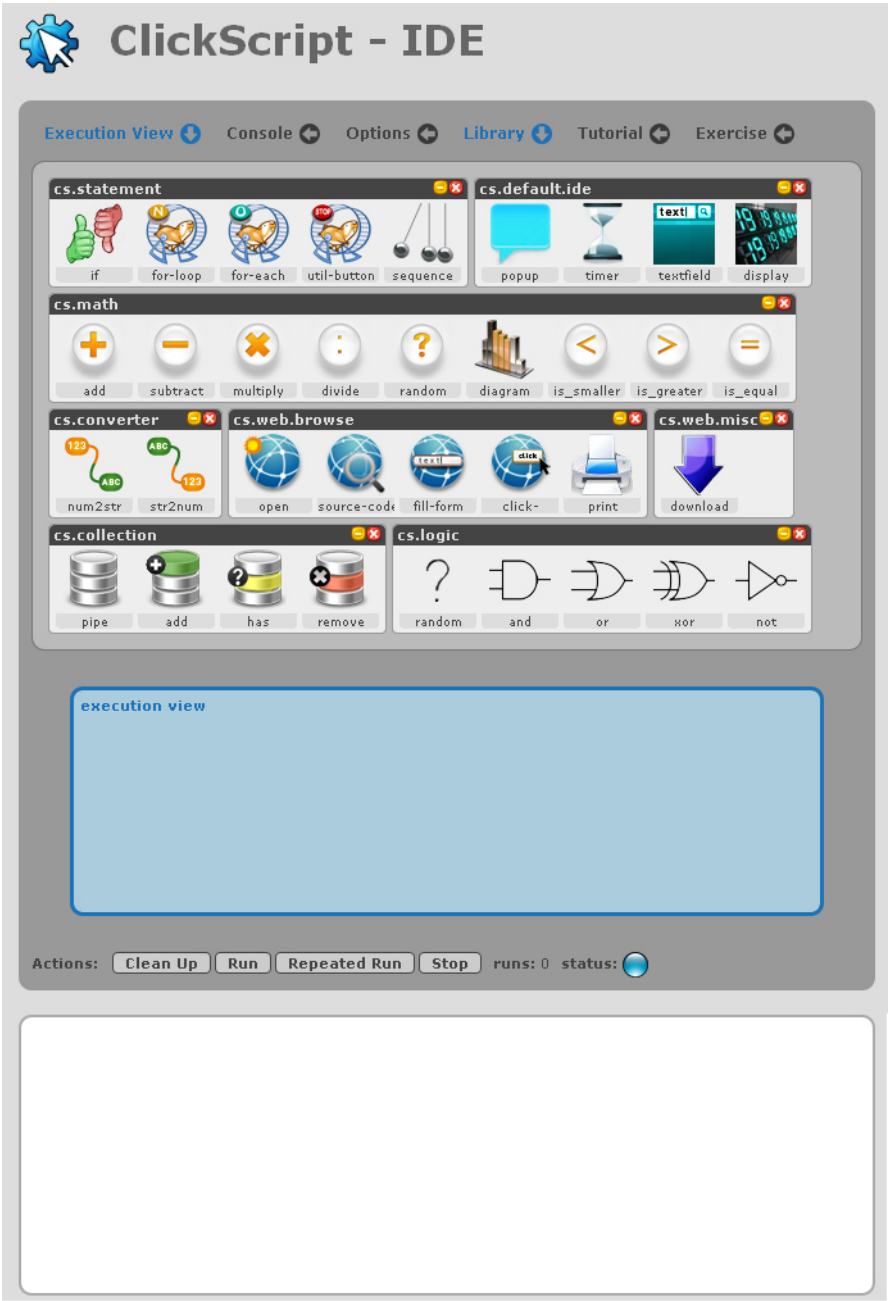


Figure 2.11: ClickScript IDE

Menu Bar At the top of the IDE we find the *menu bar*. It consists of menu switches to show or hide parts of the IDE. This includes an option-menu, which is integrated into the menu part. It includes options to show or hide libraries and enable or disable the debug mode. It also allows choosing the size and level of detail of the buttons for adding a component in the library.

Library Right under the menu bar part we find the *library* section containing all loaded libraries. As soon as someone clicks on one of the icons in this part of the IDE, a component of the appropriate type will be added to the Programming View. Right clicking such a icon will show a detailed description of the component type.

Execution View As described in Section 2.2.2 the *Execution View* (blue part in Figure 2.11) is used during execution to show the state of different components and to have a user-interface to the program. To not get lost in all these different icons, it is possible to label them by simply double clicking on them and entering a name. If the user moves the cursor over a component in the Programming View, its corresponding Execution View gets highlighted by a red border as a usability feature for ClickScript programmers.

Right below the Execution View relevant actions for the execution of a click-script are located, such as : *clean up* the Programming View (delete the script), *run* the script, *repeated run* to run the script in repeated mode, and *stop* to stop execution of a run in repeated mode. Also placed in this part we find a counter which indicates the current number of runs during a repeated run as well as a light indicating the current execution state (blue: finished, green: running, yellow: waiting, red: user stopped execution or an error occurred).

Programming View The main part of the IDE, where the ClickScript programmer is going to 'click together' his program, is called *Programming View* (white part at the bottom of Figure 2.11). If you have loaded two components from the library by clicking on their corresponding icons in the library, you can connect them by first clicking on the socket of the first component and then on the socket of the second component. Active sockets will be highlighted by a blue border and blue background.

To remove a wire, you can double-click it. When moving the cursor over a component, two small icons appear. A red button to remove the component and an information button (blue) to get a description of the component, of its functionality and of its required inputs, outputs, and fields.

There exists also the possibility to resize blocks of the statement by clicking on the plus or minus icons on a block.

Console Figure 2.12 shows the console of the IDE. It can be used by components to output data. Furthermore, the console can show debugging information or the worklist mutations during execution, if the corresponding options are activated in the option menu. Component errors, as a result of invalid component development, also get displayed in the console.

Figure 2.12: Console Part of IDE

Tutorial In the *tutorial* part we find an introduction to ClickScript as well as a few examples to get started with ClickScript.

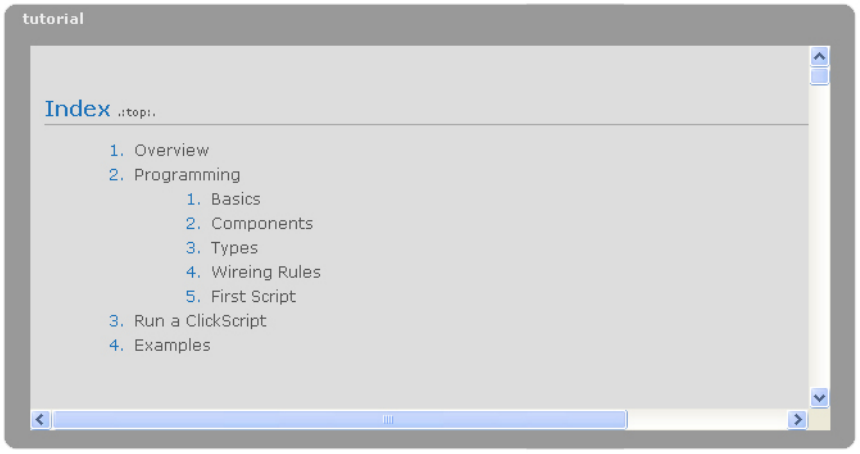


Figure 2.13: Tutorial Part of IDE

Exercise If activated in the menu bar, the *exercise* part is shown at the bottom of the IDE. Figure 2.14 depicts the first level of this exercise. First, always the exercise appears and after fifteen seconds a button 'solution' gets enabled which permits the user to see the solution of the exercise. Some exercises also include a tip to the exercise. Each level has its own level code. Entering the level code into the textfield in the upper left corner of the exercise part, allows the user to jump directly to this level.

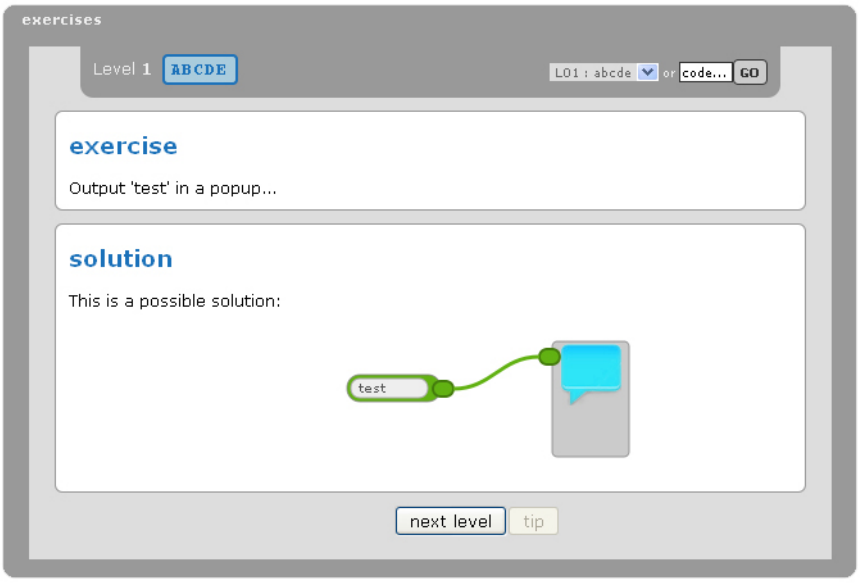


Figure 2.14: Exercise Part of IDE

2.4 Extensibility

2.4.1 Library

ClickScript benefits from its extensibility. As we will see in Section 4.3, a ClickScript developer that knows the basics of JavaScript can easily extend ClickScript by his own libraries, in only a few lines of code.

Using the ClickScript Firefox add-on (see Section 4.7 for details) allows us to do more sophisticated things across the web as well as locally on the computer. There exists a few restrictions for webpages in general, we can bypass by this ClickScript add-on. For example it allows us to break the *same origin policy*, which normally prevents getting data from domains your page is not from, or accessing the users file system. To show the variety of application domains of ClickScript, we will go through the libraries developed in this thesis and pick out some interesting features and components. Keep in mind there is an explanation of each component in Appendix A of the appendix.

Statements Most of the common statements were already discussed in this document before. In this library we have the following statements: IF/ELSE, FOR-LOOP, SEQUENCE and FOR-EACH. There is one more statement called UNTIL-STOP, which has a block named BODY and an Execution View with a stop button but no in- or output socket. Executing this statement will repeat the execution of its block BODY until its stop button is pressed.

Logic The logic library consists of the components of the most common Boolean operators ,such as AND, OR, XOR and NOT. In addition a RANDOM component which generates a true or false value randomly is part of this library.

Collection As already discussed in Section 2.1.3, we also have collections in ClickScript. This library allows us to handle such collections. As showed in Figure 2.15, these modules can also be used to build collections inside of a loop. The blue light will switch on and the yellow light will not, because the value '2' has been removed from the collection.

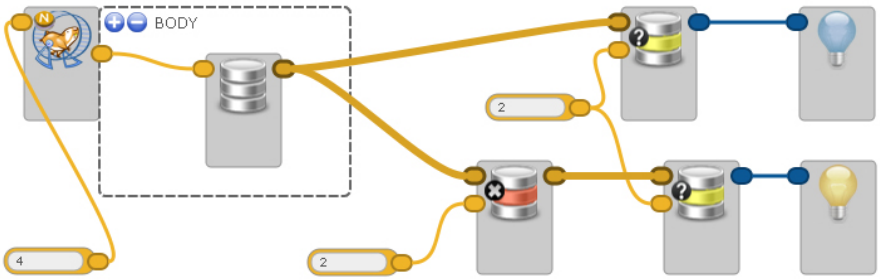


Figure 2.15: Collection Library Example

Mathematics Simple mathematic operations like addition, subtraction, multiplication, and division, but also value comparison are typical representative modules of this library. More sophisticated components are shown in Figure 2.16. The script in that example connects a RANDOM component, which outputs a random value between 10 and 30 with a BAR-DIAGRAM that plots the value (the script is executed in repeated mode).

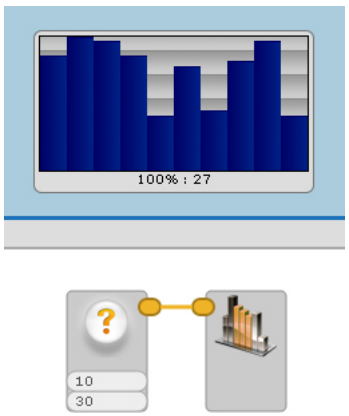


Figure 2.16: Mathematic Library Example

Web Automation Like we mention above in this section it is possible to navigate through webpages because of the Firefox add-on. The components in the web automation library works on iframes and make it possible to fill values into form fields, read out values from a page, press buttons and print open pages. The script given in Figure 2.17 shows an example in which ClickScript opens the site www.google.ch, fills the word 'clickscript' into the first field, presses the first button on the page and prints the search results.

Figure 2.17: Web Automation Example

Controlling Ubiquitous Components We have developed a few modules to control Web of Things [2] components (see Section 5.2 for detailed description).

Robotic This library shows an approach to use ClickScript as a simulation environment. Well known modules are LIGHTs and SWITCHs. Another interesting component is the CAR module, which has one input that takes a number between zero and three. This number indicates the speed of the car. Depending on the speed, the picture of the car in the Execution View will be replaced by another animated gif. Not only motion but also things like temperature can be simulated, as shown in Figure 2.18 which depicts HEATING and COOLER modules that will vary the room temperature. The TEMPERATURE module is not wired to the other objects but will show the current room temperature.

There are libraries like *String* and *Ide* that we did not talk about in this section, but we already have discussed a lot in the sections before. And of course there are lot of ideas to be developed for future use, e.g. as the following two examples:

Web Creation It should not only possible to browse pages but also to create own pages by modules. For example, using a module to create a table with two cells where an input corresponds to the content of each cell or using a module to create the title of a page.

File System With the support of the Firefox add-on we also get access to the file system of the user. This enables ClickScript to move, delete or copy files. Already implemented is a DOWNLOAD module with the two inputs for source (url) and target (file path) to download files onto your hard disk from the Internet.

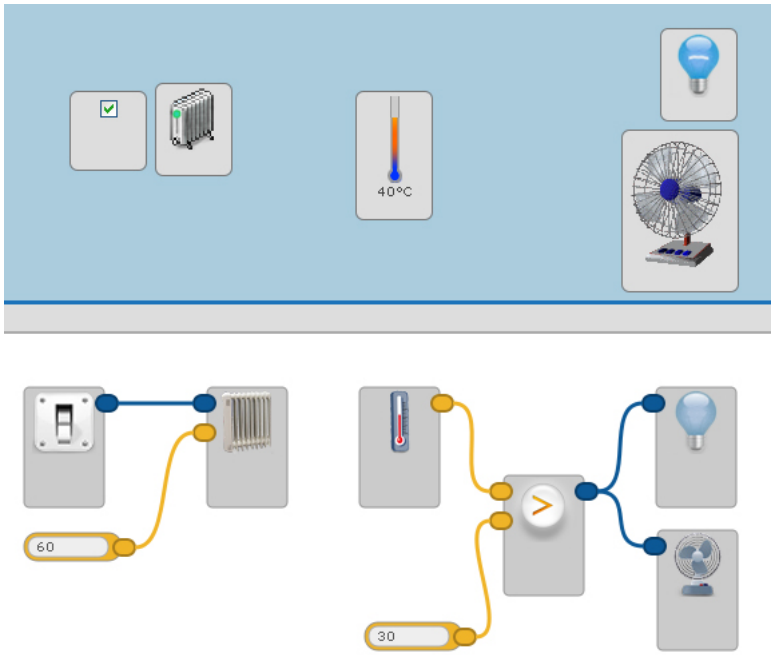


Figure 2.18: Robotic Example

2.4.2 Modularity

The idea of Modularity is, to save a clickscript itself as a component. This not only allows us to reuse scripts in an easy way but also makes complex scripts more readable.

Chapter 3

Architecture and Concepts

In this chapter, we will discuss the layered architecture and the execution concepts of ClickScript.

3.1 ClickScript - Layers

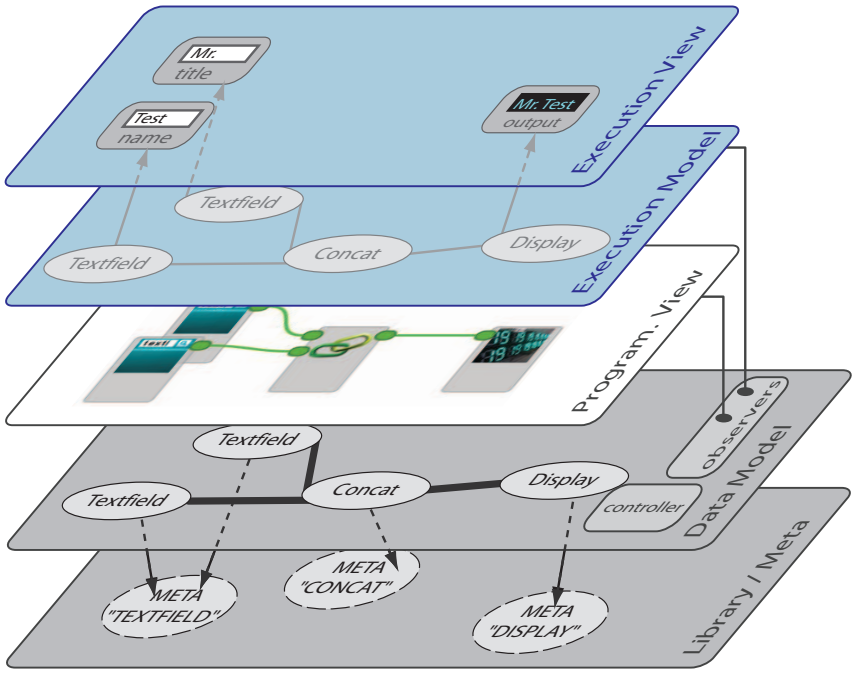


Figure 3.1: The Layers of ClickScript

ClickScript is built up in several layers. Figure 3.1 illustrates these layers with a simple clickscript. It gets a *title* and a *name* from a textfield, concatenates them and displays the full 'title name' string in a display. In the following

paragraphs, the layers will be explained bottom up: from the library layer, up to the Execution View layer, as shown in Figure 3.1.

Library / Meta Layer The library layer, also called meta layer, consists of the component definitions. This layer stores for each component type its meta data. As we can see in the figure, there exists exactly one meta component instance for each component type (marked in the figure as dashed ellipses). If a library developer adds its own component to ClickScript, the component definition is added to this layer.

Data Model Layer The script structure with its components and connections is stored and maintained in the data model layer, also known as programming model layer. This layer also holds the controller to the data model. Therefore, it implements the controller and model part of the MVC pattern (model-view-controller pattern [3]. Other layers may add additional views of the data model by registering itself as observers to the data model controller.

Programing View Layer The programing view is the visual part where the ClickScript user creates his clickscript. It contains the whole user interaction functionality like adding components, creating new wires or moving components. Commands for model changes will be propagated to the data model layer. Therefore, this layer is registered as an observer to the data model layer. Not shown in Figure 3.1 are the references from a component in the Programming View or Execution Model layer down to the data model layer which are needed to identify the programming view or execution model component in the data model layer.

Execution Model Layer The second to top layer contains the execution model. This model is registered as an observer to the data model layer. It contains the structure of the current script and functionality to run the components. This includes the state of the components, like if they are currently running, already finished or ready to run. To get a closer look to the execution of a clickscript, see Sections 3.2.2 and 3.2.3.

Execution View The top layer was introduced in Section 2.2.2. On this layer, we find the view part of the components during runtime. Execution Views may be used to get user input (e.g. HTML text fields in the figure) or as an output device to visualize the state of a runtime component (e.g. the simple output display in the figure).

3.2 Execution Model

3.2.1 Wiring Rules

In terms of usability, it is important to show the user invalid wiring immediately. For this purpose, invalid wires get marked as invalid, meaning its style changes to a black dashed line (see Figure 3.2). The wires symbolize the data flow between the components. In the following, we go through the five wiring rules which guarantee a successfully executable clickscript.

Rule 1 - Same Types The colors of the sockets symbolize the type of a socket, and only connections between sockets of the same color are allowed. Thus, connections like shown in Figure 3.2 are not allowed.

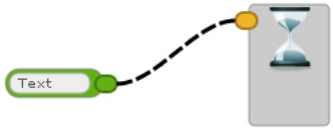


Figure 3.2: Violation of Wiring Rule 1 (Same Type)

Rule 2 - Input to Output The data always flows from one output socket to an input socket of a next component. Therefore, an output socket can only be connected to an input socket. Connections like in Figure 3.3 are invalid.

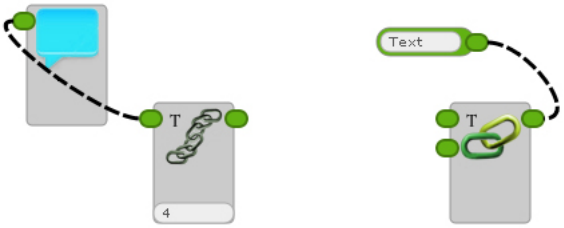


Figure 3.3: Violation of Wiring Rule 2 (Input to Output)

Rule 3 - One Wire per Input Each socket represents one input value used by a component. There has to be exactly one wire connected to each input socket. On one hand a script cannot complete execution if there are open Input sockets. Thus, it is not allowed to have unconnected input sockets in the script (Note that it is allowed to have unconnected output sockets). On the other hand if there would be more than just one wire connected to an input socket, the component had to choose a wire during the execution to get an input value. There is an exception to this rule in the case of an if/else statement. If one incoming wire is from a TRUE block of a component and the other one from the components FALSE block, the wiring is correct because it is not possible that both of these predecessors got executed (see 3.4 (b) for example).

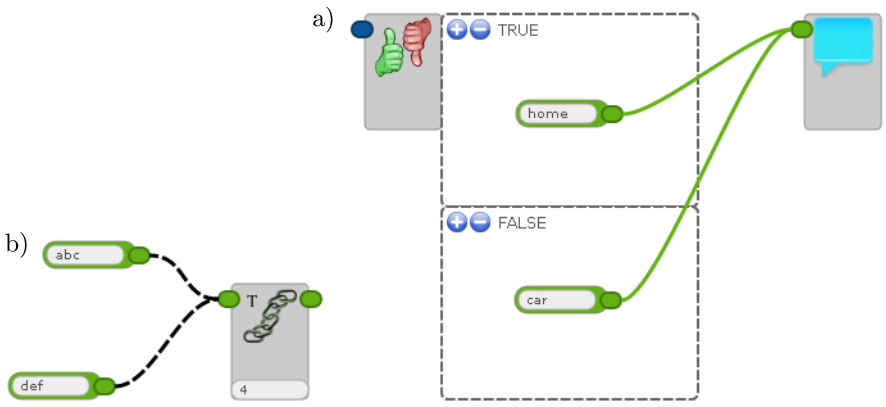


Figure 3.4: (a) Violation of Wiring Rule 3 (One Wire per Input), (b) Exception of Rule 3 for IF/ELSE Component

Rule 4 - No Cyclic Graph To avoid deadlocks it is also forbidden to build cycles in the graph. A cycle would mean, that a component *A* depends on the result of a component *B* and vice versa (see Figure 3.5), which would lead us inevitably to a deadlock. Also, one has to pay attention to the wiring in connection with blocks. Outgoing wires of a block are handled like output wires and the same for incoming wires. Therefore, a script like shown in Figure 3.5 (c) is also not allowed (see Section 3.2.3 for details on this issue).

3.2.2 Script Structure

ClickScript is a data flow programming language not an imperative programming approach like a lot of mainstream programming languages like Java or C#. This means that we program by defining the data flow. It is possible to map clickscripts to a directed graph symbolizing this data-flow. This is possible because of wiring rule 2 mentioned in Section 3.2.1 which defines that the wires always connect an output socket of a component to an input socket of another component. Combined with wiring rule 4 (no cyclic graph) the resulting directed graph is acyclic. Thus, we got a DAG (directed acyclic graph) as a script structure for each clickscript. Ignoring the FOR-LOOP statement, Figure 3.6 shows an example script and Figure 3.7 its structure as DAG. As needed in the following subsection, there is an additional node called *start* (orange colored) which is added in front of all components without any input and not residing in a statement block. This node is also the root of the structure which now represents a DAG tree.

Statements like loops and conditionals need additional considerations. There are not only connections to the statement, but also connections into and out off the blocks of a statement like shown in the FOR-LOOP statement in Figure 3.6. This FOR-LOOP has an incoming wire from the primitive with value 6 as well as an outgoing wire to the SUBTRACTION component. If we decide to handle all incoming connections like statement inputs and all outgoing connections as outputs of the statement, then we have integrated the statements to the DAG

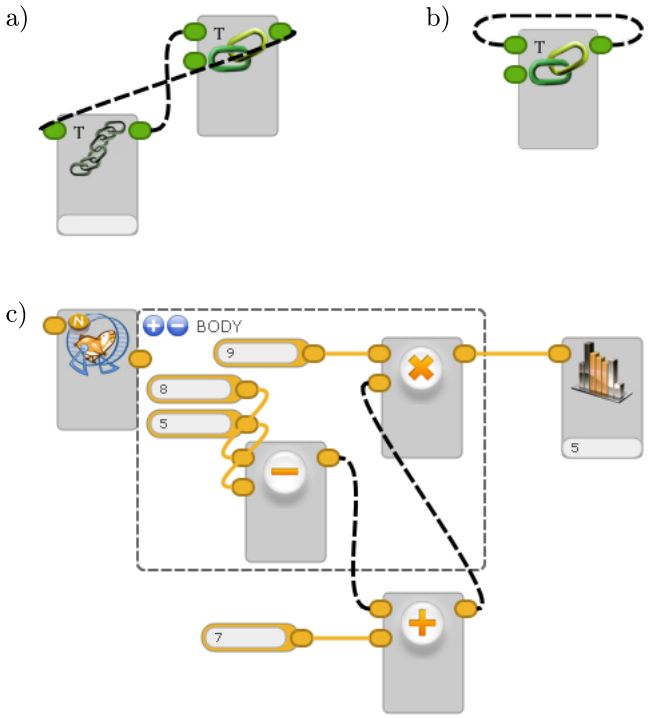


Figure 3.5: Violation of Wiring Rule 4 (Cyclic Graph)

(see Figure 3.7). The statement itself is now integrated to the DAG, but what about its blocks? Each block itself represents a subroutine with incoming connections as inputs and outgoing connections as outputs. Assuming all incoming wires have a value, the components inside a block build themselves a DAG like shown in the dashed box in Figure 3.7. The small black boxes symbolize in- or outputs and are parts of the DAG.

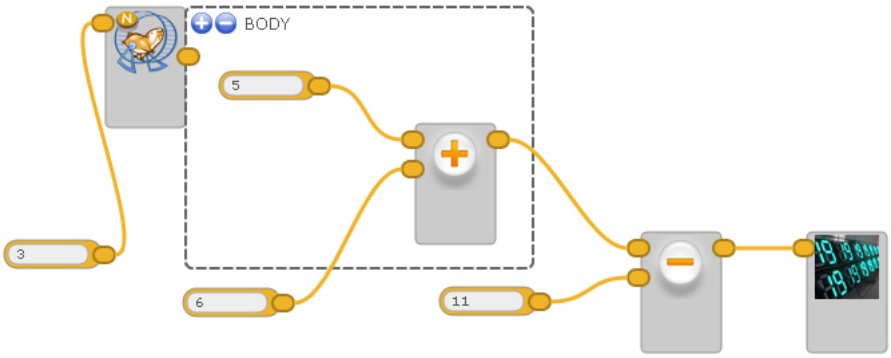


Figure 3.6: Example Script

The approach of boxing statements like this, has also impact concerning the execution model as explained in the next section.

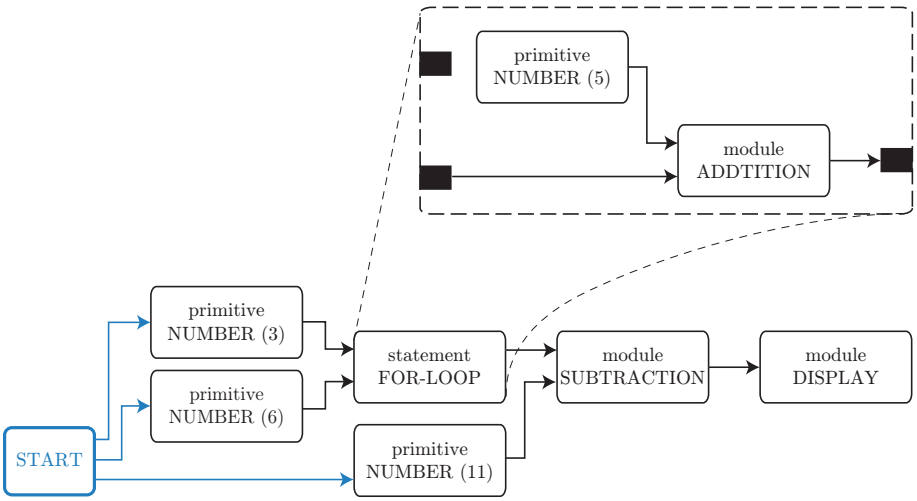


Figure 3.7: Schematic DAG for Figure 3.6

3.2.3 Execution

The basic approach to run a clickscript is to take its DAG-representation defined in the previous Section and start at the root nodes (components with no inputs in the most outer block), run these components and execute afterwards the successor nodes if they are ready to be executed. To execute the script from Figure 3.6, we would start with the root of the DAG tree (*start* in Figure 3.6), always executing blocks which have all their inputs ready. Because of the boxing model of statements, described in 3.2.2, the statement FOR-LOOP only gets executed if all its inputs are ready (also incoming wires) and it will finish only if the statement has finished its execution. Therefore, the SUBTRACTION will start its execution after the three runs of the FOR-LOOP's block and will take the two inputs 11 and 11.

In the pseudo-code of Listing 3.1, the two main functions of the execution are listed. To start a script, we would start with running the outer most statement called *cs.statement.program* (visualized by the Programming View itself). In its execution-function there will be just one command to run its block, which is represented by the whole Programming View. Therefore, to start the execution of a clickscript, we call runBlock() on the root statement cs.statement.program.

block.getAllStarterComponents() If we want to run a Block, we need to get all starting components (successors of the root). As we already know from Section 3.2.2, a block has its own DAG representation. We get the first part of the starting blocks by collecting the components with no input sockets. Also, we have to add the components with inputs coming only from outside of the block.

component.isReady() Another interesting question is to distinguish whether a successor component is ready or not. The following condition have to be ful-

```
1
2  runBlock: function(){
3
4     // mark block as active
5     block.isActive = true;
6
7     // get all components to start with
8     starterComponent = block.getAllStarterComponents();
9     foreach( component in starterComponent ){
10       component.runComponent();
11    }
12
13    // update the state of a block
14    block.isActive = false;
15    block.isFinished = true;
16 }
17
18 runComponent: function(){
19
20    // execute execute-function defined by the metadata from
21    // the library in this function also the output sockets
22    // get filled
23    component.execute();
24
25    // check successor components an run them if ready
26    foreach(successor in component.getSuccessorComponents()){
27       if(successor.isReady()){
28       successor.runComponent();
29    }
30 }
```

Listing 3.1: "Pseudo Code for Execution of Block and Component"

filled:

1. All input wires are loaded (its predecessors has been executed)
2. Its parental block is activated
3. For all input wires originating in a block, the block has to be finished

The first item is a basic condition. No component can be executed, if its inputs are not evaluated by its predecessor components. The second condition prevents executing a component inside a block which should not yet be executed. For example, if a component's successor is inside a FALSE block of an if-statement and if the FALSE-block is not active (maybe it never will be) the component should not be executed. The last condition is not as simple as the other two: If we assume that each block represents a component with inputs and outputs (incoming and outgoing wires) and wires could cross multiple blocks, we should make sure that the wires successor only gets executed if the blocks which are crossed by the wire as outgoing connection are already finished. A statement has finished if all its blocks have finished. This is a direct consequence of the boxing approach of statements mentioned in Section 3.2.2. This means that example (a) in Figure 3.8 will not popup three times like example (b). If the goal of a loop is to generate a collection, it is required to use collection components like shown in Figure 3.8 (c). This script would show a collection {1,2,3} in a popup.

Worklist With the algorithm described above, it is possible to run a click-script. The whole program would be executed as one single function call. This does not work for functions like AJAX-calls which take some time. The whole program would be blocked until the response of the request arrives. Also, it is difficult to determine the time when a program has finished, because a program may not have finished when all modules have been executed. One example would be all the components in an if/else statement inside the block which never gets executed. Due to this reasons, a worklist is added to the ClickScript runtime. The worklist stores successor components until they are successfully executed. If a component from the worklist is ready we run the component. When a component from the worklist has finished execution, we add the successor components to the worklist and remove the current component from the worklist. When the worklist is empty, we know the program has finished. Each time we are able to execute a component from the worklist, we will do an additional loop through the worklist. In case of an asynchronous component it could be that the worklist is in a waiting mode until the asynchronous component is finished and started an additional check of the worklist. Listing 3.2 shows the pseudo code for the worklist.

Repeated Run As described in Section 2.2.1, repeated run can easily be realized. After the whole program has finished, all starting components of the program are added again.

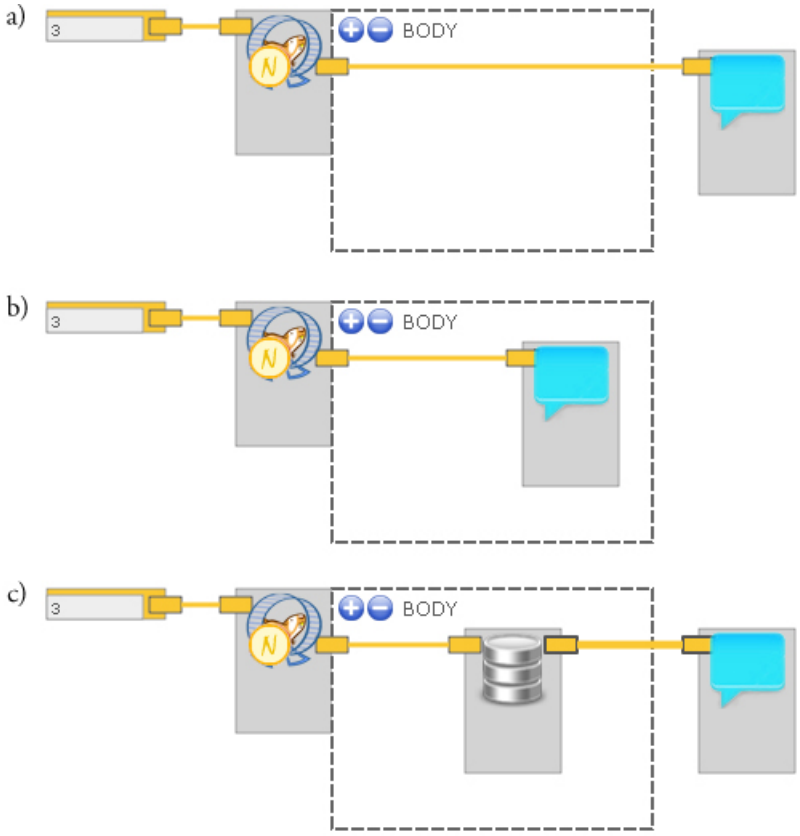


Figure 3.8: Execution of Statements

```
1
2  class worklist {
3     runWorklist: function(){
4
5        runAgain = false;
6
7        foreach(component in worklist){
8           if(component.isReady() && !component.isRunning()){
9              component.run();
10             runAgain = true;
11          }
12          if(component.isFinished(){
13             worklist.add(component.getSuccessors());
14             worklist.remove(component);
15             runAgain = true;
16          }
17       }
18
19       if(runAgain)
20          this.runWorklist();
21       else {
22          if (worklist.size() == 0)
23             alert("Finished execution");
24          else
25             alert("Waiting...");
26       }
27    }, ...
28 }
29
30 /**
31 * If execution function of a component is asynchronous
32 * it has to call a I-am-finished function:
33 */
34 class component {
35    forceFinish: function(){
36       component.isFinished = true;
37       worklist.runWorklist();
38    },...
39 }
```

Listing 3.2: "Worklist Pseudo Code"

3.3 Type system

Types can be added to the library in a simple way new components (see Section 4.4 for details). Figure 3.9 shows the UML diagram of the type system for the sockets of ClickScript. A type may be a collection or not. Collections represent a container of elements of its type Figure 2.15 shows an example of the collection operators. The type itself gets name and color from its structure. The structure may also contain several attributes which itself are of type Type. These attributes allows to develop some more complex type structure.

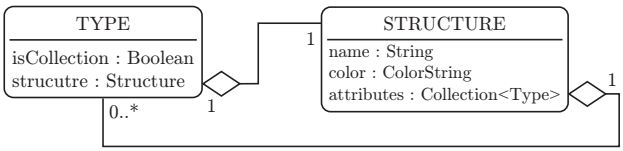


Figure 3.9: Types in ClickScript

3.4 Usability

We have chosen a visual programing language because it makes it easier to use, also for people not working regularly with programming languages. In the following we listen a few additional ideas to support the usability of ClickScript.

Visual Types The feature of colored and styled Types (colors and stroke width of collection types) allows to distinguished them very easily. Wrong usage of types is immediately shown to the user by a black dashed line.

Smart Component Description By right clicking a component in the library will show a tooltip with a complete description of the component and a list of its inputs, outputs and fields. The same information is shown by clicking on the blue question mark on a component in the Programming View.

Error Handling Every incorrect wiring is directly shown to the programmer. If a clickscript only contains valid connections, the program will be executed without any error, provided that the individual components don't fail. In comparison to traditional programming languages the programmer does not have to care about special cases and exceptions. In case of an error occurring during the execution of a component (because of a programming error in the component) the execution of the script stops and the user gets informed about the error through the console.

Description of Execution View Components Moving the cursor over a component in the Programming View marks its representation in the Execution View (if any) by a red border. Often it makes sense to label the Execution

Views of the components, for example if there exists more than just one text field component and we want to know which one is needed for a specific purpose. In this case, the user can label the Execution View of a component by double clicking it. An example of this is shown in Figure 3.1, the views are labeled with *title*, *name* and *output*.

Multi Typing Lots of components have input-sockets which may handle different types of input. For example the POPUP component, has a default input of a *String*, but it can also handle type Number directly. Therefore, an input socket may have different valid types depending of the type the component developer can handle as input.

Smart Fields Input sockets can be mutated to fields and vice versa. This allows the user to quickly enter a value to an input socket by typing the value directly to the field.

Multi Language Support Because ClickScript is not only developed for computer scientists, there has to be the possibility to switch the language of the description of modules, tutorial and other facilities of the ClickScript IDE.

Chapter 4

Implementation

This chapter goes one step deeper into ClickScript by discussing chosen techniques, class structure and focusing on how to extend ClickScript by your own types and components.

4.1 Used Techniques

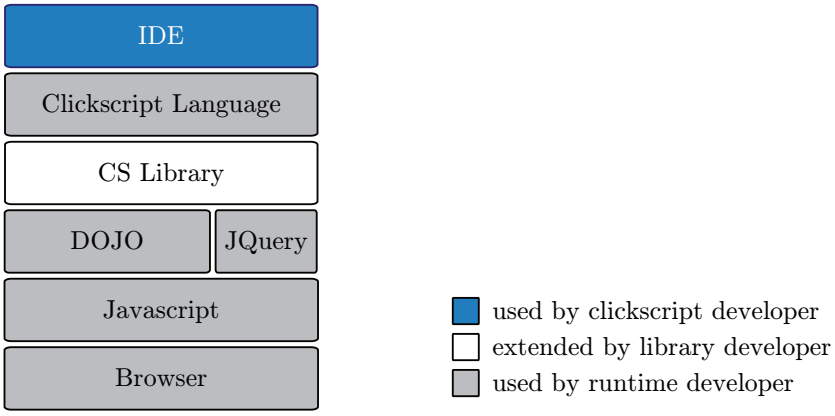


Figure 4.1: Struture of ClickScript

One goal of this thesis is to develop a programming language which can be used by almost everyone. As a consequence of this goal, we decided to write ClickScript in the *browser*, because nowadays everyone has a browser and is familiar with using it. It also allows people to place their clickscript directly on a webpage. We have chosen *JavaScript* as the developing language, because it runs on every browser and does not need additional plugins. Not to write code for each browser separately and to get access to lots of helpful, predefined features, we had to choose a JavaScript library. To draw our clickscripts we need some sort of vector facilities. Thus, and because Firefox implements the *SVG (Scalable Vector Graphics)* [4] standard and Internet Explorer only VML (Adobe

SVG Plugin is available), we looked for a JavaScript library which can handle both standards. *Dojo* [5], as one of only a few libraries, offers the possibility to use these two standards through one single interface. Another reason to choose Dojo as the basic JavaScript library is its pseudo object orientation by classes and inheritance which makes code more readable and its very large function range.

To keep developing of own components as simple as possible and because Dojo is not that well known we added *jQuery* [6] as an additional library for component developing. To be flexible in placing clickscripts onto web pages and to keep the ClickScript source code size small, we just reference the JavaScript libraries from external servers.

Based on these techniques we have the *ClickScript libraries* developed by component developers (see Section 4.3 for further information). On top of ClickScript we built the ClickScript IDE which is then used by a ClickScript programmer.

4.2 Filestructure

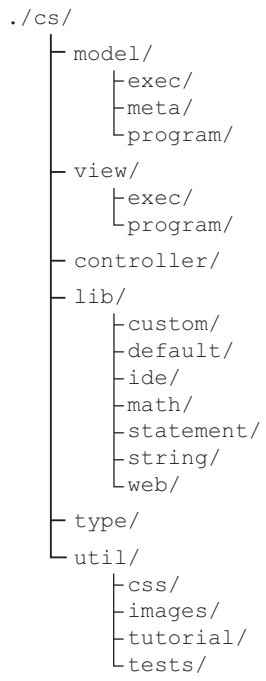


Figure 4.2: File Structure of ClickScript

In Dojo, class names directly depend on the location of the JavaScript class file. For example if we have a class *cs.util.Container*, its code is located in the folder */cs/util/Container.js*. The file structure and therefore the class naming is chosen on the first level according to the *model view control* structure (first

three subdirectories of cs folder, listed in Figure 4.2). The three main folders are split according to the clickscript layers discussed in Section 3.1. Therefore, the class for components in the Execution View is named *cs.view.exec.Component* and located in the file */cs/view/exec/Component.js*. The next few paragraphs will roughly discuss the ClickScript code folders listed in Figure 4.2.

model The *model* folder contains the classes for the structure of the *execution model layer*, *meta layer* and the *data model layer*.

view All the classes for the views are located in the *view* folder. These are the classes for the structure of the *Execution View* and the *programming view*. There is also a folder *util* in this directory which contains supporting classes for the visualization like classes for textfields in SVG, visual togglers, tooltips and others.

controller The *controller* folder contains the classes which control the behavior of the layers.

lib All the predefined libraries are stored in subfolders of the *lib* folder. As described in Section 4.3, it is not necessary to put own components into this directory.

type Classes for the type system of ClickScript are stored in the *type* folder.

util The *util* folder contains additional files needed for the IDE like images, css and HTML files as well as the exercise and tutorial part. Also, there exist a unit test for most of the classes which are also placed in this folder. Finally, we also find a few basic classes to facilitate programming in JavaScript (e.g. classes for container, map and class handling).

4.3 Writing own Components

A main part of the ClickScript's attractiveness is given by the component diversity. Therefore, it is important to know how to extend ClickScript with your own libraries and components. This section explains how to write custom components.

The code of already installed components is located in the sub directories of the folder *cs/lib/*. To add a component to clickscript you have to add a *component definition object* to the csComponentContainer as shown in Listing 4.1.

```
csComponentContainer.push({
    name: "",
    description: "",
    inputs: [],
    outputs: [],
    fields: [],
    image: "",
    exec: function(state){},
    blocks: [], /* statements only */
    view: {
        source: ""
        /* additional functions */
    },
    reset: function(state){}
});
```

Listing 4.1: Structure of Component Definition Object

4.3.1 Modules

To explain the properties of such an object (see Listing 4.1), the IS SMALLER module is introduced as an example (see Listing 4.2). This module has two inputs of type *number* and one output of type *boolean* which will be true if the first input value is smaller than the second one.

name Defines the library name (first part of the string until the last point) and the component name (substring from last point to the end of the string). Therefore, the library category is *cs.logic* and the components name is *is smaller*.

description A small description of the component which will be visible to the user if he activates the info tooltip by pressing the info button on a component. Always uses the names of input sockets (here *NUMBER1* and *NUMBER2*) and output sockets (here *IS SMALLER*), to describe the functionality of a module.

inputs Is used to define the input sockets by adding objects with the two properties: *name* and *type* to the array. The name will also be shown in the info tooltip.

outputs Is used to define the output sockets in a similar way as *inputs*.

fields Is used to define fields for primitives (shown in Figure 2.3) or for additional static inputs for components (e.g. the minimum and maximum definition of the RANDOM component in Figure 2.16). Because a fields value is not a requirement for the execution of a script, fields may be used as optional inputs with a default value. In all other cases it is only recommended to use a field instead of an input socket if it is absolutely unlikely that an

```
csComponentContainer.push({
   name: "cs.logic.is_smaller",
   description: "Compare the two input values. Output is true
       if NUMBER1 < NUMBER2",
   inputs:
   [
      {
         name: "NUMBER1",
         type: "cs.type.Number"
      },
      {
         name: "NUMBER2",
         type: "cs.type.Number"
      }
   ],
   outputs:
   [
      {
         name: "IS SMALLER",
         type: "cs.type.Boolean"
      }
   ],
   image: "web/things/smaller.png",
   exec: function(state){

      // get input values from input sockets
      var n1 = state.inputs.item(0).getValue();
      var n2 = state.inputs.item(1).getValue();

      // calculate output
      var out = n1 < n2;

      // write result to output socket
      state.outputs.item(0).setValue(out);
   }
});
```

Listing 4.2: Code for IS SMALLER Module

other component needs the value as its input. Fields are not used in this example.

image Path to the component's picture. The picture should have a size of 48x48 pixel.

exec Defines the execution function. The execution function is explained in more detail below.

reset Not used in this example and will be discussed in Section 4.3.4

The core of the component definition is the *execution function* ('exec' property). The function always takes one argument: *state*. The object represented by *state* allows direct access to the important properties of the current component. It is shown in Figure 4.3:

```
1 {
2    inputs: [/* cs.util.Container of input sockets */],
3    outputs: [/* cs.util.Container of output sockets */],
4    fields: [/* cs.util.Container of fields */],
5    blocks: [/* cs.util.Container of blocks,
6                statements only! */],
7    view: /* view object */
8 }
```

Listing 4.3: State Object for Execution and Reset Functions

In the above example of the IS SMALLER module in Listing 4.2 we have an example of the execution function. In line 26 and 27 the two input values are read from the input sockets by accessing the *state.inputs* container. After processing the input values (line 30) the result is written to the output socket (line 33). The execution function is always structured in the following three parts:

1. Read input values from input socket
2. Process input values
3. Write results to the output sockets

The component developer can assume that the type of the input values corresponds to the type of the input socket. No further guarantees can be made, meaning for example that if he needs an integer to get the number of loops in a FOR-LOOP statement which input socket is of type number, the developer has to make sure he gets an integer and not a float. Therefore, the code to get n would then look like: *n = Math.floor(state.inputs.item(0).getValue());* which rounds a possible floating point number to an integer.

As an additional feature, it is possible to execute a component asynchronously. To do this one has to use the function *setAsync()*. As soon as the asynchronous function call has finished it is required to mark the component as finished by

```
1  csComponentContainer.push({
2     name: "cs.default.ide.timer",
3     description: "This modules waits for N seconds",
4     inputs: [{
5        name: "N",
6        type: "cs.type.Number"
7     }],
8     outputs: [],
9     image: "ide/wait.png",
10    exec: function(state){
11
12       // mark this component as asynchronous
13       this.setAsync();
14
15       // get input value
16       var n = state.inputs.item(0).getValue();
17
18       // make sure we got miliseconds
19       n = Math.round(n*1000);
20
21       // call finishAsync after n seconds
22       var component = this;
23       setTimeout(function(){component.finishAsync();},n);
24    }
25 });
```

Listing 4.4: Code for TIMER Module

calling the *finishAsync()* function. Listing 4.4 shows the code for the timer module, which has only one input indicating the number of seconds to wait. Because the execution function is called on the execution component's object, *this* directly points to the execution component.

4.3.2 Statements

The only difference between a statements and modules are the blocks. A component is marked as a statement simply by adding blocks to the component definition object.

The Listing 4.5 shows the definition of the if statement. Looking at the input, we decide in line 18 whether block 0 or block 1 is executed. It is important to set the other block as finished using the function forceFinish() because a statement only gets finished if all blocks are marked as finished (see Section 3.2.3). To execute blocks, there are two important functions called *setOnFinish()* and *reset()*. Listing 4.6 shows the execution function of the FOR-LOOP statement. The *setOnFinish* function allows us to define a function which will be executed after a block has finished. The *reset* function will reset the contained components by setting its *isFinish* and *isActive* flag back to false (note

```
csComponentContainer.push({
   name: "cs.statement.if",
   description: "If INPUT is true block TRUE gets executed,
       block FALSE otherwise",
   inputs: [{
      name: "INPUT",
      type: "cs.type.Boolean"
   }],
   outputs: [],
   fields: [],
   image: "statement/if.png",
   blocks: [
      {name: "TRUE"},
      {name: "FALSE"}
   ],
   exec: function(state){

      // dependent on the input of the statement, either block
      // one or two gets executed
      if(state.inputs.item(0).getValue()){

         // set FALSE block finished
         state.blocks.item(1).forceFinished();

         // run TRUE block
         state.blocks.item(0).run();
      } else {

         // set TRUE block finished
         state.blocks.item(0).forceFinished();

         // run FALSE block
         state.blocks.item(1).run();
      }
   }
});
```

Listing 4.5: Code for IF/ELSE Statement

```
exec: function(state){

   // get number of repetitions from input socket
   var n = Math.floor(state.inputs.item(0).getValue());

   // initialize iterator
   this.currentN = 1;

   // after a block execution redo block (n-1) - times
   state.blocks.item(0).setOnFinish(
      function(state){
         this.currentN++;
         if(this.currentN <= state.inputs.item(0).getValue()){
            state.blocks.item(0).reset();
            state.outputs.item(0).setValue(this.currentN);
            state.blocks.item(0).run();
         } else {
            state.blocks.item(0).forceFinished();
         }
      },
      this,
      state
   );

   // write iterator to output socket
   state.outputs.item(0).setValue(this.currentN);

   // execute block the first time
   state.blocks.item(0).run();
}
```

Listing 4.6: Exec Function of FOR-LOOP Statement

that this has nothing to do with the reset function of the component definition object).

4.3.3 Primitives

Components are marked as primitives if there is an empty image property and exactly one field and output socket defined in the component definition object (see Listing 4.7 for an example).

4.3.4 Execution View and Reset Functionality

Regarding the component definition object in Listing 4.1, we still need to describe the two properties *view* and *reset*.

```
1   csComponentContainer.push({
2         name: "cs.default.primitive.number",
3         description: "This primitive represents a single
              number.",
4         inputs: [],
5         outputs: [{
6            name: "NUMBER",
7            type: "cs.type.Number"
8         }],
9         fields: [{
10           name: "NUMBER",
11           type: "cs.type.Number"
12        }],
13        image: "",
14        exec: function(state){
15
16           // get value from field
17           var n = state.fields.item(0).getValue();
18
19           // check type of field value
20           n = parseFloat(cs.isNumber(n)?n:0);
21
22           // write to output socket
23           state.outputs.item(0).setValue(n);
24        }
25     });
```

Listing 4.7: cs.primitive.Number Component Definition Object

```
1  csComponentContainer.push({
2     name: "cs.default.ide.textfield",
3     description: "This module allows user input through a
          textfield in the Execution View",
4     inputs: [],
5     outputs: [{
6        name: "TEXT",
7        type: "cs.type.String"
8     }],
9     fields: [],
10    image: "ide/textfield.png",
11    blocks: [],
12    exec: function(state){
13       // get value from text field in Execution View
14       var text = state.view.getValue();
15
16       // write result to socket
17       state.outputs.item(0).setValue(text);
18    },
19    view: {
20       // source of HTML textfield
21       source: "<input type='text' value=''/>",
22
23       /**
24        * @return value from Execution View's textfield
25        */
26       getValue: function(){
27          // $().val() is jQuery syntax
28          return $(this.getNode()).val();
29       }
30    }
31 });
```

Listing 4.8: Code for TEXTFIELD Module with Execution View

View To add an Execution View to a component, a property *view* is added to the component definition object. This view property needs an object with a *source* property. This source has to be of type string with valid XHTML code in it. For the structure of the HTML, it is important to have one HTML root node, but there are no further limitations to this source code. Listing 4.8 shows an example for the TEXTFIELD module. In this case, there is only one *input* element in the HTML source code. Adding the component TEXTFIELD will automatically add an HTML textfield as item to the Execution View. It is also possible to add custom functions to the view object as in the example of the textfield which declares a getValue() function. To access the HTML-node in the Execution View, you can call the *this.getNode()* function (see Listing 4.8 line 28). From the execution function you can call the view functions by accessing *state.view.custom_function()*, as done by the textfield component in Listing 4.8 on line 14.

Reset Sometimes it is useful to initialize or reset local variables of a component or reset parts of the view. For this purpose we have the reset function which get called after pressing the buttons run or repeated run. To illustrate this, look at the example in Figure 2.16: The random module will generate a number between 10 and 30. Then, the value will be added to the bar diagram. As we can see in the code of the bar module in Listing 4.9, the values are stored in a *local variable* called *this._container*. Executing the component will first add a new value from the input socket to the container and then display the container as a bar diagram. In the case of a *repeated run* the container is not reset in between successive runs of the script. The Execution View then shows a bar diagram like shown in Figure 2.16.

If no reset function is specified, old values never get deleted. The reset function has to clean up attributes of a component, as shown for the DIAGRAM module in Listing 4.9.

4.4 Adding new Types

New types can be added to the library. Listing 4.10 shows the example of adding the number type. In ClickScript types always have the prefix 'cs.type'.

```
1  if(!cs.library.hasType("cs.type.Number")){
2      cs.library.addTypeStructure("cs.type.Number","rgba
           (240,180,40,1)");
3  }
```

Listing 4.10: Code for Adding Number Type

This call adds the type *cs.type.Number* to the library. The first argument of the function *addTypeStructure* can be of type *cs.type.Structure* to add more complex types as described in Section 3.3.

4.5 ClickScript on the Web

One advantage of ClickScript is that it can be run on every web page very easily. To do so, you just have to copy the source code of ClickScript and an IDE HTML file to a web directory. A template for the HTML file can be found in Section C in the appendix. By adding your own style sheet you can change the look of the IDE and by removing div blocks in the body part you can remove parts of the IDE. For example to remove the tutorial block ⟨ *div id='csTutorial'* /⟩ has to be omitted. It is also possible to define the default libraries which should be loaded (Appendix C line 33). Further, you have to update all paths in the template. You have to make sure the module path (Appendix C line 12) for the cs module is set to the ClickScript source folder.

```
csComponentContainer.push({
   name: "cs.math.bar-diagram",
   description: "This module adds incoming VALUEs to a bar
       diagram.",
   inputs:
      [{
            name: "VALUE",
            type: "cs.type.Number"
         }],
   outputs:
      [],
   image: "math/bar-diagram.png",
   exec: function(state){

      // input value
      var newValue = state.inputs.item(0).getValue();

      // add new value to container
      this._container.add(newValue);

      // draw diagram
      state.view.draw(this._container);

   },
   view: {
      // empty background area, ready for bar diagram
      source: "<div style='width:200px;height:100px;border
          :1px solid black;'></div>",

      draw: function(a_container){
         /*draw bar diagram out of a_container*/
      }
   },
   reset: function(state){
      // initialize local variables
      this._container = new cs.util.Container();
   }
});
```

Listing 4.9: Code for DIAGRAM Module

4.6 Implementation State

ClickScript as described in this thesis is almost fully implemented. It is possible to develop programs and execute it. Not yet implemented are the graph check (cycle and two wires to one input check), a few mentioned libraries (e.g. Web Creation and File System) as well as most of the usability improvements mentioned in Section 3.4 (Multi Typing, Modularity, Smart Fields and Multi Language Support).

4.7 Firefox Add-On

The ClickScript add-on includes just the ClickScript code files and an IDE HTML. The HTML file is placed in the folder of Firefox add-ons and can be opened with privileged rights. This means, that we can bypass the *same origin policy* (e.g. for web automation) and access the users file system (e.g. downloading files).

Chapter 5

Case Study

5.1 Programming - People scripting ClickScript

5.1.1 Case and Expectation

So far we claimed that ClickScript is invented to introduce people to the world of programming. It is easy to use and convinces through its functional range. But is it really that interesting to work with ClickScript? Is it as easy and intuitive as predicted? In order to verify this assumption we did some tests programming ClickScript with different kind of people. These volunteers were in the range from an eleven year old scholar up to a person with an age of sixty. We also worked with people that had already some programming experience but most of them were non-programmers. The idea of this experiment was to get an impression of how long it takes, to people knowing anything about ClickScript, to write own small programs.

5.1.2 Run

There was no fixed procedure of working with these people. Normally we started by showing them a few simple examples like shown in Figure 2.1 and examples of how to do simple calculations with ClickScript. Afterwards they had to perform some exercises on their own. After a brief introduction to the Execution View, they manage to write an own implementation of a traffic light script (see example Figure B.1). The logic library (see Section A.2.6) was understood by most of the volunteers by just playing around with its different components and with SWITCH and different LIGHT components. After one or two hour of usage, most of them were able to write more advanced scripts. Most of the volunteers were also highly motivated to try their own ideas and did it right away.

5.1.3 Evaluation

In general the experiment was a great success. The basic operations of programming ClickScript, like adding and removing, wiring components, managing items in the Execution View and running scripts were understood by everyone in just a few minutes. People that already had experiences in programming just needed a few minutes more to understand the execution concept and compose any kind of scripts by reading the component descriptions. People that never programmed before, of course, needed more time to fully understand the concepts of ClickScript. But everyone of the volunteers really enjoyed playing with ClickScript. Most of them underlined how cool it is to program ClickScript due to the diversity of the components. From most of the volunteers we heard that they went on using ClickScript after the short testing session.

Therefore, ClickScript reached its goal of usability and proofed that non-programmers managed to learn to use ClickScript in only a few hours. Not covered by this small experiments was, how reasonable it is to learn ClickScript as a preparation to traditional, text based, programming languages. But the goal of motivating people to write their own programs is reached for sure!

We could imagine to use ClickScript in school to advance the logical thinking and as a motivation to the world of programming. In a next step it would be necessary to write serious teaching material with also a series of meaningful exercises.

5.2 Extendability - Combine with Web Things

5.2.1 Case and Expectation

The case study tested how easy it effectively is to add completely new components to ClickScript. The idea of this experiment was to combine the famous world of *Web of Things* [2] with ClickScript. The idea of Web of Things is to expand the web by adding smart devices like different sensors, household appliance and others, to the network. For our experiment we added the following two devices to the network: A plug (see 5.1 on the left) with a fan attached, and a temperature sensor (see 5.1 on the right). Web of Things now allows to control the plug and the temperature sensor over the network through HTTP requests. At the time running this case study there existed no ClickScript AJAX-request components at all.

5.2.2 Run

After only two hours the three components TEMPERATURE-SENSOR, PLUG and IS SMALLER were ready. The only challenge was to handle AJAX requests as asynchronous modules in ClickScript. The result of the experiment was a small clickscript which switched the fan on or off depending on the room temperature. The script is shown In Figure 5.2. The Web of Things modules need an network-address as input. If we run this clickscript in repeated run

Figure 5.1: Switch and Sensor

mode, the fan switches on if the room temperature is greater than 26 degrees. Of course the requests take some time, thus each run will take a few seconds.

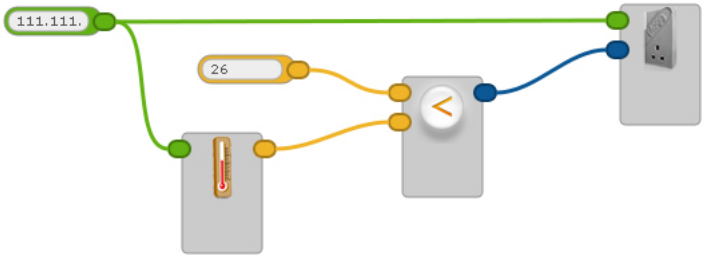


Figure 5.2: Script to Test 'Web of Thing' Components

5.2.3 Evaluation

It was a very successful case study. In only two hours an absolutely new type of components was added to ClickScript. Of course from now on it will take again less time to develop components using AJAX requests. Developing components with known techniques takes us less time. The development of the simple math operations library (addition, subtraction, division, multiplication, is-equal, is-greater) took less than one hour. Finding or designing a smart pictures for the different components might take few times longer.

Chapter 6

Conclusions

6.1 Conclusion

The programming language ClickScript is a really simple way to create own programs. ClickScript is a high level programming language. Therefore, it takes just a few clicks from the user generate exciting programs. Exciting in the sense that the running script includes user interaction and many different visual activities.

The extensible design of ClickScript allows adding new components very easily. There is a lot of space for further innovative components which will increase the popularity of ClickScript. In comparison to other programming languages, however, ClickScript is bound to the limits of browser technology. Since the new HTML standard 5.0 also includes audio and video integration, we are very confident that in the future the functional range of the browser will increase.

We have reached our goal of motivating non-programmers with a graphical appealing environment to develop their own fancy programs by just clicking and connecting a few pictures in the browser. Everyone who tested ClickScript was motivated and got into a 'let-me-try-this' fever.

6.2 Related Work

ClickScript is an easy to learn programming language and is suitable as an introduction to programming. It is a visual programming language with a lot of functionalities and is executed in the browser in the form of a webpage. In this section a few other programming languages related to these main characteristics of ClickScript are presented.

6.2.1 Kara

Kara [7] was developed by ETH as a programming environment written in Java. Its goal is also to introduce programming, but in this case based on finite state machines. Figure 6.1 shows the programming environment of *Kara the ladybug*. On the left part there is *the world of Kara* in which the user can place tree trunks and mushrooms which can be pushed around by Kara, as well as cloverleaves which can be picket up by Kara. On the right side of the Figure there is the window of the so called *Kara program editor*. The user can control its ladybug with the finite state machine in the upper half of the window. Like in ClickScript it is possible to 'write' a program by clicking. The action of a state can be manipulated in the lower half of the program editor. The figure shows a simple state machine for a program, in which the ladybug walks straight forward until it reaches a tree trunk, and picks up all cloverleaves on its way.

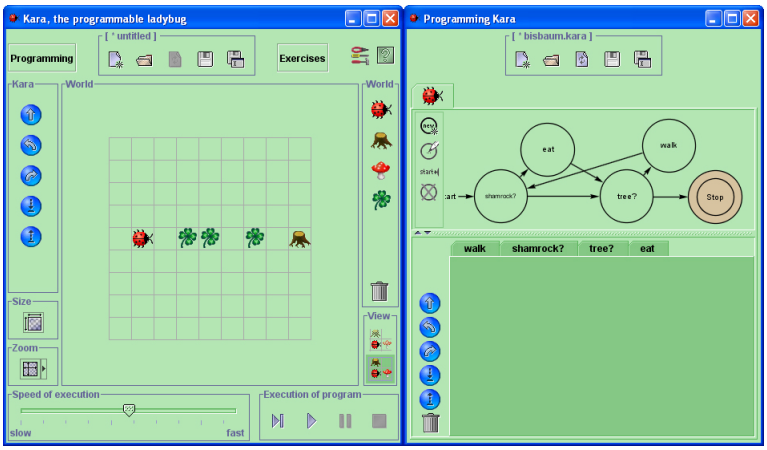


Figure 6.1: Kara's World and its Program Editor

6.2.2 Scratch

Another programming language focused on introducing programming especially to young people is called *Scratch* [8]. Scratch was developed and is still supported by the Lifelong Kindergarten Group at the MIT Media Lab to provide children (ages eight and up) access to the programmer's world. Scratch allows the user to program its own animations, games, music and drawings. First the user can add objects represented as pictures to the sprite library. Then it is possible to add one's own scripts to these objects. These scripts are called *scratch scripts*. Scratch scripts are developed by dragging and dropping blocks together to a stack as shown in Figure 6.2. These scripts can be activated by different *hat blocks* on top of the stack which trigger the execution.

This way the user can add new objects to the *stage* and move, rotate or resize them. As additional functionality there is the possibility to control a pen in the *stage* as well as playing different sounds. It is further possible to trigger keyboard and mouse events as well as using control blocks like *repeat until*, *if*

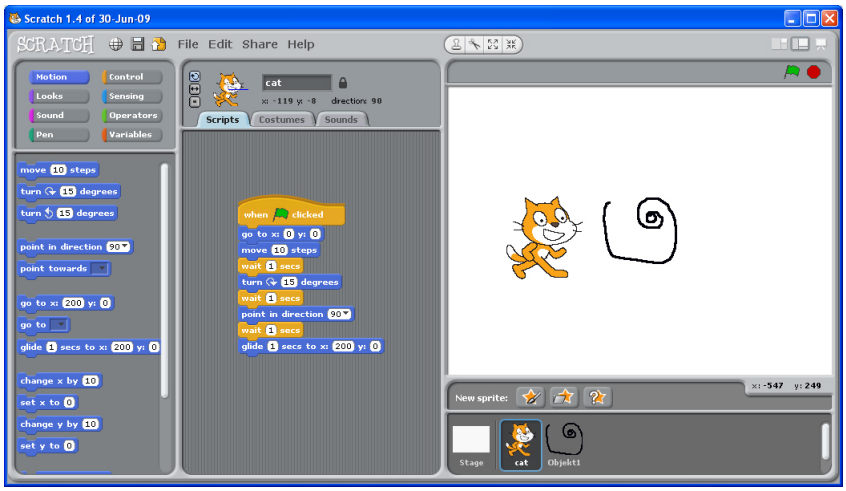


Figure 6.2: Development Environment of Scratch

and others. As an additional feature, the development environment allows you to publish your scratch script on the web as a Java Applet.

In comparison to ClickScript, Scratch is really an elaborated software and not a data flow but a control flow programming language. The scratch stack only defines the order of actions and always has its object as input. Therefore, the scripts do not allow to reuse outputs on several following blocks , whereas in ClickScript it is possible to connect an output socket of a component to several input sockets of other components. Also, in Scratch it is not possible to use one block as input to several other blocks. To be fair, Scratch also supports some kind of variables which can be set in one stage of the execution and reused later inside another block. Scratch clearly is focused on a very young audience whereas ClickScript due to its functional range and its extensibility is targeted also at adults.

6.2.3 Yahoo Pipes

As already mentioned at the beginning of this chapter, there are many characteristics based on which ClickScript can be compared to other programming languages. Let us switch to a true visual programming language approach in the browser, *Yahoo Pipes* [9]. This application can be used as an aggregator of web content (see Section 6.3). It uses the web as a huge data source and allows to process (e.g. concatenate, filter or sort) information from RSS feeds, web pages and XML-files. In Yahoo Pipes the user clicks his execution plan together and then the program will run on the server instead of in the browser. The execution on the server allows getting content of other webpages without caring about the *same origin policy* (described in 2.4.1).

As shown in Figure 6.3, the programming language is also a data flow language like ClickScript. Yahoo Pipes' components are powerful but also more difficult to use. Its main application is focused on fetching, processing and

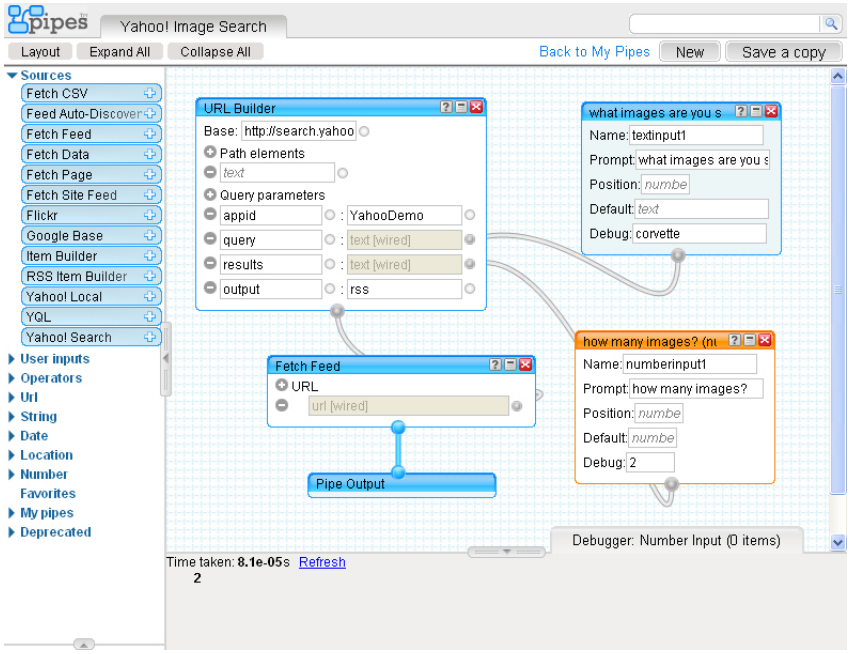


Figure 6.3: Yahoo Pipes Developing Environment with Script to Look for Images

republishing data.

6.2.4 LabView

LabView [10] (short for Laboratory Virtual Instrumentation Engineering Workbench) was developed in 1986 by *National Instruments* and is still a very successful and sophisticated language. LabView was originally invented to support non-programmers in linking data and lab equipment. In the mean time it has become more general. It is mainly used for data acquisition, instrument control and industrial automation.

ClickScript was designed to provide the programming concepts of LabView in the browser. LabView is also a data flow programming language. There are two workplaces called *Connector Panel* and *Front Panel* which correspond to the *Programming* and *Execution View* in ClickScript. In comparison to ClickScript LabView is compiled and not just interpreted. This allows LabView to achieve similar performance as other high-level programming languages. The data flow model allows a high degree of parallelization. Therefore, the programming language also includes multithreading and multicore support.

Since a software license of LabView is quite expensive, nobody would use it just for getting in touch with programming. On the other, hand if a LabView license is available, using LabView with an interface to hardware components (also supported for Lego robots: Lego Mindstorm) may be a very intuitive and interesting way to get in touch with programming languages. LabView allows to control motors and read sensors in the real world. Even such scenarios are

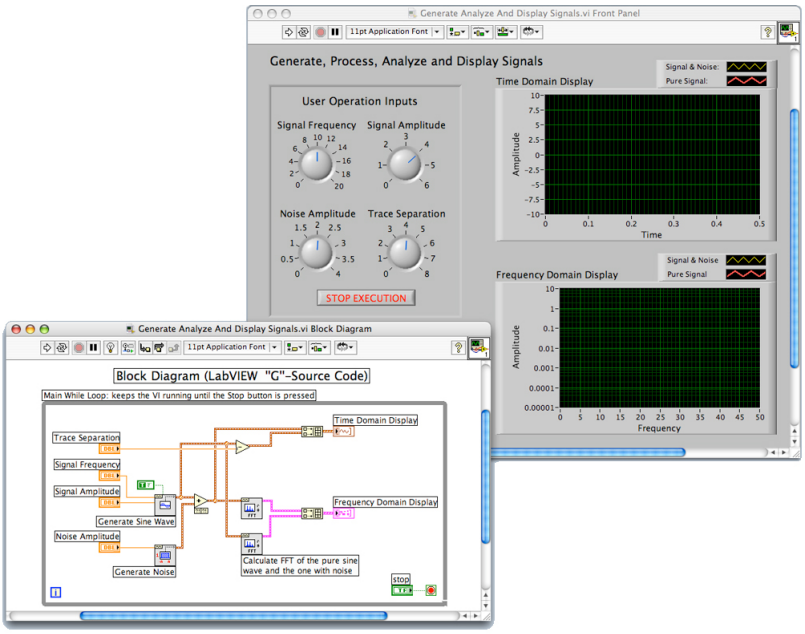


Figure 6.4: LabView Example with User Interface

possible in ClickScript by using Web of Things components are discussed in the case study in Section 5.2.

6.3 Future Work

Visualization of Invalid Wiring As mentioned in Section 4.6, until now only violation of rule 1 'not same type' and rule 2 'input to output' is visualized by the black dashed line. To make sure that each script is working, it is important the user gets informed about all invalid wirings. This also includes a cycle check, at latest before execution (violation of rule 4).

Control Flow Analysis There is another unsolved problem of invalid wiring. There are no rules to check control flow issues for the statement components. As an example for the statement SEQUENCE per definition the first block FIRST and afterwards block SECOND get executed. There is nothing to prevent the user from connecting an output of an output socket of the second block to an input of a component in the first block. Therefore, the component of the first block is waiting for the output of the component in the second block, but the second block will not be executed before the first block is finished. However, the other way around, connecting an output socket of a component to an input socket in a second block is a valid wiring. We see the same thing with the IF/ELSE statement where it absolutely makes no sense to connect components between blocks of the statement.

Save and Load Scripts A missing feature of the current implementation of ClickScript the possibility to save and load scripts. One way to realize this feature would be to add an additional observer to the data model which would record the events on the model to save a script, then an event player could rebuild the script. Another possibility would be to serialize the code structure. Maybe it would make sense to store the library too. The script structure including the library definitions could also be used to run a script placed on the web requiring the add-on's privileges. The add-on could copy the structure from the loaded web page to its playground.

Modularity As described in Section 2.4.2 it would be nice have a feature for grouping a few components to one subroutine represented as a new component. By clicking on it the subroutine could be unfolded back to its inner components.

Debugger The integrated debugging functionality allows to see what is going on in the worklist by means of text messages on the console. But it would be much more intuitive to see the values loaded to the wires directly in the Programming View. This also would have the great advantage of visualizing the coding principle for people learning ClickScript.

IDE There are lot of small things to integrate or fix in the ClickScript IDE, such as adding multilanguage and multityping support (see Section 3.4 for details). Also it would be nice to be able to zoom in and out of the Programming View as well as moving it in all directions.

Additional Libraries Section 2.4.1 mentions web creation and file system libraries, which are not yet implemented. But also libraries for database requests, RSS feed handling and sounds may be a nice idea to implement. As we pointed out earlier, ClickScript lives through its functional range. Therefore, having a lots of different components is a very important issue.

Firefox Add-On The add-on could be extended by a lot of useful functionalities. For the web automation it would be a great thing to get CSS selectors for a specific part of the page. This CSS selector could be evaluated through a tool implemented in the add-on which would mark parts by bordering HTML elements and showing their CSS selector. Also for this library it would be useful to have a web recorder for recording the browsing-history of a user and afterwards translating it to a clickscript which then could be manipulated manually. Another enhancement would be to save ClickScript options in the add-on.

Teaching Material As already mentioned in Section 5.1, we can imagine to use this programming language in high schools to motivate scholars to write or click together their own programs. For this purpose good teaching material is essential.

Appendix A

Components and Types

A.1 Statements

In this section all ClickScript components are listed. Keep in mind that a more advanced explication of the components can be found in the IDE of ClickScript by right clicking library elements or by clicking on the info button on a component in the Programming View. To reference the sockets of a component the writing rules are: *inNUMBER* means input socket on position NUMBER, same for the output sockets *outNUMBER* and fields *fldNUMBER* (e.g. *in1*: input socket 1, *in2*: output socket 2 or *fld1*: field 1). Counting starts at the top and if there is only one output socket the RESULT always is written to this socket.

Execution Views are listed separately in Section A.3.

A.1.1 cs.statement

FOR-LOOP (a) Executes block BODY *in1*-times. Iterator starting at 1 will be written to socket *out1* before each execution of the block.

FOR-EACH (b) Executes block BODY for each element of collection *in1*. The current element will be written to socket *out1* before each execution of the block.

UTIL-STOP (c) Executes block BODY until someone presses the stop button (see Section A.3(a)) on the corresponding Execution View.

IF/ELSE (d) Depending on *in1* block TRUE or FALSE executes.

SEQUENCE (e) First block FIRST and afterwards block SECOND executes.

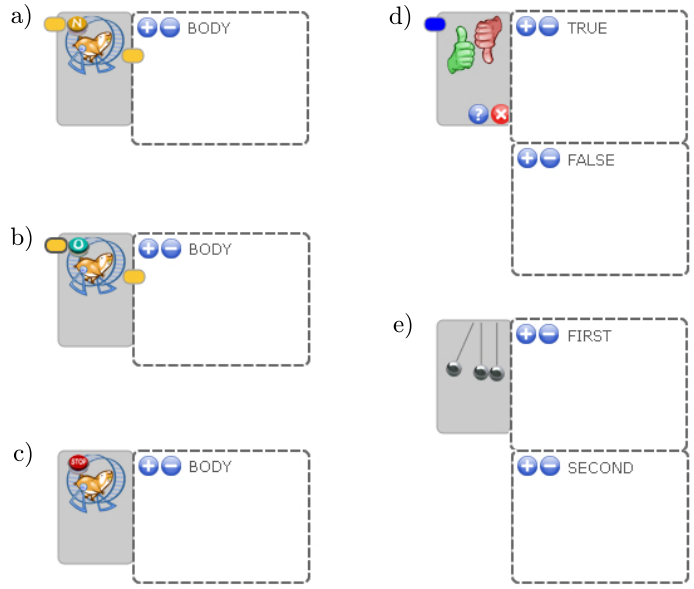


Figure A.1: Library 'cs.statement'

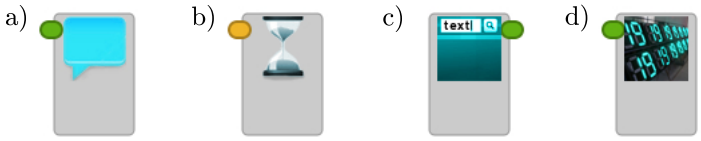


Figure A.2: Library 'cs.default.ide'

A.2 Modules

A.2.1 cs.default.ide

POPUP (a) Shows input string *in1* in a JavaScript alert-box.

WAIT (b) Interrupts program for *in1* seconds.

TEXTFIELD (c) Writes the input value from the HTML textfield (see Section A.3(b)) to *out1*.

DISPLAY (d) Displays short text messages read from socket*in1* to the display (see Section A.3(c)).

A.2.2 cs.math

ADD (a) Adds *in1* to *in2*.

SUBTRACT (b) Subtracts *in2* from *in1*.

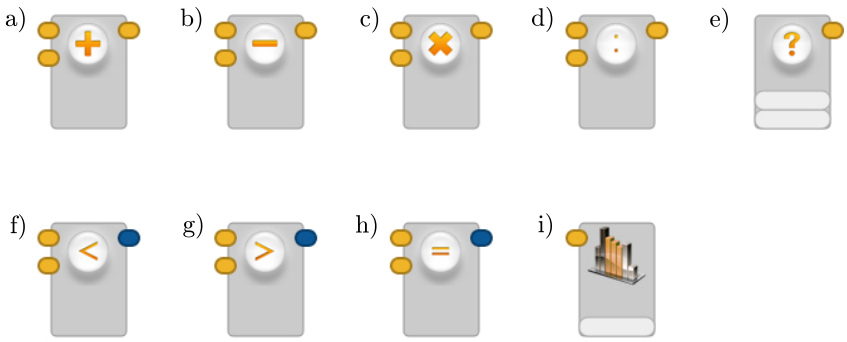


Figure A.3: Library 'cs.math'

MULTIPLY (c) Multiplies *in1* with *in2*.

DIVIDE (d) Divides *in1* by *in2*.

RANDOM (e) Returns a value between *fld1* (default: 0) and *fld2* (default: 100).

DIAGRAM (f) Stores incoming values *in1* to its internal container and displays this container as bar diagram (see Section A.3(d)). *Fld1* is optional to limit the number of displayed values.

IS SMALLER (g) True if *in1* is smaller than *in2*

IS GREATER (h) True if *in1* is greater than *in2*

IS EQUAL (i) True if *in1* is equal to *in2*

A.2.3 cs.web.browse

Figure A.4: Library 'cs.web.browse'

This library only works within the ClickScript Firefox Add-On.

OPEN (a) Opens the url *in1* in a mini-browser and forwards the mini-browser to *in2*.

SOURCE (b) Returns the source code of the mini-browser *in1* as string.

FILL-FORM (c) Takes the mini-browser *in1* and writes text *in2* into the mini-browser's $in3^{\text{th}}$ textfield.

CLICK (d) Takes the the mini-browser *in1* and presses the $in2^{\text{th}}$ button.

PRINT (e) Opens print dialog to print the page loaded in the mini-browser *in1*.

A.2.4 cs.web.things

This library only works within the ClickScript Firefox Add-On and with Web of Things sensors and controllers.

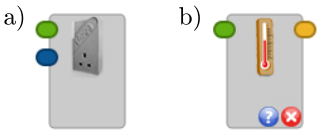


Figure A.5: Library 'cs.web.things'

SWITCH (a) Switches on or off a Web of Things switch on address *in1* according to the value *in2*.

TEMPERATURE (b) Returns temperature of a Web of Things sensor at address *in1*.

A.2.5 cs.collection

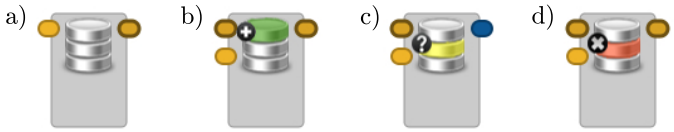


Figure A.6: Library 'cs.collection'

PIPE (a) Represents itself a collection. Adds a value *in1* to this collection.

ADD (b) Adds an element *in2* to the collection *in1*.

HAS (c) True if *in2* is an element of the collection *in1*.

REMOVE (d) Removes all elements *in2* from the collection *in1*.

A.2.6 cs.logic

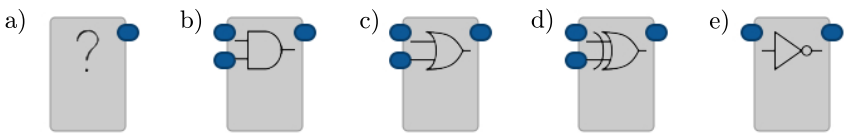


Figure A.7: Library 'cs.logic'

RANDOM (a) Returns randomly *true* or *false*.

AND (b) True if *in1* and *in2* both are true.

OR (c) True if at least *in1* or *in2* is true.

XOR (d) True if exact one of the inputs *in1* or *in2* is true.

NOT (e) Inverts *in1*.

A.2.7 cs.robotic

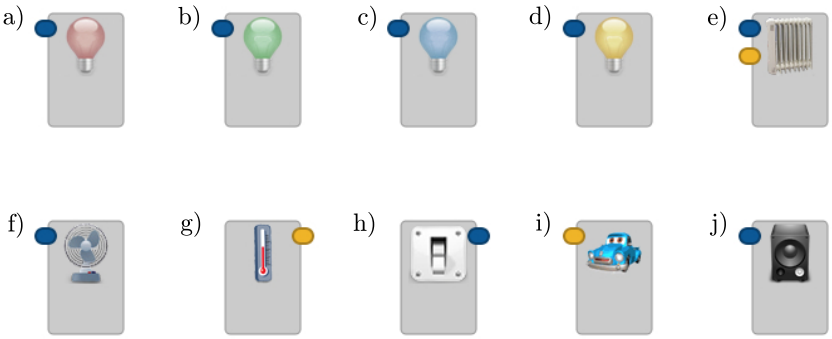


Figure A.8: Library 'cs.robotic'

RED-LIGHT (a) Switches on or off a red light in the Execution View (see Section A.3(g)) depending on *in1*.

GREEN-LIGHT (b) Switches on or off a green light in the Execution View (see Section A.3(g)) depending on *in1*.

BLUE-LIGHT (c) Switches on or off a blue light in the Execution View (see Section A.3(g)) depending on *in1*.

LIGHT (d) Switches on or off a yellow light in the Execution View (see Section A.3(g)) depending on *in1*.

RADIATOR (e) Heats the room with temperature *in2* if *in1* is true (see Section A.3(h)).

COOLER (f) Cools down the room with temperature 10°C if *in1* is true (see Section A.3(i)).

TEMP-SENSOR (g) Displays the room temperature in the Execution View and returns it (see Section A.3(j)).

SWITCH (h) Returns true if the switch in the Execution View is activated, false otherwise (see Section A.3(e)).

CAR (i) The car (in the Execution View A.3(k)) drives with a speed depending on *in1* (0:stop, 1: start, 2: slow, 3: fast).

SOUND (i) Plays a sound depending on *in1*. *Fld1* (default:1) allows to choose the sound (1: car horn, 2: ufo, 3: band saw, 4: machine).

A.2.8 cs.string

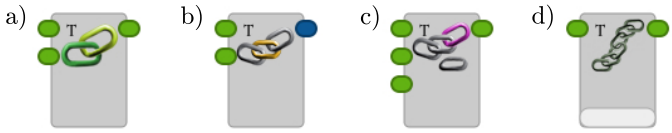


Figure A.9: Library 'cs.string'

CONCAT (a) Concatenates *in1* and *in2*.

MATCH (b) True if text *in1* contains *in2*.

REPLACE (b) Replace all *in2* in text *in1* by *in3*.

REPEAT (b) Repeats string *in1* *fld1* times (default: 0).

A.2.9 cs.converter

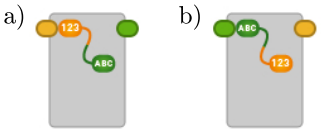


Figure A.10: Library 'cs.converter'

NUM2STR (a) Converts number *in1* into string *in2*.

NUM2STR (b) Converts string *in1* into number *in2*, if possible.

A.2.10 cs.web.misc

This library only works within the ClickScript Firefox Add-On.

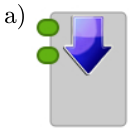


Figure A.11: Library 'cs.web.misc'

DOWNLOAD (a) Downloads a file from url *in1* to the local path *in2*. Current state is shown in its Execution View (see Section A.3(i)).

A.3 Execution Views

In this section all Execution Views of the components are listed. If there are more than just one view listed for the same label, the different views represents different states of the view. Further details on component functionalities can be found in the other sections in Appendix A.

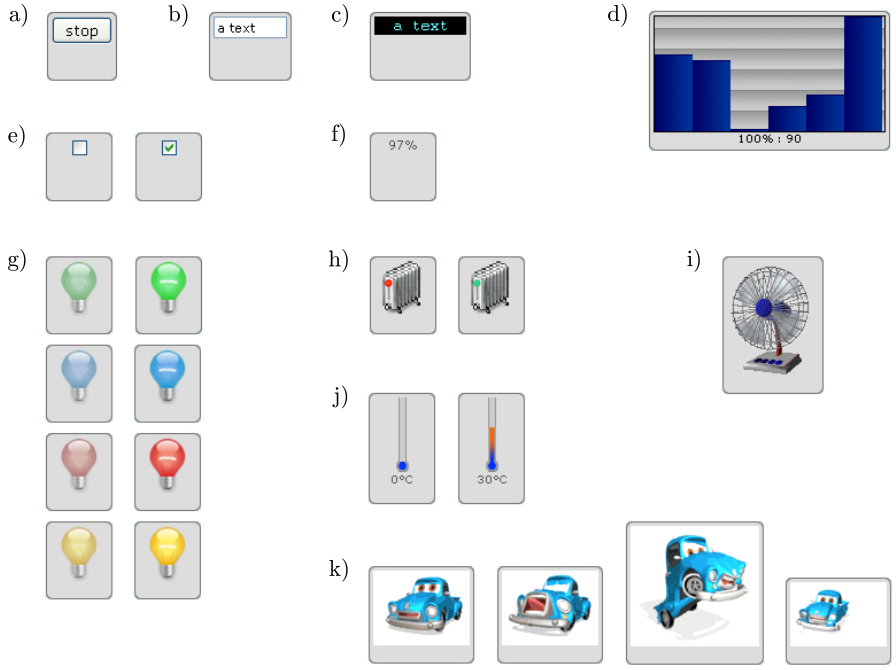


Figure A.12: Components Execution Views

UNTIL-STOP (a) Pressing the stop button will stop the until-loop.

TEXTFIELD (b) Textfield in which the user can input a text.

DISPLAY (c) Displays text.

DIAGRAM (d) Bar-diagram of the values. At the bottom at the right side of *100%* the value is shown which is chosen to represent the maximum of the y-axis.

SWITCH (e) Switch which can be switched on and off by the user.

DOWNLOAD (f) Shows current state of the download.

LIGHT (g) Series of all lights, first view represents a deactivated and second an activated light.

RADIATOR (h) First view represents the radiator as deactivated and the second one as activated.

COOLER (i) If the cooler is on, the picture is replaced by an animated-gif file which shows the cooler rotating its propeller.

TEMPERATURE (j) Temperature represented as a thermometer and as written value.

CAR (k) All four speeds: stop, start, slow and fast.

A.4 Types

Figure A.13: ClickScript Types

(a) Number (yellow), any number

(b) String (green), any text

(c) Boolean (blue), values ‘true’/‘false’ or ‘1’/‘0’

(d) Mini-browser (violet), CSS-id of an iframe

Appendix B

Example Programs

B.1 Traffic Light Example

This clickscript can be executed in repeated run. By switching on the switch in the Execution View the green light turns on and the car starts driving. By switching off the switch the car stops and the red light turns on.

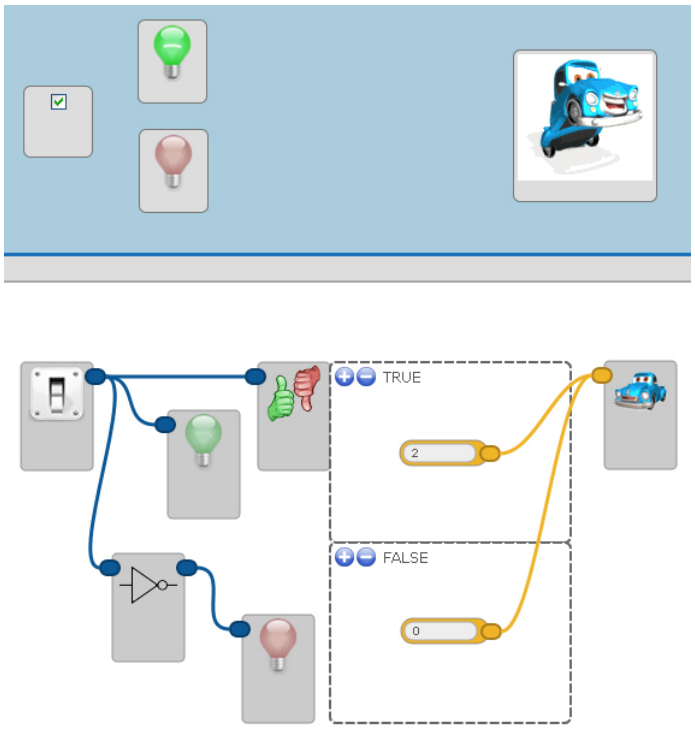


Figure B.1: Traffic Light Example

B.2 For-Each Example

Executing this clickscript, a collection with the elements '3','4' and '5' is created. Afterwards each element is shown in a popup.

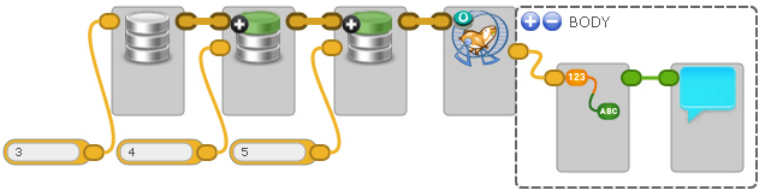
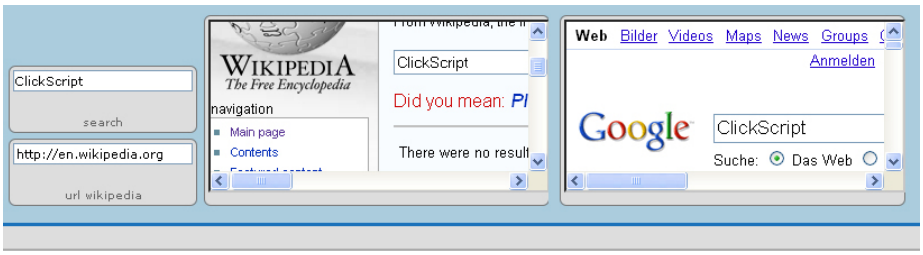


Figure B.2: For-Each Example

B.3 Web Example

This scripts looks for the search word in the 'search' textfield. First it tries to find the word in Wikipedia, if unsuccessful it enters the search word on the Google page. Anyway the page found gets printed. To check whether the word was found on Wikipedia or not, we check the source code for the string 'no results' which appears on a Wikipedia page if the word is not in the database.



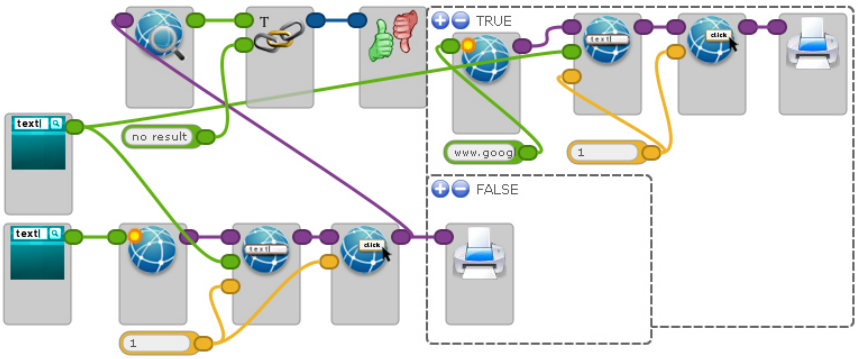


Figure B.3: Web Example

B.4 Temperature Example

Running this script in repeated run, the user can control the room temperature by switch on and off heating and coolers. The green light will turn on if the temperature is between 25 and 30 degrees.

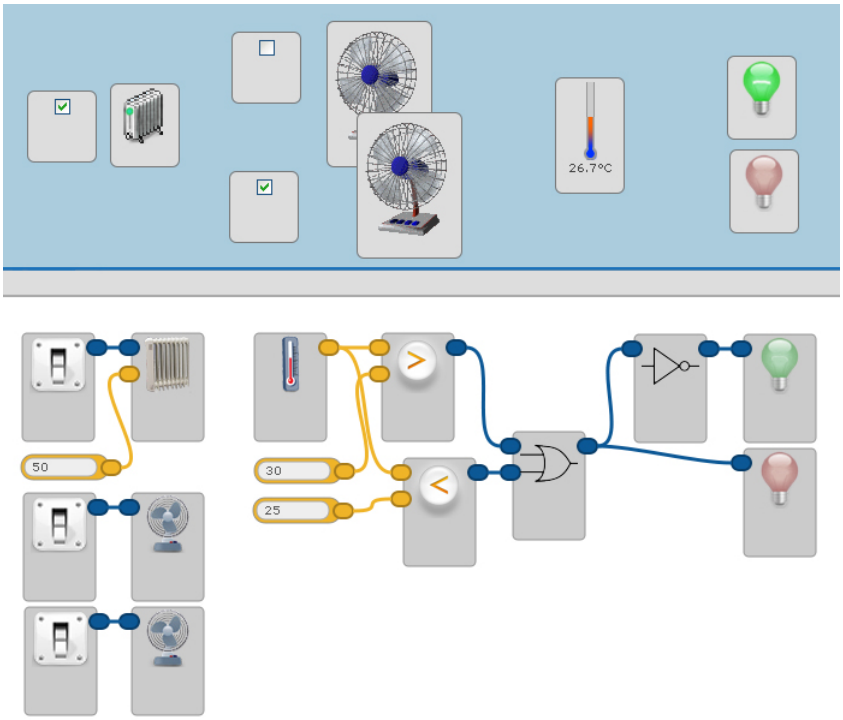


Figure B.4: Temperature Example

Appendix C

Web Template

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http
    ://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" lang=
    "en"
          xmlns:svg="http://www.w3.org/2000/svg"
          xmlns:v="urn:schemas-microsoft-com:vml"
          xmlns:xlink="http://www.w3.org/1999/xlink">
   <head>
      <title>ClickScript - IDE</title>
      <link rel="SHORTCUT ICON" href="./cs/util/images/
          logo32x32.gif"/>
      <!-- LOAD DOJO -->
      <script type="text/javascript">
            djConfig={
            parseOnLoad: true,
            isDebug: false,
            baseUrl: "./",
            modulePaths: { "cs": "cs/" }};
      </script>
      <script type="text/javascript" src="http://o.aolcdn.com/
          dojo/1.3.2/dojo/dojo.xd.js.uncompressed.js"></script
          >

      <!-- LOAD JQUERY -->
      <script type="text/javascript" src="http://ajax.
          googleapis.com/ajax/libs/jquery/1.3.2/jquery.min.js"
          ></script>

      <!-- LOAD IDE STYLES -->
      <link href="./cs/util/css/csIde.css" media="screen" rel=
          "stylesheet"/>

      <script type="text/javascript">//<![CDATA[
      // load ide
      dojo.require("cs.controller.IdeController");
```

```
28
29          // prepare component container and config object
30          if(!window.csComponentContainer){window.
                csComponentContainer = [];}
31          cs.config = {rootPath : ((djConfig && djConfig.
                modulePaths && djConfig.modulePaths.cs && djConfig.
                baseUrl) ? djConfig.baseUrl + djConfig.modulePaths.
                cs : "./lib/dojo/cs/" )};
32
33          // load libraries
34          dojo.require("cs.lib.default.init");
35          dojo.require("cs.lib.statement.init");
36          dojo.require("cs.lib.math.init");
37          dojo.require("cs.lib.string.init");
38          dojo.require("cs.lib.collection.init");
39          dojo.require("cs.lib.logic.init");
40          dojo.require("cs.lib.robotic.init");
41
42          dojo.addOnLoad(function(){
43             // init IDE
44             window.cs.ide = new cs.controller.IdeController();
45
46             // load custom modules
47
48             // show library
49             cs.ide.showLibrary();
50          });
51          //]]></script>
52       </head>
53       <body oncontextmenu="return false;">
54          <h1><img src="./cs/util/images/logo48x48.png" alt="logo"
                /><span>&amp;nbsp;&amp;nbsp;ClickScript &amp;#8211; IDE</span></
                h1>
55          <div id="csMenu">
56             <div id="csViewSwitches"></div>
57             <div id="csLib"></div>
58             <div id="csOptions"></div>
59             <div id="csExecutionView"></div>
60             <div id="csActions"></div>
61          </div>
62          <div id="csPlayground"></div>
63          <div id="csConsole"></div>
64          <div id="csTutorial"></div>
65          <div id="csExercise"></div>
66          <div id="footer">
67             &amp;copy; 2009 powered by <a href="www.clickscript.ch">
                   clickscript.ch</a>
68          </div>
69       </body>
70    </html>
```

Listing C.1: HTML Code for Web Template

List of Figures

Listings

Bibliography

[1] ClickSript Webpage. `http://www.clickscript.ch`.

[2] Dominique Guinard and Vlad Trifa. Towards the web of things: Web mashups for embedded devices. In *Workshop on Mashups, Enterprise Mashups and Lightweight Composition on the Web (MEM 2009), in proceedings of WWW (International World Wide Web Conferences)*, Madrid, Spain, April 2009.

[3] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *ECOOP '93: Proceedings of the 7th European Conference on Object-Oriented Programming*, pages 406–431, London, UK, 1993. Springer-Verlag.

[4] Scalable Vector Graphics (svg) 1.1 specification. `http://www.w3.org/TR/SVG11/`.

[5] The Official Dojo Documentation. `http://docs.dojocampus.org`.

[6] jQuery Documentation. `http://docs.jquery.com`.

[7] W. Hartmann, J. Nievergelt, and R. Reichert. Kara, finite state machines, and the case for programming as part of general education. *Human-Centric Computing Languages and Environments, IEEE CS International Symposium on*, 0:135, 2001.

[8] John H. Maloney, Kylie Peppler, Yasmin Kafai, Mitchel Resnick, and Natalie Rusk. Programming by choice: urban youth learning programming with scratch. pages 367–371, 2008.

[9] Yahoo Pipes Documentation. `http://pipes.yahoo.com/pipes/docs`.

[10] P. J. Moriarty, B. L. Gallagher, C. J. Mellor, and R. R. Baines. Graphical computing in the undergraduate laboratory: Teaching and interfacing with labview. *American Journal of Physics*, 71(10):1062–1074, 2003.