SPARKLY ARCHITECTURE DESCRIPTION DOCUMENTATION

Foreword

This is the foreword of the project “Sparkly” which is a block-based editor for SparkS language. Before the start it should be known that Sparkly editor is based on Blockly editor by Google in block based parts, and based on CodeMirror in text based parts which are open-source projects. What we wished to achieve was to create an editor for Spark Calibration technicians, to make them code easier. This is a web based editor, which uses Blockly as a base, and includes both traditional text-based programming and block-based programming. These two parts can interact with each other, and the users can use any part they want to use. To give more details about this interaction, both parts are converted into each other, so that users can code with blocks, then see the result in the text part, and vice versa.

Other than the main idea “blocks”, Sparkly includes many features like textual highlighting, auto-indentation, auto-completion, save/load, find/replace as most of the editors do. One of the goals of the project is to provide a standalone text editor to users. User can ignore the block part and use the editor as a traditional text based editor any time he/she wants.

The project is currently on development. Blocks can be turned into code at the moment. Turning code to blocks is the main focus of the development now. We are also working on diagnostic reporting, and trying to make blocks to code conversion more stable.

Now the foreword ends here and the details will be given in the proceeding parts of the document.

Components of Sparkly

Sparkly is made of 2 basic subsystems and 2 minor components. The main work takes place in these 2 subsytems, and then the subsystems communicates with these 2 minor components. Here are the components of the project. First of all, Sparkly works in a web browser either online or offline. We have two subsytems Block and Text based parts. Block based part and Text based part communicates with each other and the browser while running. We will also be using a parser of SparkS language, that takes input from textual part, parses it and then sends back the parsed version for diagnostic reporting and text-to-block conversion.

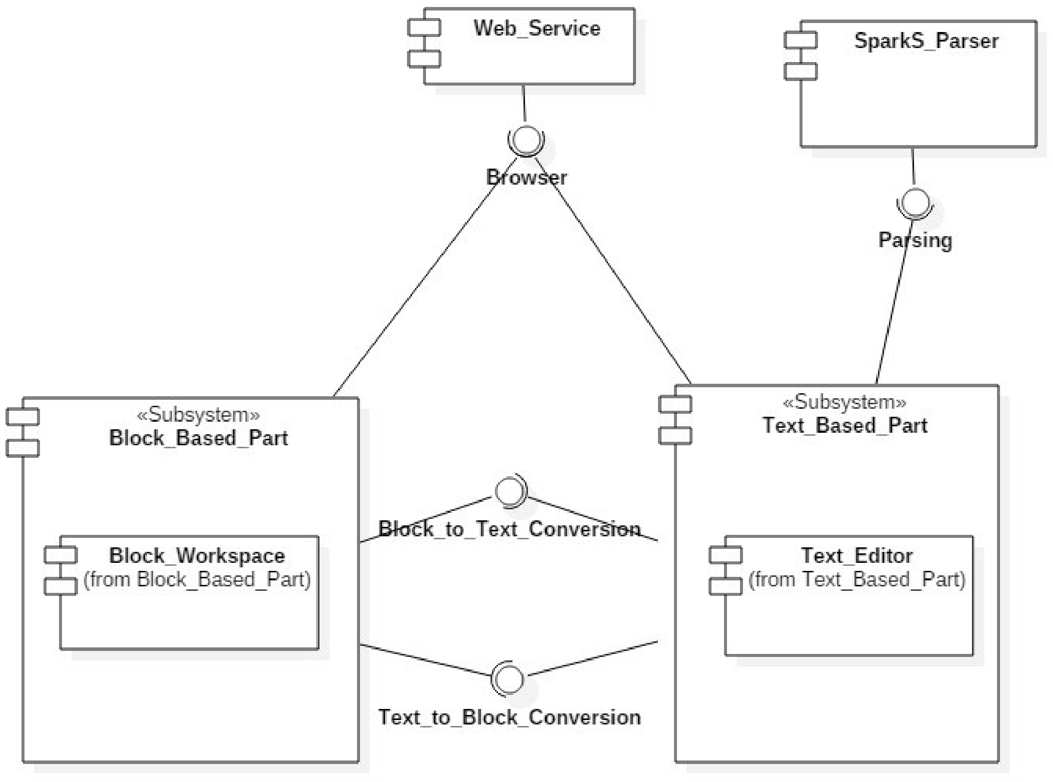
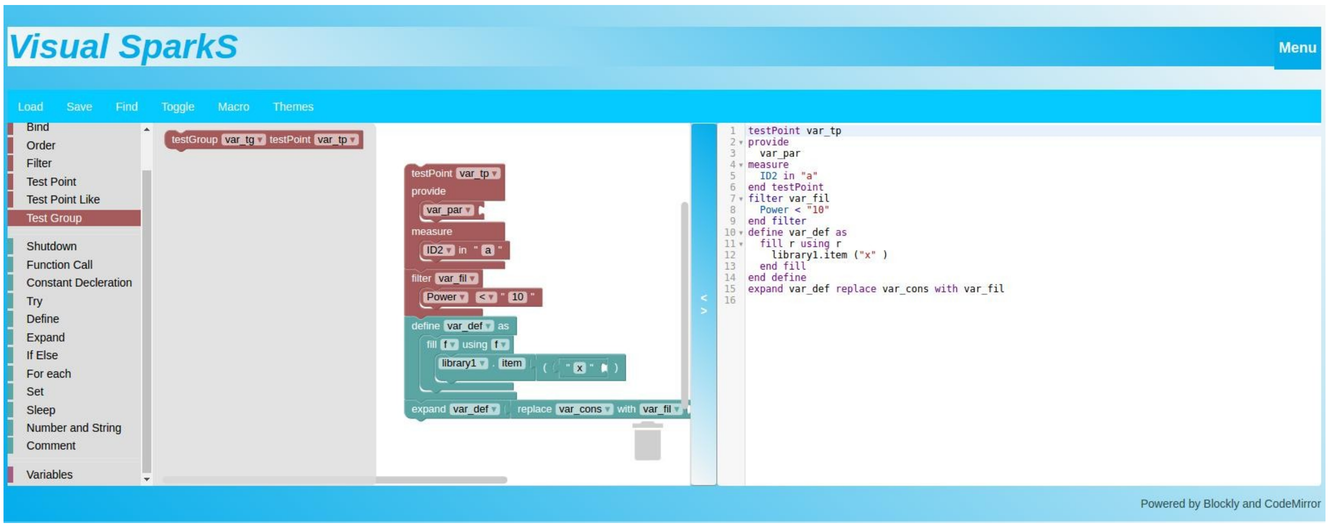


FIGURE 1: COMPONENT DIAGRAM OF SPARKLY

Usage

Sparkly includes two main panels and a toolbox on the screen. Left panel is the place where the blocks are placed, and on the right side lies the text editor. Blocks are dragged into left panel from the toolbox. Those blocks can be connected to give a more compact look. Making connected blocks mandatory is one of the ideas we consider for the future.



```
testPoint var_tp
provide
  var_par
measure
  ID2 in "a"
end testPoint
filter var_fil
  Power < "10"
end filter
define var_def as
  fill r using r
    library1.item ("x" )
  end fill
end define
expand var_def replace var_cons with var_fil
```

For the novice users, using blocks is a better option to code. All they have to do is to drag&drop the blocks to the left panel. The blocks contain input parts, signs or indicators to canalize users to enter correct values to correct spaces, and empty spaces that allows to attach other blocks.

- Inputs can be:
  - Just plain spaces that accepts any text, such as numbers, strings, and even the variable names if user wants to.
  - Dropdowns that gives a dropdown list to user, to fill the space. Those dropdowns contains pre-defined elements. But it may be possible to fill the dropdowns dynamically.
  - Variables that looks like dropdowns, but instead of listing the elements that are set previously, filled with all the variables created in the project.

So it means that even if the blocks are used only with drag&drop, user still save to enter parameters to the functions.

Original Blockly has a user interface that converts blocks into Python, JavaScript, Lua, Dart, and PHP. Our project uses one of Blockly’s demos, called generator demo. It has a simpler unser interface, and has the text editor with a larger size than the original Blockly.

Our blocks are basically divided into 2 main groups, header and body. Those types were previously defined in SparkS grammar, and all we had to do was to divide them visually, to show specific blocks together. Header functions should come before body functions, so we make it impossible to connect a header below a body block, and a body block above a header block.

Some blocks have voids inside of them to be filled with specific blocks only. Those specific blocks are in the same category in the toolbox to makethe user understand which block can fill the gap in the other block. We also restrained those blocks to be connected to the blocks other than selected blocks. So that those “sub blocks” can only be used to fill the gaps in other blocks from the same category. You can’t attach them to any other place.

Like typing in other editors, our editor colors the code to make it easier to distinguish code’s parts. It also automatically indents, and while writing for loops, if else conditions, it automatically completes the code to prevent user from wasting time on these issues. Users can select from 4-5 different color themes depending on their tastes.

Now let’s continue with how Sparkly works.

How Sparkly Works

Since we have two subsytems, we divided the workload into two main branches, one of them is text based part and the other one is block based part. In this section It will be given details, how we modified Blockly and how we used CodeMirror editor to create our own integrated project.

1) How Block Based Part Works

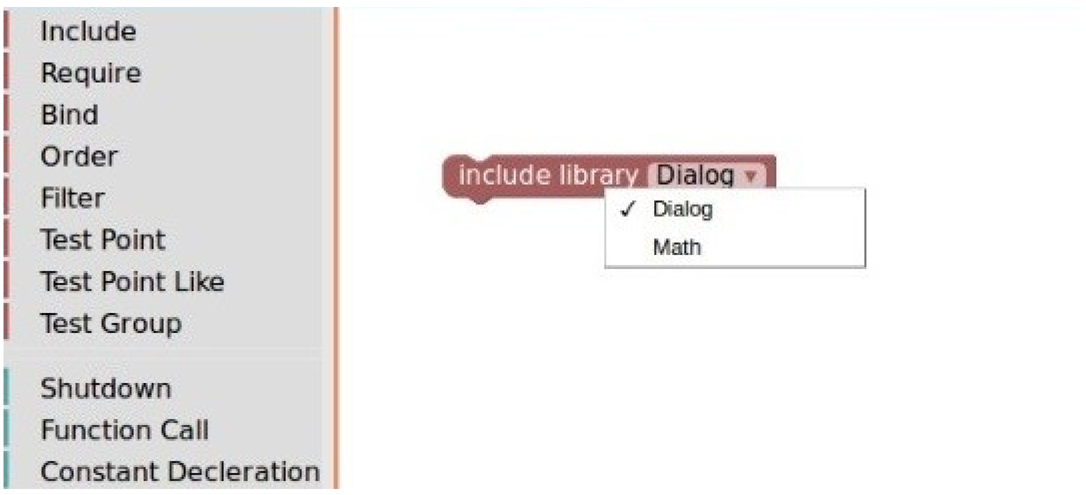
As mentioned before, our base is Blockly, and we are actually modifying only around 10% percent of whole project. Blockly project is made of a core that does most of the work, generators that that contains languages' grammars and code generators, demos and tests that provides different test fields to Blockly users, block generators that defines blocks' behaviours, and finally there are several folders that has minor roles on some extra features like language support.

Since the Blockly has a solid base, we just needed to define our own blocks, and language on blockly, and change the user interface just a little bit. Our work can be categorized into 3 groups, when it comes to modifying Blockly.

a) Blocks

In this folder, we have defined our blocks, how they interact with each other, what kind of inputs can a block take, what color should a block be, what will their dropdowns (if any) contain and etc. So the block files defines all the blocks in all categories, in a detailed version. Those files are written in JavaScript, and for each category (body and header in our case) there is one file like body.js and header.js.

Let's give some examples to be more clear about what these block files do. In usage part it's been mentioned that we have make it impossible to connect a body block to a header block from top, and vice versa. This option is accesible from block's files. You can add some tooltips to blocks, so that while user's mouse is hovering over a block, this tooltip becomes visible to give information to user. Block's color, name and general structure are decided in those files too. Let's give an example:



Here is a block named include. It has an indicator that gives the name of the block. It also has a dropdown that makes the user choose the library he/she wants to include. This is one of the simplest block. Now let’s see how it’s code that defines it seems:

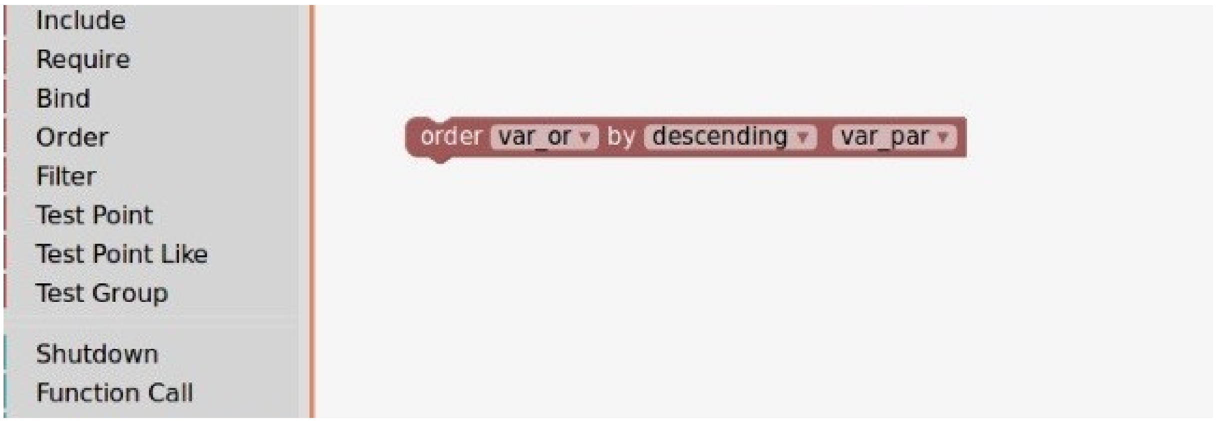
```
Blockly.Blocks['include_library'] = {
    init: function() {
        this.appendDummyInput()
            .appendField("include library")
            .appendField(new Blockly.FieldDropdown([["Dialog", "Dialog"],
["Math", "Math"]]), "NAME");
        //.appendField(new Blockly.FieldVariable("<item>"), "NAME");
        this.setInputsInline(false);
        this.setPreviousStatement(true, 'header');
        this.setNextStatement(true, ['header','body']);
        this.setColour(0);
        this.setTooltip('');
        this.setHelpUrl('http://www.example.com/');
    }
};
```

Originally Blockly contains block files like, lists.js, logic.js, math.js, text.js, variables.js for their corresponding functions. Since SparkS is a domain specific scripting language, we just need the functions that is specified by Spark Calibration workers.

b) Generators

Now that we saw what is needed to make a block appear. We also need to produce code as output from those blocks. For this we have generator files. Basically a language is defined in blockly with 2 types of files. A language needs a LANGUAGE.js file that gives basic rules of a language, keywords, operation priorities etc. Then a language needs CATEGORY.js files that corresponding with its blocks. These category files will define the output that blocks will generate, in other words, the code.

As mentioned before, blocks have inputs, dropdowns, etc. As the block definitions has to define what does these special parts of the block do, generators should define what is the output of these special parts. Now let’s a look at how a block’s generator seems:



This block has 3 parameters. First the name of the order, it can be thought as an object, which is an order type object, and var_or is the name of this order variable. Descending dropdown decides whether this order will be in ascending or descending order. Final parameter named var_par is the attribute that this order will be made. Let's consider this is a special ordering function that orders Pokemons. Var_par is the attributes of the pokemons like speed, attack, defense etc. To create a code from this block, we need to take it's parameters, manipulate them to be a meaningful statement in SparkS language. Here is the code for order's generator:

```
Blockly.SparkS['order'] = function(block) {
    var variable_name1 =
Blockly.SparkS.variableDB_.getName(block.getFieldValue('name1'),
Blockly.Variables.NAME_TYPE);
    var dropdown_name = block.getFieldValue('NAME');
    var variable_id =
Blockly.SparkS.variableDB_.getName(block.getFieldValue('ID'),
Blockly.Variables.NAME_TYPE);
    var code = 'order ' + variable_name1 + '\n  ' + dropdown_name + ' ' +
variable_id + '\nend order \n';
    return code;
};
```

The parameter that holds var_or in the example is put into variable_name1, descending is put into dropdown_name, and finally var_par is put into variable_id. Then we concatenate those variables according to function's syntax. The result of this generation will be:

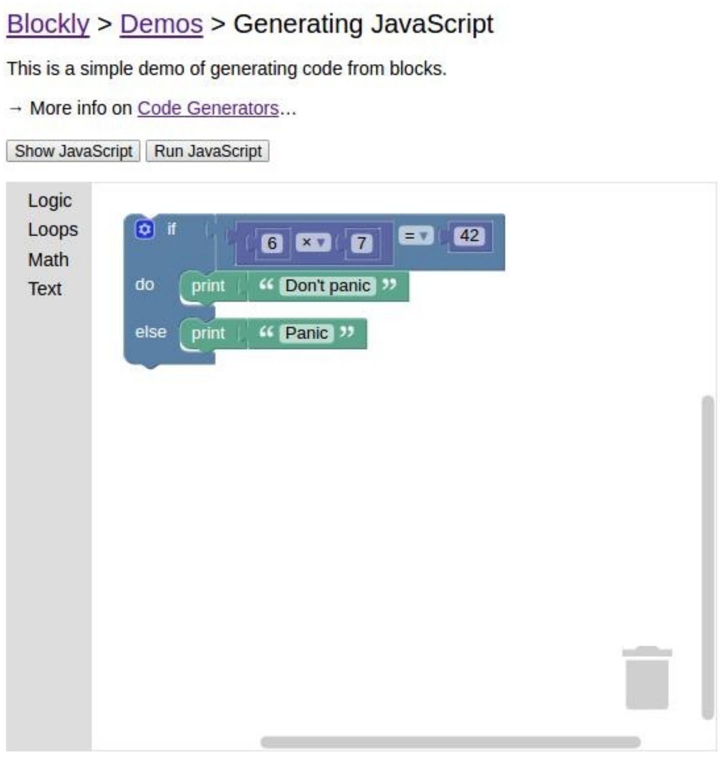
```
        order var_or
                descending var_par
        end order
```

This generated code will appear in text editor along with other blocks' codes. It's easier to use blocks sometimes so that user will save time on coding. On the other hand if user can write a correct statement in SparkS language, it will be converted into the corresponding block.

c) Demo

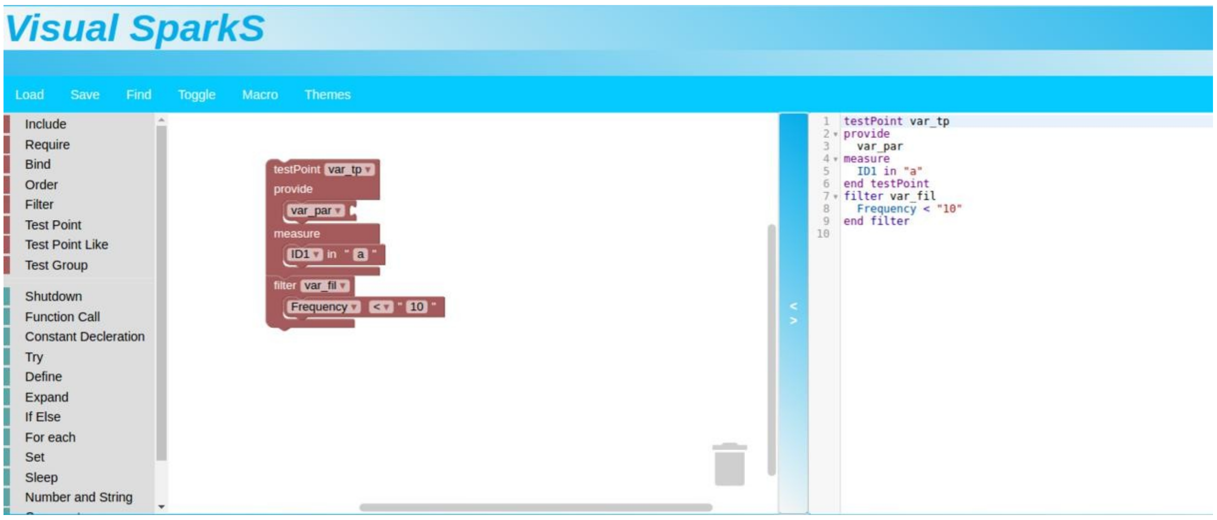
After setting block and generator files correctly, we need a playground to use our blocks to create SparkS script. Here is where it happens. Originally the demo, called index.html, opens up the browser, and let's the user to use blocks to generate code. When the demo opens, we can see that it contains the toolbox filled with categories of Blockly (math, logic, loop, text, etc.) a left panel for placing blocks, and no right panel to show the code. This text part was read-only, actually just an alerting window that shows the code, and users were able to interfere with the code only via the blocks. But we have modified this demo, so now both parts are editable, which is the most basic goal of the project.

Through the time this demo evolved into out main application. Originally it was made to generate JavaScript and then run it. Now it includes everything that we've done so far. Here are the pictures of old generator demo and the new respectively:


Blockly > Demos > Generating JavaScript

This is a simple demo of generating code from blocks.

→ More info on Code Generators…

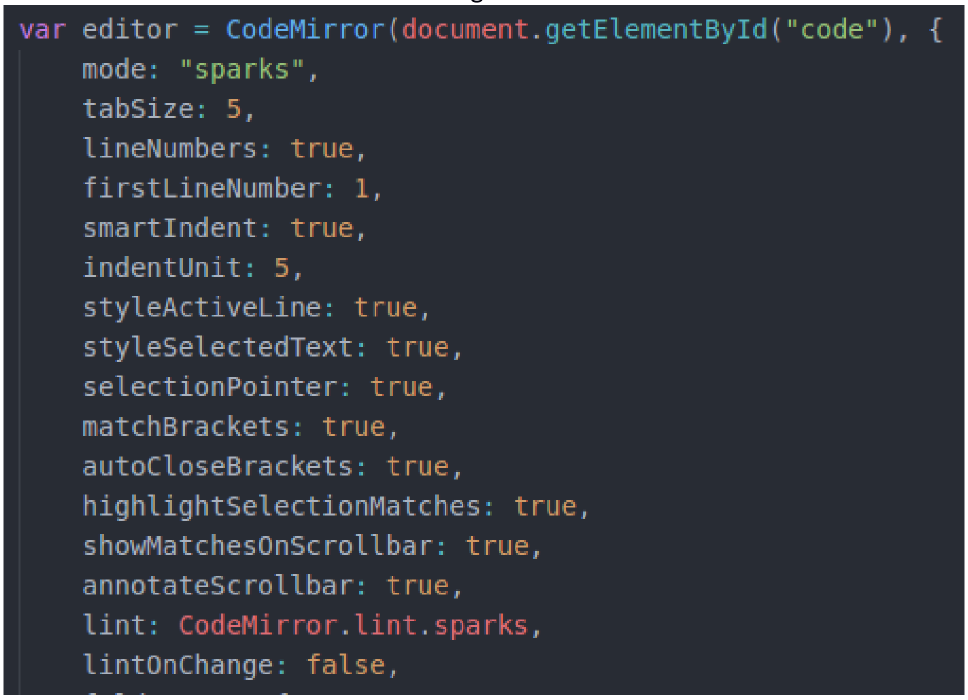


```
testPoint var_tp
provide
  var_par
measure
  ID1 in "a"
end testPoint
filter var_fil
  Frequency < "10"
end filter
```

For now it's possible to say that nearly everything that we've done in this project, other that block and generator files, are in this demo's html file. Even if it will change in the future, blocks from codes are generated here, categories and the blocks of categories included to the wokrspace here, and of course the design of the user interface is implemented here.

2) How Text Based Part Works

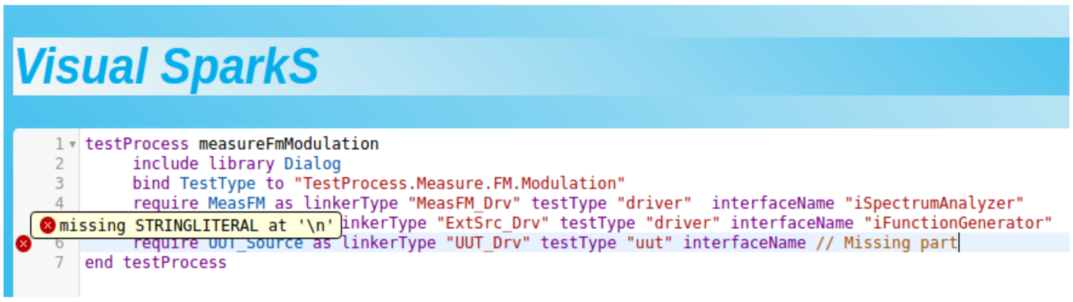
For the textual part, we are using CodeMirror which is an open-source text editor implemented in JavaScript. In fact, we have integrated CodeMirror text editor with the block based part without changing the core structure and the main functionality of CodeMirror. Also, we added a new programming language mode to CodeMirror to be able to make the features of CodeMirror such as indentation, highlighting and code folding compatible with SparkS domain-specific-language. To use the features of CodeMirror, an editor instance should be defined in HTML file. In this definition, the desired features provided by CodeMirror should be enabled to use them. For instance, a part of our text editor definition is like the following:



```
var editor = CodeMirror(document.getElementById("code"), {
    mode: "sparks",
    tabSize: 5,
    lineNumbers: true,
    firstLineNumber: 1,
    smartIndent: true,
    indentUnit: 5,
    styleActiveLine: true,
    styleSelectedText: true,
    selectionPointer: true,
    matchBrackets: true,
    autoCloseBrackets: true,
    highlightSelectionMatches: true,
    showMatchesOnScrollbar: true,
    annotateScrollbar: true,
    lint: CodeMirror.lint.sparks,
    lintOnChange: false,
```

a) Diagnostics Reporting

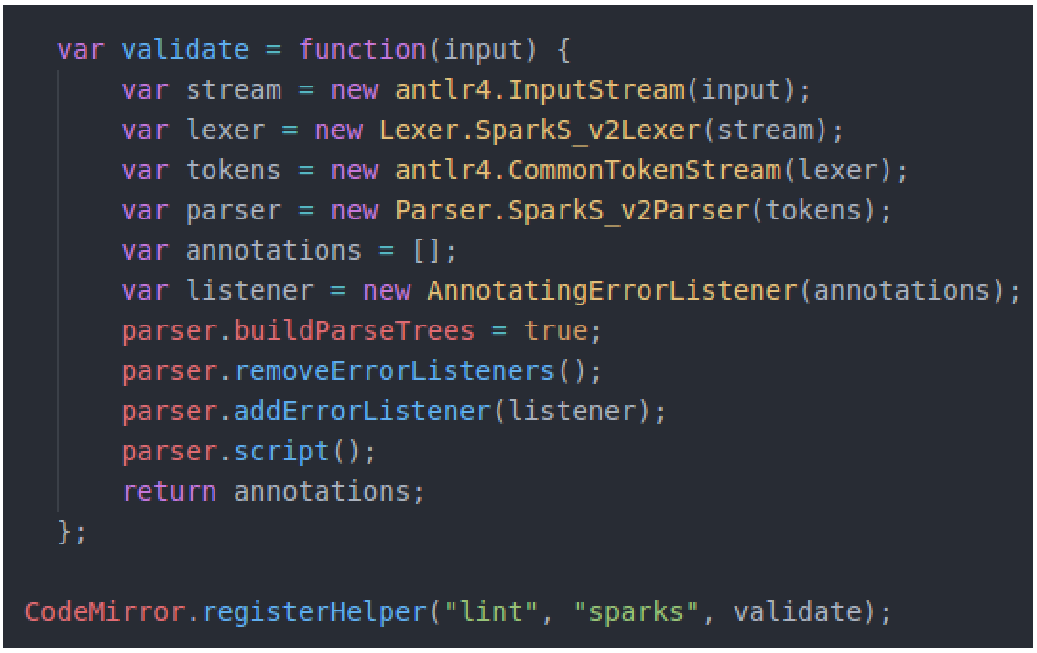
In our text editor, we have diagnostics reporting feature which provides to user the ability to see the errors of the code written in the text editor. To be able to implement this feature, we used ANTLR4 parser of SparkS and integrated that with the linting function of the CodeMirror which is compatible with some well-known programming languages such as HTML, JavaScript and CSS. The process of finding the errors is basically as follows: When the user writes some code, the parser interactively passes over the code, and checks it by using a listener. When it finds an error, it appends the error to an array of errors with its information about line number and the message generated by the parser. After the parsing has finished, the text editor shows the errors on the left with an error icon and the corresponding parser message by using linting function of CodeMirror. An example usage of diagnostic reporting is shown below.



```
testProcess measureFmModulation
     include library Dialog
     bind TestType to "TestProcess.Measure.FM.Modulation"
     require MeasFM as linkerType "MeasFM_Drv" testType "driver"  interfaceName "iSpectrumAnalyzer"
                       inkerType "ExtSrc_Drv" testType "driver" interfaceName "iFunctionGenerator"
     require UUT_Source as linkerType "UUT_Drv" testType "uut" interfaceName // Missing part
end testProcess
```

missing STRINGLITERAL at '\n'

To be able to make diagnostic reporting work, it is important to integrate the parser properly. For this purpose, a validation function is needed to configure the parser's components and to start parsing operation in a proper way. Moreover, an error listener should be generated which is a feature of ANTLR in order to be able to catch the errors during the parsing process. Lastly, parsing operation should be started by calling the start rule function (parser.script() in the code below) which is defined in the ANTLR grammar of the language. As a result, parser integration has been done as the following:



```
  var validate = function(input) {
      var stream = new antlr4.InputStream(input);
      var lexer = new Lexer.SparkS_v2Lexer(stream);
      var tokens = new antlr4.CommonTokenStream(lexer);
      var parser = new Parser.SparkS_v2Parser(tokens);
      var annotations = [];
      var listener = new AnnotatingErrorListener(annotations);
      parser.buildParseTrees = true;
      parser.removeErrorListeners();
      parser.addErrorListener(listener);
      parser.script();
      return annotations;
  };

CodeMirror.registerHelper("lint", "sparks", validate);
```

b) Maintenance

In this section, the necessary information about how to maintain and modify Sparkly will be given by referencing the design and the structure of the corresponding part. Like the other parts in this document, this section also contains two main parts one of which is block based part maintenance and the other one is text based part maintenance. The maintenance issues about block based part was stated in detailed in the section "How Block Based Part Works".

First of all, it is important to know the file structure of the text based part in order to understand how the things work. In the folder that contains the index file, the folders and files that make text editor work properly are listed below with the necessary explanations:

- addon : This is CodeMirror's addon folder which provides simple editor features like highlighting, indentation and searching. The detailed information about these files can be found in CodeMirror's online manual.
- antlr4 : This is the runtime files of Antlr4. It was downloaded from Antlr's website.
- generated-parser : This folder contains the corresponding js files of the generated parser based on the grammar of SparkS. When the grammar file changes, these files should be regenerated.

- lib : This folder contains the necessary libraries which are CodeMirror's main js file and require.js library which is used to be able to include a js file in another js file.
- mode : This folder is for language modes supported by CodeMirror. In mode folder, there is a folder "sparks" in which we define SparkS language for CodeMirror. When a change in SparkS occurs like a new keyword addition, this definition should be changed accordingly.
- rudimentary : In this folder, there are again CodeMirror's files of some simple features like addon folder. However, the files in this folder were changed according to the requirements of the project. Moreover, there is a folder, namely "lint", which contains the implementation and the style of diagnostics reporting feature. For the files in rudimentary folder, CodeMirror's online manual is still valid.
- index.html : All features stated above are used in this file. Also, the corresponding explanations about the code were written as comments on top of the corresponding part. This file also contains the definition of the editor instance as seen in the screenshot at the beginning of the "How Text Based Part Works" section. This definition can be changed according to the decision of the user.

Like any other editor, we have also some predefined keyboard shortcuts which can ease the programmer's job. The definitions of these shortcuts can be found in the HTML file. Below the definition of the editor instance, there is a part "extra keys" which is used to define extra shortcuts apart from the ones defined by CodeMirror. CodeMirror's shortcuts can be found in "lib/codemirror.js" file in which there is a list, "keymap" containing the default shortcuts. Currently, defined shortcuts apart from CodeMirror's shortcuts are like the following:

✓ "Ctrl+Alt+A" : Turn on/off Autoconversion

✓ "Ctrl+Alt+B" : Convert to Block

✓ "Ctrl+Alt+C" : Convert to Code

✓ "Ctrl+L" : Load File

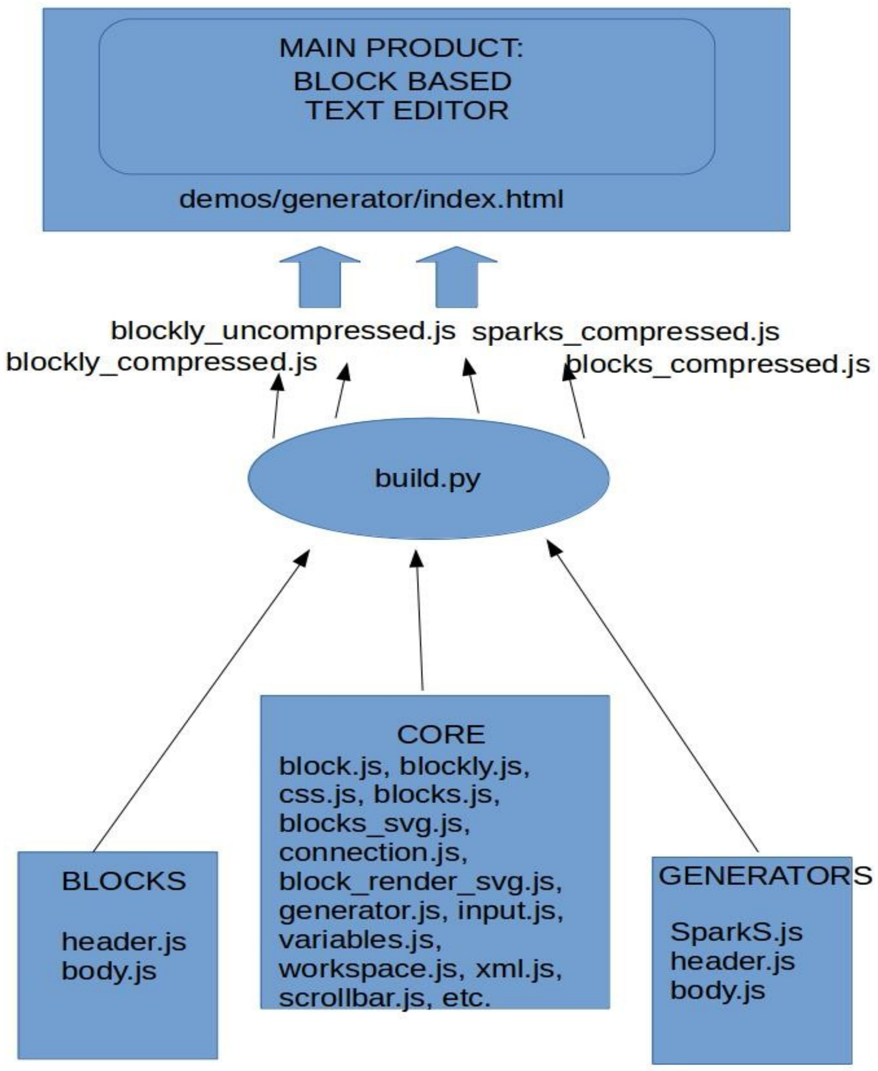
✓ "Ctrl+S" : Save File

✓ "F11" : Fullscreen

Connections Between The Files

Blockly has a complex structure that has lots of dependencies in each file. It was difficult to get over problems sometimes, because of this depencies. To change a functions structure, you also need to check every other function that it uses, and every other function that use it. So sometimes it took months to get results in specific areas.

There's a python file called build.py, which is like the compiler of the project. It compresses generator, core and block files together to create just one file called SparkS_compressed.js, in our case. Most of the other files includes these kind of compressed files so it's important to build the project before the changes are applied. It needs Google's closure library to build the project. Now let's show the relationships in files with a basic diagram.


MAIN PRODUCT:
BLOCK BASED
TEXT EDITOR
demos/generator/index.html
blockly_uncompressed.js
sparks_compressed.js
blockly_compressed.js
blocks_compressed.js
build.py
BLOCKS
header.js
body.js
CORE
block.js, blockly.js, css.js, blocks.js, blocks_svg.js, connection.js, block_render_svg.js, generator.js, input.js, variables.js, workspace.js, xml.js, scrollbar.js, etc.
GENERATORS
SparkS.js
header.js
body.js

Functions of Sparkly

In this section the functionalities of Sparkly will be given in details with the usage of sequence diagrams with use case scenarios.

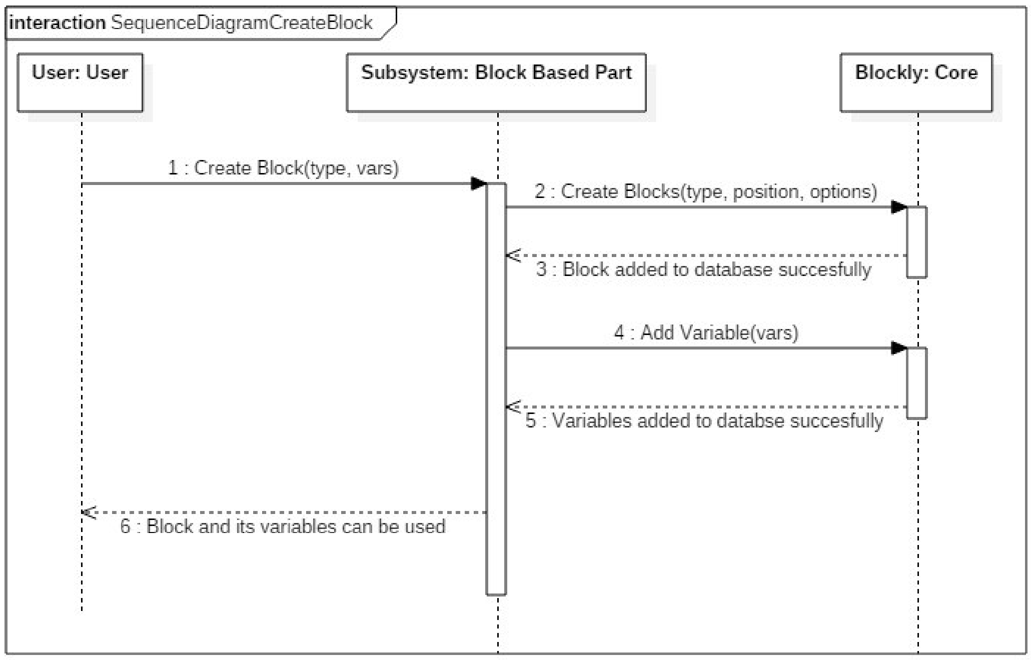

interaction SequenceDiagramCreateBlock
User: User
Subsystem: Block Based Part
Blockly: Core
1 : Create Block(type, vars)
2 : Create Blocks(type, position, options)
3 : Block added to database succesfully
4 : Add Variable(vars)
5 : Variables added to databse succesfully
6 : Block and its variables can be used

Figure 2a: Sequence Diagram of Creating Block

Use Case Name	Create Block
Use Case ID	SD1
Included Use Cases	None
Primary Actor	User
Description	User creates a block with parameters (if there are any), the block is added to the workspace.
PreCondition	None
Trigger	User drags&drops a block to the workspace
Main Success Scenario	1. User drags&drops a blocks to the workspace 2.User fills the spaces in the block woth desired variables, values, constants etc. 3.The block is added to the workspace, and if there are any variables inside, its added to variables database
Alternative Scenario	If the generators and blocks are coded without a mistake, there's no alternative scenario that leads this case to any kind of failure
Post Condition	User can use this block from now on

Figure 2b: Use Case Scenario of Create Block

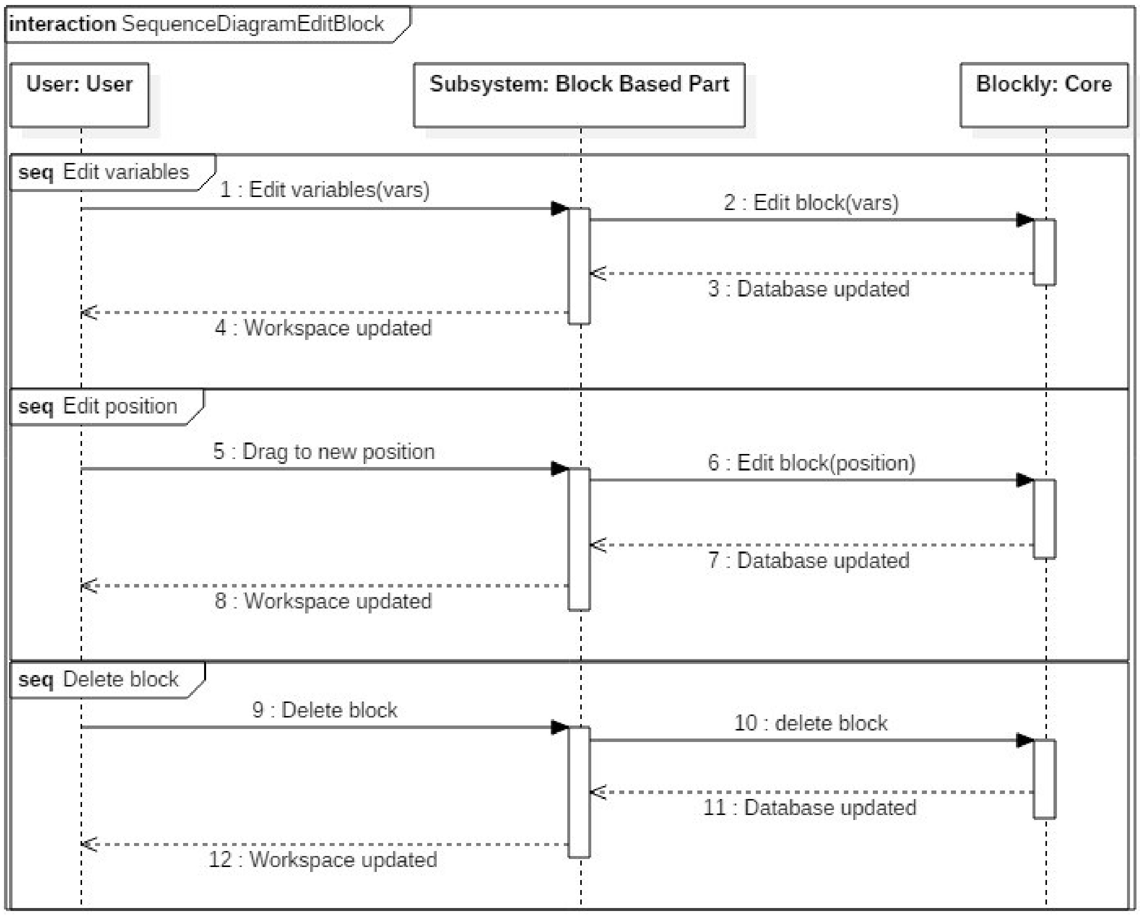


Figure 3a: Sequence Diagram of Editing Block

Use Case Name	Edit Block
Use Case ID	SD2
Included Use Cases	SD1
Primary Actor	User
Description	User can edit the position and the parameters of the block, or delete it
PreCondition	There should be a block to edit or delete
Trigger	User can drag to another place to move the block, can re-enter values to edit the parameters, and drag the block to the trashcan icon to delete the block
Main Success Scenario	1a. User Drags the block to another space in the workspace 2a. Block's position is Changed 1b. User changed the values in the block 2b. Database is updated with the new values 1c. User drags the block to the trachcan icon 2c. block is deleted from the workspace
Alternative Scenario	If user unintentionally drags the block to the trachcan, it can delete the

	BLOCK EVEN IF IT'S NOT MEANT TO BE
POST CONDITION	WORKSPACE IS UPDATED WITH THE DESIRED CHANGES

FIGURE 3B: USE CASE SCENARIO OF EDIT BLOCK

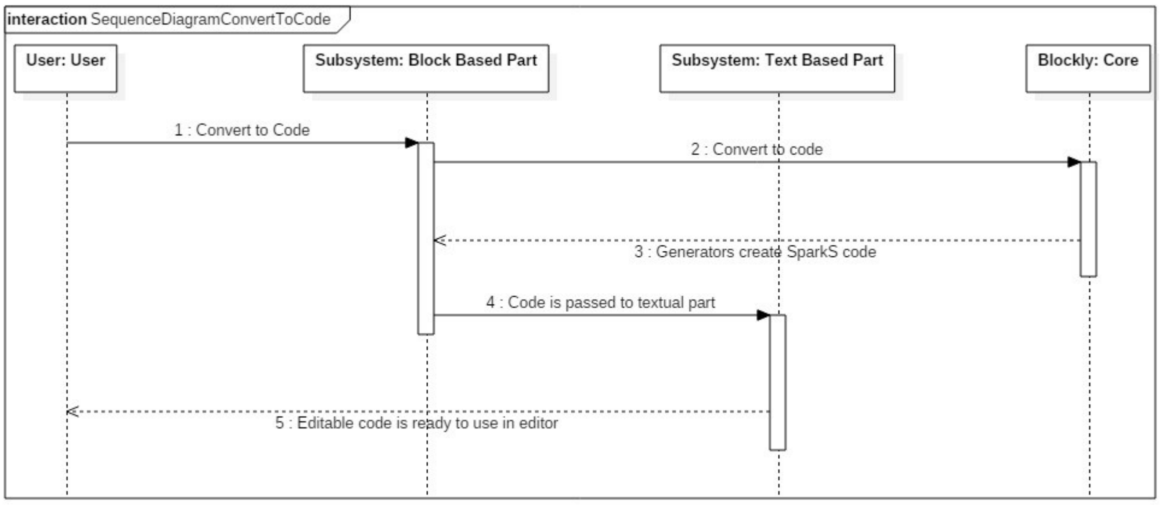

interaction SequenceDiagramConvertToCode
User: User
Subsystem: Block Based Part
Subsystem: Text Based Part
Blockly: Core
1 : Convert to Code
2 : Convert to code
3 : Generators create SparkS code
4 : Code is passed to textual part
5 : Editable code is ready to use in editor

FIGURE 4A: SEQUENCE DIAGRAM OF CONVERTING BLOCK TO CODE

USE CASE NAME	BLOCK TO CODE CONVERSION
USE CASE ID	SD3
INCLUDED USE CASES	NONE
PRIMARY ACTOR	USER
DESCRIPTION	GENERATES CODE FROM BLOCKS
PRECONDITION	EVEN IF IT GENERATES CODE FROM BLOCKS, THE WORKSPACE CAN BE EMPTY, SO TECHNIQUELY THERE'S NO PRECONDITION
TRIGGER	USER CLICKS THE BUTTON FOR THIS COVERSION
MAIN SUCCESS SCENARIO	1.USER CLICKS THE BUTTON 2.GENERATOR FILES PRODUCE A CODE FROM THE BLOCKS 3.TEXT EDITOR IS FILLED WITH GENERATED CODE
ALTERNATIVE SCENARIO	IF SOME BLOCKS HAVE EMPTY SPACES FOR PARAMETERS, THE SYSTEM WARNS THE USER TO FILL THEM, AND REFUSES TO CONVERT
POST CONDITION	TEXT EDITOR IS UPDATED

Figure 4b: Use Case Scenario of Block to Code Conversion

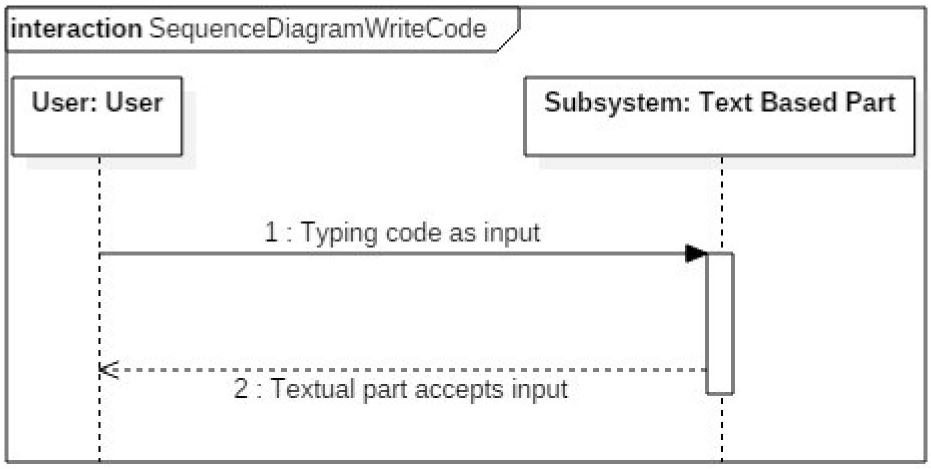
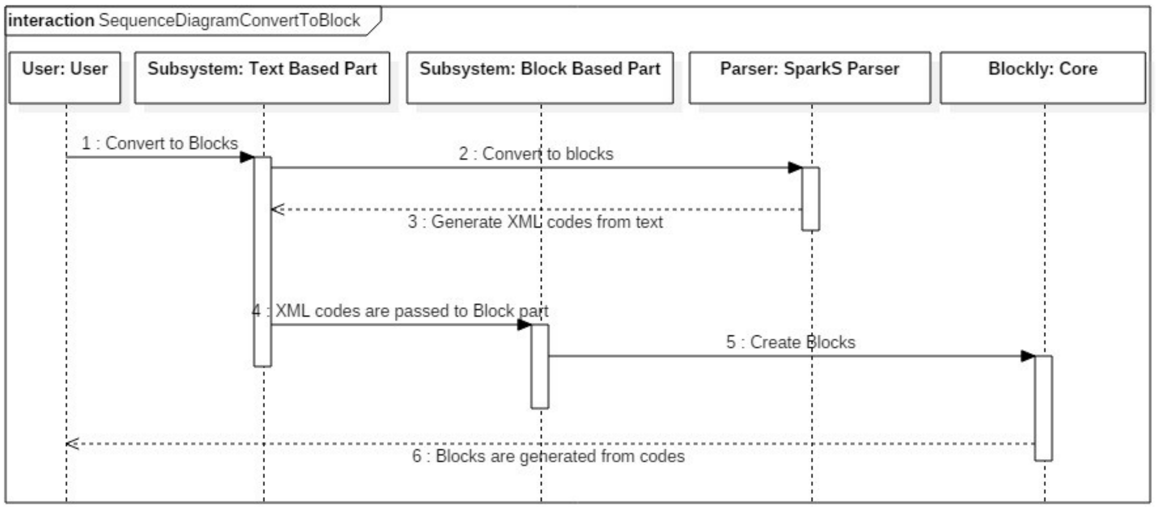

interaction SequenceDiagramWriteCode
User: User
Subsystem: Text Based Part
1 : Typing code as input
2 : Textual part accepts input

Figure 5a: Sequence Diagram of Writing Code

Use Case Name	Writing Code
Use Case ID	SD4
Included Use Cases	None
Primary Actor	User
Description	User writes the code to text editor part
PreCondition	None
Trigger	By typing letters, text editor takes the input
Main Success Scenario	1.User types the code 2.Text editor accepts the input
Alternative Scenario	There's no such case
Post Condition	Text editor is updated with the code that user types

Figure 5b: Use Case Scenario of Writing Code



interaction SequenceDiagramConvertToBlock
User: User
Subsystem: Text Based Part
Subsystem: Block Based Part
Parser: SparkS Parser
Blockly: Core
1 : Convert to Blocks
2 : Convert to blocks
3 : Generate XML codes from text
4 : XML codes are passed to Block part
5 : Create Blocks
6 : Blocks are generated from codes

FIGURE 6A: SEQUENCE DIAGRAM OF CONVERTING CODE TO BLOCKS

USE CASE NAME	CODE TO BLOCK CONVERSION
USE CASE ID	SD5
INCLUDED USE CASES	NONE
PRIMARY ACTOR	USER
DESCRIPTION	CODES ARE CONVERTED INTO BLOCKS
PRECONDITION	THE TEXT EDITOR CAN BE EMPTY, SO IT PRODUCES NO BLOCKS. WE CAN SAY THERE'S NO PRECONDITION
TRIGGER	USER CLICKS THE CONVERISON BUTTON
MAIN SUCCESS SCENARIO	1.USER CLICKS THE BUTTON 2.PARSER PARSES THE INPUT 3.XML CODE IS GENERATED FROM THIS PARSED INPUT 4.XML CODES ARE SENDT TO BLOCK BASED PART 5.BLOCKS ARE GENERATED
ALTERNATIVE SCENARIO	IF THE CODE IS UNRECOGNIZABLE BY THE PARSER, IT WILL GENERATE ERRORS INSTEAD OF BLOCKS, AND STOP BLOCK CREATION
POST CONDITION	WORKSPACE IS UPDATED WITH THE CODE WRITTEN IN TEXT EDITOR

FIGURE 6B: USE CASE SCENARIO OF CODE TO BLOCK CONVERSION

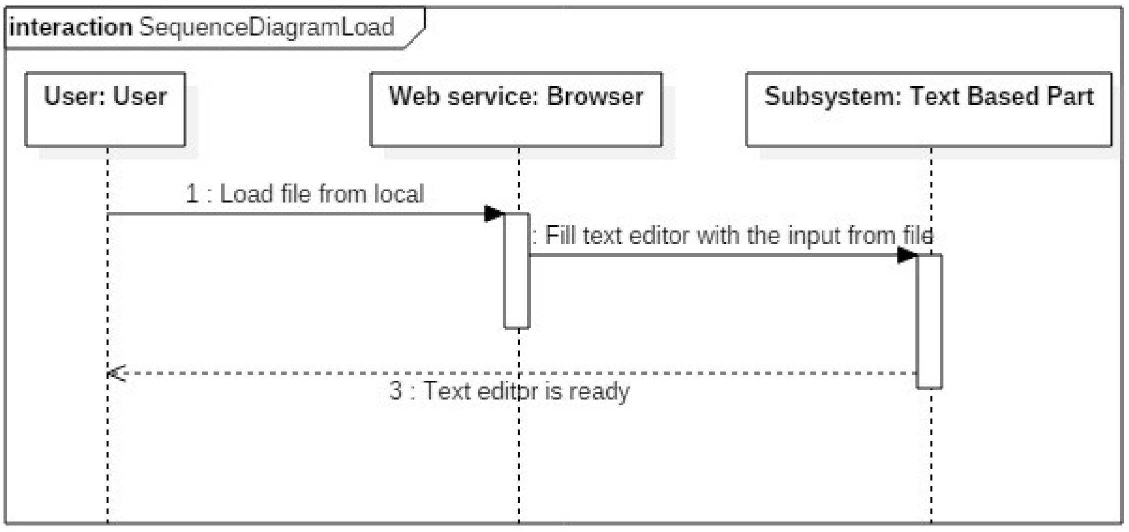

interaction SequenceDiagramLoad
User: User
Web service: Browser
Subsystem: Text Based Part
1 : Load file from local
: Fill text editor with the input from file
3 : Text editor is ready

FIGURE 7A: SEQUENCE DIAGRAM OF LOADING FILES

USE CASE NAME	LOAD FILE
USE CASE ID	SD6
INCLUDED USE CASES	NONE
PRIMARY ACTOR	USER
DESCRIPTION	A SPARKS SCRIPT FILE WILL BE LOADED
PRECONDITION	NONE
TRIGGER	USER CLICKS TO LOAD BUTTON
MAIN SUCCESS SCENARIO	1.USER CLIKCS THE LOAD BUTTON 2.CODE IS TAKEN FROM THE FILE 3.TEXT EDITOR IS FILLED WITH THIS CODE
ALTERNATIVE SCENARIO	IF FILE TYPE IS NOT SUPPORTED, THE CODE WILL NOT BE SHOWN
POST CONDITION	USER CAN START WORKING ON A CODE THAT IS WRITTEN OUTSIDE OF SPARKLY

FIGURE 7B: USE CASE SCENARIO OF LOAD FILE

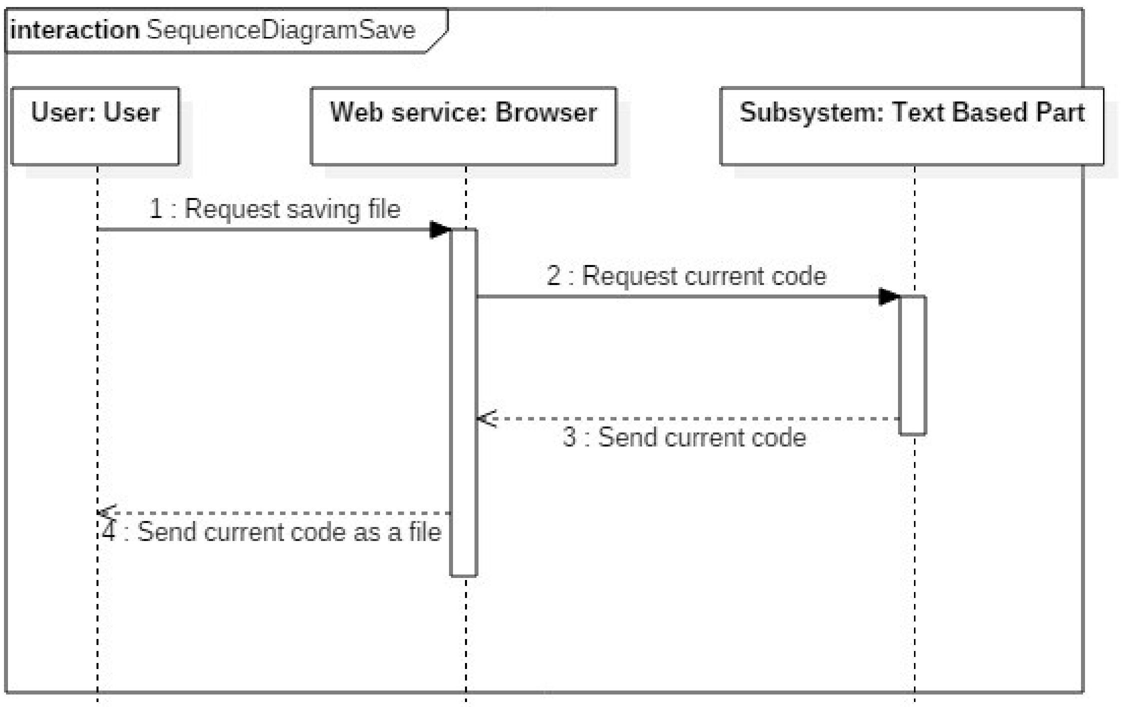


FIGURE 8A: SEQUENCE DIAGRAM OF SAVING FILES

USE CASE NAME	SAVE FILE
USE CASE ID	SD7
INCLUDED USE CASES	NONE
PRIMARY ACTOR	USER
DESCRIPTION	CODE WILL BE WRITTEN INTO A NEW FILE, AND THIS FILE WILL BE DOWNLOADED TO USER'S COMPUTER
PRECONDITION	NONE
TRIGGER	USER CLICKS TO THE SAVE BUTTON
MAIN SUCCESS SCENARIO	1.USER CLICKS ON THE SAVE BUTTON 2.CODE IS TAKEN FROM THE EDITOR 3.CODE IS WRITTEN TO A FILE 4.THE FILE IS DOWNLOADED
ALTERNATIVE SCENARIO	THERE'S NO SUCH CASE
POST CONDITION	USER WILL GET THE CODE DOWNLOADED TO HIS/HER LOCAL STORAGE

FIGURE 8B: USE CASE SCENARIO OF SAVE FILE

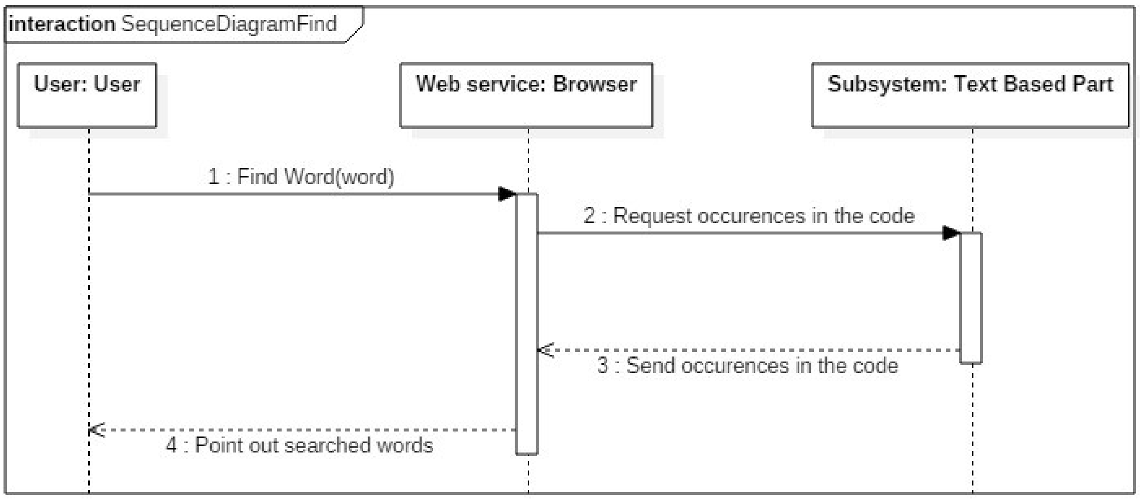

interaction SequenceDiagramFind
User: User
Web service: Browser
Subsystem: Text Based Part
1 : Find Word(word)
2 : Request occurences in the code
3 : Send occurences in the code
4 : Point out searched words

FIGURE 9A: SEQUENCE DIAGRAM OF FINDING KEYWORDS

USE CASE NAME	FIND
USE CASE ID	SD8
INCLUDED USE CASES	NONE
PRIMARY ACTOR	USER
DESCRIPTION	FINDS OCCURENCES OF TYPED KEYWORDS IN THE CODE
PRECONDITION	NONE
TRIGGER	USER CLICKS THE FIND BUTTON AND ENTERS KEYWORD
MAIN SUCCESS SCENARIO	1.KEYWORD IS ENTERED 2.THE KEYWORD IS SEARCHED IN THE CODE 3.THE KEYWORD'S OCCURENCES ARE LISTED, AND POINTED OUT IN TEXT PART
ALTERNATIVE SCENARIO	IF THE KEYWORD IS NOT FOUND, THEN THERE'S NOTHING TO BE POINTED OUT
POST CONDITION	USER CAN EASILY SEE THE OCCURENCES OF DESIRED WORDS

FIGURE 9B: USE CASE SCENARIO OF FIND

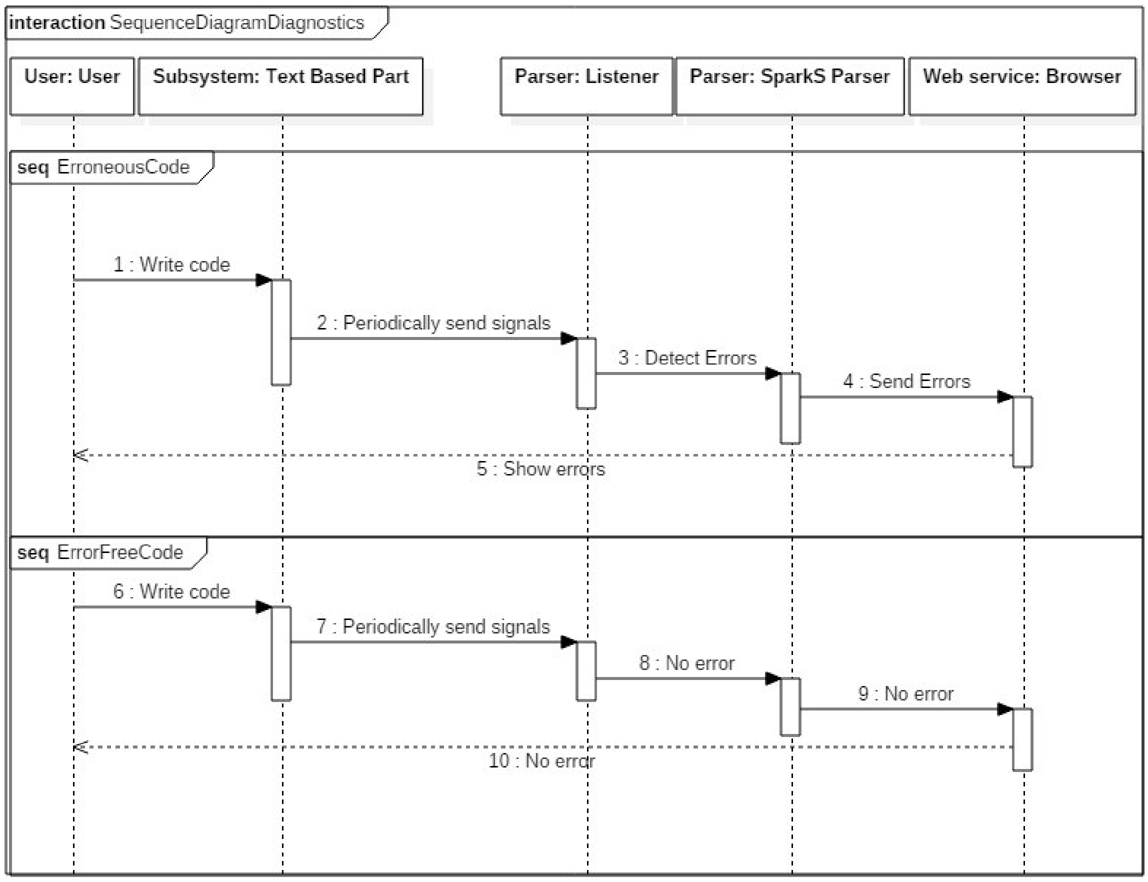


FIGURE 10A: SEQUENCE DIAGRAM OF DIAGNOSTICS REPORTING

USE CASE NAME	DIAGNOSTICS REPORTING
USE CASE ID	SD9
INCLUDED USE CASES	NONE
PRIMARY ACTOR	USER
DESCRIPTION	FINDS MISTAKES IN CODE, AND REPORTS THEM TO THE USER
PRECONDITION	NONE
TRIGGER	TRIGGERED PERIODICALLY AS USER TYPES CODE
MAIN SUCCESS SCENARIO	1. USER TYPES 2. SIGNAL IS TRIGGERED 3. CODE IS PARSED 4. ERRORS ARE FOUND BY THE PARSER 5. ERRORS LISTED IN THE EDITOR
ALTERNATIVE SCENARIO	THERE'S NO MISTAKE IN THE CODE, THEN NO ERRORS WILL BE LISTED
POST CONDITION	ERRORS WILL BE LISTED IN THE CORRESPONDING SECTION

FIGURE 10B: USE CASE SCENARIO OF DIAGNOSTICS REPORTING

Feature List

In Sparkly, we are providing the following features right now :

- SparkS language support
- Both block-to-code and code-to-block conversion
- Load/Save file
- Toggle between text part and block part
- Block and the corresponding code matching in both ways
- Text editor with
  - ✓ Syntax highlighting
  - ✓ Autoindentation
  - ✓ Autocompletion/Suggestion
  - ✓ Code folding
  - ✓ Bracket&keyword matching/closing
  - ✓ Diagnostics reporting
  - ✓ Match highlighter
  - ✓ Find/Replace
  - ✓ Several themes
- Block editor with
  - ✓ Customized blocks for SparkS
  - ✓ Erroneous block highlighting
  - ✓ Comments inside blocks
  - ✓ Block collapsing
  - ✓ Outer block detection with color change
  - ✓ Zoom in/out

Software Dependencies

Throughout the development process of Sparkly, we have used the following open-source softwares with their versions :

- Google Blockly 1.0.0
- CodeMirror 5.19.0
- ANTLR 4
- Require.js library

Moreover, to be able to run Sparkly, one has to use “Mozilla Firefox” as a web browser.