Programming Life - Test and implementation plan

Group 5/E:
Felix Akkermans
Niels Doekemeijer
Thomas van Helden
Albert ten Napel
Jan Pieter Waagmeester

March 21, 2012

Contents

1 Introduction

In this report the different testing techniques we will use for this project will be explained. Because our solution has a clear division between server and client and because these will be developed in different programming environments, we will also need different testing strategies for the client and server. In chapter 2 a prioritization of the requirements can be found using the MoSCoW system. Chapter 3 will explain how we will test the server and client, and what strategies we will use. Lastly chapter 4 will cover the risk analysis, describing the risks for the successful implementation of the system.

2 MoSCoW prioritization

In this chapter we will specify our priorities of requirements using the MoSCoW model. This model divides requirements on how viable it is to implement certain features: Must-Haves are features that the application cannot do without. These are all necessary for the program to function properly. Should-Haves group the features that are high-priority, but are not critical for the system. Could-Haves are features that would be nice to be have, should the time allow it and Wont-Haves are features that will not be implemented (in this version of the program).

2.1 Must Haves

Available gates The application must be able to present a list of available gates to the user. These gates can be used to model the circuit.

Design circuit The user must be able to design a circuit by specifying gates (using a drag-and-drop) and the relations between these gates.

Available proteins The application must be able to present the user with an overview of available proteins to assign to signals.

Protein specification The user must be able to specify which protein is used for a certain signal.

Export circuit The application must to able to save a circuit.

Import circuit The application must be able to load an exported circuit.

Input values specification The user must be able to specify the input values used for the simulation of the circuit.

Circuit simulation The application must be able to simulate a valid circuit and present the output values to the user.

2.2 Should Haves

Re-use BioBricks The application can import pre-defined circuits as extra gates. This is not a necessity, but would be a great addition to the program (and will ease building circuits). Among others, protein specification, importing and exporting will be more difficult to implement.

Circuit validation The user must to be able validate his circuit in the application. This is not a must as simulation will fail if the circuit is invalid, but modeling will be faster if the user can easily get feedback.

2.3 Could Haves

Determine proteins by specifying circuit, input and output values It is possible to let an algorithm choose the best proteins for the signals in a circuit, given user specified input and output values. This feature should be a nice extra and will be implemented if time allows it.

Local back-up If, for whatever reason, a crash occurs (the connection drops, the server stops functioning, etc.), it would be nice to provide the user with a backup of his/her work. This feature has not much to do with the main goal of this application (creating and simulating a circuit), so that is why it is a could-have feature.

Multi-client The application must be able to handle multiple clients concurrently. This is not a point of attention, as modeling can easily be done one circuit at a time. Another issue is that implementing and properly testing this feature desires significant attention, that is why we will do it if we have enough time.

2.4 Wont Haves

Determine cirtcuit and proteins by specifying input and output values It is possible to let an algorithm design a circuit based on merely given input and output signals. We deem designing such an algorithm takes up a lot of time and is impossible to do properly in our limited timespan.

Biological plausibility It is near impossible to create a program in which a user can model a biological circuit that will work in the real world as there are just too much (unpredictable) factors to take into account. With our limited knowledge of the subject, we will not try to pursue a biological plausible implementation.

3 Implementation and tests

3.1 Order of implementation of features

The concept of Scrum is to always have a working product. We will try to follow this concept. Because there is a distinction between the server and client side in our application, it should be easy for the group to work at the same time. The first steps of the building process would be to create a framework for sending/receiving messages between these two subsystems.

After that, steps can be made to gradually build up the application. This is our planning, in order of implementation:

- Server-Client communication (including definition of object formats);
- List available proteins and gates;
- Design circuit;
- Specify proteins;
- Validate circuit;
- Import/Export circuits;
- Specify input values;
- Simulate circuit;
- Re-use circuits;
- Local back-up

3.1.1 Iterations

In our process of building the application, we will have Scrum iterations of two weeks each. This means that we only decide what to implement for the coming two weeks. After these two weeks, the application should have been improved (and still working!) and we will decide again for the coming two weeks.

We will have five iterations in total before delivering the final product. This is our planning:

1. Set up a basic back-end for the client and server side. It should be able to communicate the list of proteins and available gates.
2. The user should be able to model a circuit and specify the proteins for the signals.
3. Importing, exporting and specifying input values should work. Server side must be able to validate and simulate a circuit.
4. Client side must be able to show the simulation and be able to re-use circuits as new gates.
5. Margin for finishing touches and perhaps extra features such as local back-ups.

3.2 Server Test plan

3.2.1 Unit testing

Filesystem read testing:

1. Read a .syn file that contains the a relatively simple graph and a list with enough proteins, which are all assigned to edges.
2. Read a .syn file that contains the a relatively simple graph and a list with not enough proteins, where undefined proteins are assigned to edges. Validate that an exception is thrown.
3. Read a .syn file that contains the model of item 1, with two simulation data series.
4. Read a corrupt .syn file that contains the above model, but with corruptions in each part. Validate that the reader will detect this and throw an exception.
5. Read a .syn file that contains the an extremely large model. Validate that the function call ends within a certain time.

Filesystem write testing:

1. Write a .syn file that contains the a relatively simple graph and a list with enough proteins, which are all assigned to edges.
2. Write a .syn file that contains the a relatively simple graph and a list with not enough proteins, where undefined proteins are assigned to edges. Validate that an exception is thrown.
3. Write a .syn file that contains the model of item 1, with two simulation data series.
4. Write a corrupt .syn file that contains the above model, but with corruptions in each part. Validate that the reader will detect this and throw an exception.
5. Write a .syn file that contains the an extremely large model. Validate that the function call ends within a certain time.

HTTP API:

1. Issue a listFiles() call and validate that all .syn files in the specified folder are returned, and also validate that interactions with the filesystem subsystem have been made.
2. Issue a getFile() call with an existing filename and validate that the correct call is made to the filesystem reader to read the given filename.
3. Issue a getFile() call with an non-existent filename and validate that the filesystem reader throws an exception.
4. Issue a getFile() call with an existent filename of a corrupt file, and validate that the filesystem reader throws and exception.
5. Issue a putFile() call with a provided model and filename, and validate that the file is written, and also validate that interactions with the filesystem subsystem have been made.
6. Issue a listProteins() call when a model is loaded and ensure that all the proteins are returned with their meta-data.
7. Issue a modelToSBML() call when a model is loaded and ensure that all the model is written correctly, by checking for written data and validating that is complies with the SBML schema. Also validate that interactions with filesystem subsystem have been made.
8. Issue a simulate() call with an existent filename and validate that the correct call is made to the simulation subsystem, and a call is made to validate the model first. Input values can be arbitrary.
9. Issue a simulate() call when with an existent filename, of which the model is corrupt, and ensure that a call is made to validate the model and that an exception is thrown. Input values can be arbitrary.
10. Issue a simulate() call with a non-existent filename to the simulation subsystem and ensure that an exception is thrown. Input values can be arbitrary.

JSBML Solver testing/simulator:

1. Call the simulator to solve a relatively simple model and ensure that a new thread is started and output values are returned within a certain time.
2. Call the simulator to solve a extremely large model and ensure that a new solver thread is started. If the solver crashes or hangs for longer than a specified time, ensure that an exception is thrown and the thread is terminated.

Webserver testing

1. Request a connection with the webserver with the request to send the GUI and validate that a connection is set up and the GUI page is served.
2. Ensure that after a certain time and idle connection is discarded.

3.2.2 Integration testing

On integration testing a

3.2.3 Acceptance testing

3.3 Client Test plan

Client testing can be separated from the development of the server by mocking different server-replies in simple text-files. These different test files contain replies the server might give to certain requests from te client and must be interpreted by the client code.

3.3.1 Unit testing

In order to test various small parts of the system we'll use unit tests. Defining unit tests is a good way to test for expected functionality but also ensure stability in functionality while changes to other parts of te code are made. Since we'll use the JQuery-framework in our frontend code, its unit testing framework QUnit[1] seems a logical choice.

3.3.2 Integration testing

Some integration can be tested through the use of QUnit as well, but it might be usefull to use Crawljax[2], especially for some more complex interaction tests.

The core of our integration tests will consist of behaviour of several units, covered with unit tests, functioning together.

3.3.3 Acceptance testing

User stories developed for each development run provide valuable information about acceptance tests to be executed. For each user story we'll create one or more acceptence tests, which may consist of unit and integration testing.

For some things not really measureable, like usability, we'll use additional manual tests, both performed by the members of the development team and volunteers around us.

[1] Documentation for QUnit: http://docs.jquery.com/QUnit
[2] Home of Crawljax: http://crawljax.com/

3.4 Testing Client-server integration

Integration testing is done when the individual software modules are combined and need to be tested as a whole. One approach is to use unit tests which test the whole system. The server can be run on the same machine as the client, in this way the unit tests can test these two components.

4 Risk analysis

Our program will be used by several users. These users all work in high end research teams and information should be secure. To help do this we will install some safeguards. But what risks are we looking at? Where are the potential threats? And how do we counter them. We will look at what information is visible to other users and security login. We will also have a look at what type of data transfer we will use and how safe they are. The final thing we look at is what happens when the application has an emergency shutdown of some sort and how files will be backed up.

4.1 Users and their files

Users working with our application will be working in high end research teams. Their data might be classified or, in any case, should be hidden for other users. To do this one would need a secure user account for each user. However, due to the time issues we have, we will not implement this. This could be a future function to be implemented.

4.2 Data transfer

The data transfer will happen between the server and the client. This has no major security threats. There could be some sort of injections, but this will prove to be difficult. We will check if everything saved on the server is a valid BioBrick or another datatype which we use. Any other stored data will produce error messages and will be blocked off.

4.3 Emergency shutdown

What will happen in case of an emergency shutdown. There are two things that should occur. The program should shutdown clearly, leaving no room for messing around. Secondly, the program should backup all the last changes. This will be done as a separate file, because we do not want to override the main project. Due to time issues, we cannot ensure the implementation of this.