Programming Life - Test and implementation plan

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

March 23, 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

Connection Client and server must be able to communicate. If there is no connection, the user should be notified.

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.

- The application must be able to visualize a gate using a simplified image. This image should relate to the function of the gate. For example, for the AND gate, it is logical to use the AND symbol normally used in circuit design.
- The user must be able to drag and drop gates from the list into the working area.
- The user must be able freely to move the gate around in the working area, but gates snap to grid points on the working area.
- The user must be able to draw relations between the gates in the form of wires.
- The user must be able to draw input and output wires for the circuit, to explicitly state which proteins will be used as input.

Available proteins The application must be able to present the user with an overview of available proteins to assign to signals (visualized by the wires).

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 validation The user must to be able validate his circuit in the application and get feedback over where there are conflicts.

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 circuits 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.

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 very difficult to do properly given our limited timespan.

Biological plausibility It is very hard 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.The following list is our planning, in order of implementation. For each point we specify whether it will be work on server side (S), on the client side (C) or both (SC). Also specified is how much time we think is needed to implement this feature. S is a small task (around a day of work), M a task of medium size (a week) and B a big task (more than a week). These times are just indications and can be used for comparison.

- Server-Client communication (including definition of object formats); SC M
- List available proteins and gates; SC S
- Design circuit:
  - Visualize gates using a simplified images; C S
  - Drag-and-drop gates into the working area; C S
  - Moves gates aroun in the working area; C M
  - Draw wires between gates; C M
  - Draw input and output wires; C S
- Specify proteins; C M
- Validate circuit; SC S
- Import/Export circuits; SC S
- Specify input values; C M
- Simulate circuit SC M

Should there be enough time left, we will try to implement the following features in the given order:

- Re-use circuits; SC M
- Local back-up; C M
- Determine proteins using given input and output values; SC B
- Multi-client SC B

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 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. After this validation testing needs to be done, the system needs to be validated if it conforms to the requirements. This will be done by letting a third party test the software. After the testing the participants will be interviewed.

4 Risk analysis

In this chapter we will discuss what we see as possible risks to our project. We will look at the difficulty of serveral implementation steps. After that, testing difficulties will be discussed. Third point of attention is external parties. Who do we need beside our own team to finish this project successfully and how does that invoke issues? Furthermore we will examine what we will do if the project stumbles upon unforeseen problems. Finally we will have a look at every person's schedule to estimate their work pressure.

4.1 Difficulties while implementing

During the implementation of the program we might encounter some difficulties. This is only normal, but still we need to analyze what could happen and how we will respond. Our development process will be test driven. This means we will first write tests for what our program should do and afterwards work out how our application will succeed in passing this test.

One of the problems we might be facing lies in the general structure of our program. We have a client and a server. They need to communicate. Communication is a well-known area for problems. To counter this problem we do intensive testing using mock-ups for both the server and the client to test each part first. After both parts have been tested, we will have integration tests to see how both sides work together in a real environment.

Another problem during implementation Is the client itself, or rather the GUI presented in the client. The GUI has to work with dragging and dropping gates into place. We do have an idea how this should be implemented, but during the designing process we already stumbled upon some scenario's where gates might be hard to connect. We want the GUI to remain clear and not filled with gates and wires connecting them, losing sight of what belongs where.

The third problem we might face is of course the server. Problems in this sector will probably occur during the saving of data. The server is responsible for storing data and retrieving it if requested. We realize that this might go wrong, and errors might occur if the data is not stored properly or is edited by some other program.

Because we are working with a five man team, there is always a chance that some people might name things differently. One person might call a function getApplicationName, while another might call it getAppName. Miscommunications like these can cause a lot of redundant code and errors, especially if different persons keep changing classes. It is hard to keep an overview of which function has been used where, so removing a function might damage a totally different class. This problem can be easily countered with a good design and a proper ontology.

4.2 Difficulties with testing

As said before our development is test driven. This means that testing is extremely important and problems during testing should be countered early on in the development process. So what kind of problems do we expect during testing.

Creating tests is rather difficult. You have your set of requirements, but you also need to test some of the boundaries of your application. You also need to think about how intensively you are going to test every aspect of your application. This is stated in the previous chapters of this document, but still it remains a challenge not to be underestimated.

If a test is constructed properly and it fails, the application has a bug somewhere. But how do we find this bug and how do we fix it. So bug-tracking is also very important. Even though Java has some really good tools to help find bugs, it remains time consuming and we need to take this into account.

4.3 External parties

Our project is not heavily dependent upon external parties. However, we do occasionally have to wait for feedback from instructors. We are sometimes fully reliant upon information given by instructors. However we do not expect a lot of difficulties with this.

4.4 Unforeseen problems

What happens if something unexpected happens? If we stumble upon a problem we did not foresee? We might have to add additional features to our application or we find a bug which we cannot easily fix. If problems like these occur we need to use our spare time to fix these problems. We have a strict planning of when everything is due, and in planning this we kept some spare time to deal with unexpected issues.

4.5 Work stress of every team member

The work stress of most members of our team is pretty high. Four out of five follow a lot of courses, which leads to more practical work than just this project. We need to take into account deadlines of other projects. The fifth member, during the 3rd semester, only does this course. However, he is a board member of a student association which takes a lot of time as well. We do take this into account but not as much, simply because it is not a constant pressure.