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 **2**

2 Clarification of some definitions **2**

3 MoSCoW prioritization **4**
3.1 Must-Haves 4
3.2 Should-Haves 4
3.3 Could-Haves 4
3.4 Won't-Haves 5

4 Implementation and tests **5**
4.1 Order of implementation of features . . 5
4.1.1 Iterations 6
4.1.2 Milestones 6
4.2 Server Test plan 7
4.2.1 Unit testing 7
4.2.2 Integration testing 8
4.2.3 Acceptance testing 8
4.3 Client Test plan 9
4.3.1 Unit testing 9
4.3.2 Integration testing 9
4.3.3 Acceptance testing 9
4.4 System integration testing 10

5 Risk analysis **10**
5.1 Difficulties while implementing 10
5.2 Difficulties with testing 10
5.3 External parties 11
5.4 Unforeseen problems 11
5.5 Work stress of every team member . . . 11

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 Clarification of some definitions

In previous reports we were not very careful in the choice of our words. In this section, we will clarify some definitions.

BioBrick

The somewhat broad definition we presented before:

> Isolated and documented cell function to be reused in future projects. For example, the production of a light emitting protein when some other protein is available.

is not the way we used it in our documentation. Technically, any sequence of genes which is somewhat isolated and documented could be called a BioBrick, even the whole *circuit* in the modeller to be developed, however, the handout[1] uses a more narrow definition:

> ...
> , it is reasonable to choose genes (coding sequences) and promoters as the smallest building blocks (BioBricks) for this project.
> ...

From now on, when we talk about BioBricks we'll use this definition.

Transcription factor

Before, we used different terms to refer to signals in the circuit, depending on the context we were talking about. The signals in the circuit can be modelled by wires and are represented by *transcription factors* (TF) in the cell. These transcription factors are of course proteins, so 'list of proteins' is equivalent to 'list of transcription factors' in our text.

Gene coding sequence

Gene coding sequences (CDS) are parts of the gene that encode for a certain *transcription factor*. For each *gate* the CDS can be freely chosen.

Promotor

Promotors are parts of the gene which facilitate the polymerase to bind to the DNA, starting the transcription process for the CDS region after the promotor region. These promotors may need certain *transcription factors* to be present before the polymerase can bind.

Gate

A gate is defined by an promotor and a *CDS*. The promotor defines the input(s) of the gate, the *CDS* defines the output. In the case of an NOT-promotor, one transcription factor disables the binding of the polymerase. In the case of an AND-promotor, two transcription factors are required before the polymerase can bind and start the transcription reaction.

Circuit

The circuit is the collection of *gates* connected by *signals*.

[1] PDF handout: BioBricks construction for Context Project 2011/2012

SBML

Being a little unfamiliar with the format, we were not careful enough when we formulated the requirements about them. SBML is a format targeted at the biological part of the project, we need to save more information than possible in the standard SBML. We choose to persist the circuits in a different format: `.syn`. SBML will be used as an export-format and as the interface to the simulator.

Simulation

In order to simulate the behaviour of the circuit, we'll use a SBML solver as mentioned before. The SOSLib[2], which is a library to simulate SBML files written in C. It has a simple command-line interface and produces simple, easy to parse text-files containing the simulation results.

[2] Project home: http://www.tbi.univie.ac.at/~raim/odeSolver/, GitHub page: https://github.com/raim/SBML_odeSolver

3 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 have, should the time allow it, Won't-Haves are features that will not be implemented (in this version of the program).

3.1 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 distinctive AND symbol[3] 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 will snap to grid points on the working area.
- The user must be able to draw connections 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.

3.2 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.

3.3 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.

[3] IEEE Standard 91-1984

3.4 Won't-Haves

Determine circuit 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.

4 Implementation and tests

4.1 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. A small task (s) should take around a day of work, a task of medium size (m) a week and big task (b) 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 around 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

4.1.1 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 each iteration, the application should have been improved, while still working and the next iteration will be started by deciding what to implement in the two coming weeks.

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

1. *(26-03 until 06-04-2012)* 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.
   - Server: Tomcat server that handles HTTP requests;[1]
   - Server: Basic MVC framework, so that future components can easily be integrated;[1]
   - Server: Serve available proteins and gates;
   - Client: Create an *index.html* with a basic GUI;[1]
   - Client: Show the connection state;
   - Client: Show available gates
2. *(23-04 until 04-05-2012)* The user should be able to design a circuit and specify the proteins for the signals.
   - Client: Drag-and-drop gates in the working area and move them around;[2]
   - Client: Draw wires between gates and draw input/output wires;
   - Client: Specify proteins for wires
3. *(07-05 until 18-05-2012)* Importing, exporting and specifying input values should work. Server side must be able to validate and simulate a circuit.
   - Server/Client: Save circuit;[2]
   - Server/Client: Load circuits;[2]
   - Server/Client: Validate circuit;[2]
   - Server: Simulate circuit;
   - Client: Specify input values
4. *(21-05 until 01-06-2012)* Client side must be able to show the simulation and be able to re-use circuits as new gates.
   - Server: Serve output values for simulation;[2]
   - Client: Display simulation;
   - Server/Client: Re-use circuits
5. *(04-06 until 15-06-2012)* Margin for finishing touches and perhaps extra features such as local back-ups.

4.1.2 Milestones

Milestones in our project, including their planned deadlines, are:

- **Version 0.1**: A working client and server base which can report the state of the connection *(27-03-2012)*;
- **Version 0.5**: The user can design a basic circuit (gates and wires with specified proteins) *(03-05-2012)*;
- **Version 1**: The application can load, save and validate circuits *(10-05-2012)*;
- **Version 2**: The application can simulate a circuit *(17-05-2012)*;
- **Version 3**: Circuits can be reused when designing *(01-06-2012)*;
- **Version 4**: Fully tested final product *(18-06-2012)*

[1] To be finished as soon as possible in this iteration
[2] To be finished in the first week of this iteration

4.2 Server Test plan

4.2.1 Unit testing

Unit testing is performed per function used one or a series of functional testing and interaction tests. Because the server is written using Java, we will use the Java testing frameworks JUnit[4] for functional testing, and for interaction testing we will use Mockito [5].

Filesystem read testing:

1. Issue a getFile() call on a .syn file that contains the a relatively simple graph and a list with enough proteins, which are all assigned to edges.
2. Issue a getFile() call on 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. Issue a getFile() call on a .syn file that contains the model of item 1, with two simulation data series.
4. Issue a getFile() call on 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. Issue a getFile() call on a .syn file that contains the an extremely large model. Validate that the function call ends within a certain time.
6. Issue a listFiles() call and validate that all .syn files in the specified folder are returned.
7. Issue a listProteins() call when a valid document listing the TF's and CDS's is present and validate that all the listed elements are read. Boundary test by varying the number of elements from 0 to 3, and also one with in the order of 1000 elements.
8. Issue a listProteins() call when an invalid document listing TF's and CDS's is present and validate that an exception is thrown.
9. Issue a listProteins() call when no document listing TF's and CDS's is present and validate that an exception is thrown.

Filesystem write testing:

1. Issue a putFile() call with a .syn filename, and a relatively simple graph JSON.
2. Issue a putFile() call with a .syn filename, and a relatively simple graph JSON, with two simulation data series.
3. Issue a putFile() call with a .syn filename and corrupt graph JSON and validate that an exception is thrown.
4. 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.

HTTP API:

1. Issue a listFiles() call and validate that interactions with the filesystem subsystem have been made. Mock the filesystem and to make it return a certain set of files, and validate that the correct JSON is generated.
2. Issue a getFile(filename) call with an existent filename and validate that the correct call is made to the filesystem reader to read the given filename.
3. Issue a getFile(filename) call with an non-existent filename and validate that the filesystem reader throws an exception.
4. Issue a getFile(filename) call with an existent filename of a corrupt file, and validate that the filesystem reader throws and exception.
5. Issue a putFile(model, filename) call with a provided model and filename, and validate that the correct call to the filesystem has been made. Mock the filesystem to make it return true and false with all possible exceptions for separate test. Validate that the correct JSON is generated.

[4] Testing framework used to automate testing in Java. Home page of JUnit: www.junit.org/

[5] Testing framework for JUnit, used to mock classes and let them performed predefined interactions, and test on interactions made between objects. Home page of Mockito for JUnit: code.google.com/p/mockito/

6. Issue a listProteins() call. Mock the filesystem reader to provide predefined sets of return values, and validate that the correct JSON is generated. Boundary test by varying the set of return values from 0 to 3, and one with in the order of 1000.
7. Issue a modelToSBML() call with a model and ensure the correct call is made to the filesystem writer.
8. Issue several validate(model) calls and ensure that the correct call has been made to the validation subsystem. Mock the validation subsystem to return true and false with all possible exceptions and ensure that the right JSON is generated.
9. Issue several simulate(model, inputValues) calls and ensure that the the simulation subsystem is called after the validate() function is called. Mock the validation subsystem as described in test 8 and mock the simulation subsystem. Test that when the validation subsystem returns true that the simulation subsystem is not called. When the validation subsystem returns true, the simulation subsystem should be called and should return an output values object. Mock the simulator to return output values data series of length 0 to 3. Validate that the correct JSON is generated.

JSBML Solver testing/simulator testing:

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. Issue a simulate() call with 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.

Validator testing:

1. Issue a validate() call with a valid model and ensure that true is returned.
2. Issue a validate() call with a set of invalid models, in which there is a invalid model for each type of exception that can be thrown by the validator. Ensure that for each model the correct exception is thrown.

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.

Controller testing

1. Mock the filesystem reader to make take forever to complete a function call. Ensure that after a certain time, the filesystem reader thread is terminated.
2. Mock the filesystem writer to make take forever to complete a function call. Ensure that after a certain time, the filesystem writer thread is terminated.
3. Mock the simulator to make take forever to complete a function call. Ensure that after a certain time, the simulator thread is terminated.
4. Mock the validator to make take forever to complete a function call. Ensure that after a certain time, the validator thread is terminated.
5. Mock the webserver to make take forever to complete a function call. Ensure that after a certain time, the webserver thread is terminated and restarted.

4.2.2 Integration testing

Testing of the integration with the client will be done by testing the HTTP API, and testing of the integration of the the server subsystems will be covered by the unit tests.

4.2.3 Acceptance testing

User stories developed for each development run provide valuable information about how much the server side weighs in the result. For each user story we'll create one or more acceptance tests, and for those that also include server calls, most can be valued by the responsiveness and stability of the server. For some tests this is already defined as a certain time within which the server has to finish the job.

4.3 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. They will be served as a static file from a webserver, so the only part we remove from the 'normal equation' is the server *deciding* what to reply.

Making these mockups costs time, but we think de decoupling of the development and testing of the client and the server is a valuable effort.

4.3.1 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[6] seems a logical choice. It can be used just like JUnit to define small unit tests before implementation of certain functionality during Test Driven Development and to prevent regression of implemented features. Some examples of unit tests include:

1. The circuit and the smaller parts of it will exist in JavaScript as objects. They'll have `toJSON()`-methods which can be tested.
2. The circuit parts parsing from small JSON-snippets to JavaScript-objects can be tested.

4.3.2 Integration testing

Some integration can be tested through the use of QUnit as well, but it might be useful to use Crawljax[7], especially for some more complex interaction tests. Crawljax will find all click able elements on our client page and try them out in different orders. This way we can see what orders fail and need to be fixed.

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

1. Parsing the (mocked) JSON-input from the server to JavaScript objects and than back to JSON. Input and output should be equal.
2. Loading a file from the server by using Crawljax to push buttons and checking the resulting JavaScript objects.

4.3.3 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 acceptance tests, which may consist of unit and integration testing.

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

[6] QUnit: a powerful, easy-to-use, JavaScript test suite. http://docs.jquery.com/QUnit
[7] Home of Crawljax: http://crawljax.com/

4.4 System integration testing

The server and client will be tested as described in the preceding chapters. Once those modules are tested the client and server need to be integrated and tested. To test these different methods will be used:

- Unit tests will be made that will be ran with the server and client both running.
- Crawljax will again be used on the client, but this time with the server running.

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. The third party in our case will be fellow students and volunteers. After the testing the participants will be interviewed to see what they thought of the system. Was the system intuitive? Did the system feel safe? did the system do what you wanted? etc.

5 Risk analysis

In this chapter we will discuss what we see as possible risks to our project. We will look at the difficulty of several 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 persons schedule to estimate their work pressure.

5.1 Difficulties while implementing

During the implementation of the program we might encounter some difficulties. This is only normal, but still we need to analyse 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 scenarios 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. Miscommunication like this 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.

5.2 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.

5.3 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.

5.4 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.

5.5 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.