Programming Life - Architectural Design

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

March 14, 2012

Contents

1 Introduction

1.1 Purpose of the System

The purpose of our system is to create a modeling environment for biology experts and people who work in the field of genetics. Within this modeling environment, you can easily model BioBricks. BioBricks are biological gates, which are created by gene manipulation. Our system should be able to create BioBricks with basic components, like AND and NOT gates. The system should also be able to import and export BioBricks, making it possible to create larger constructions with earlier designed BioBricks.

1.2 Design Goals

Our design goal for this project is to successfully implement the system mentioned above. We aim to make the system easy to use for people without too much knowledge of computer science.

1.3 Definitions, acronyms and abbreviations

BioBrick is the key word in this project. A BioBrick is a logical gate representation of a manipulated cell. The idea behind this is: Genes can be manipulated to work as gates. By doing this we can create logical structures using cells. A BioBrick is a model for this. The term Bio comes from Biology. The term Brick comes from the idea that these BioBricks should be building blocks for a larger system. With Bricks you can build bigger structures.

Client-side is the part of the program which takes place on the computer of the client/user. The opposite is server-side. For our program, the client-side is a GUI implemented in JavaScript.

Server-side is the part of the program which takes place on the server. In our case it is implemented in Java. This part will do the heavier computations, so the client-side only has to display the results.

GUI is a Graphical User Interface. This is the part of the program which you actually see as a user. It is the screen which shows you the program and allows you to do things.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate.

AJAX (Asynchronous JavaScript and XML) is a format in which data can be exchanged between client and server side. This can be done asynchronously, meaning your main program does not have to wait for the results of the data transaction before it can continue. We will use this as a part of our communication between our JavaScript client-side and our Java server-side.

SBML is a Markup Language specially designed for Synthetic Biology. We will use this to represent BioBricks. We can read and write from and to this format and easily save BioBricks as SBML files.

MVC stands for Model View Controller. It is an architectural structure which separates the display(View) of the program from the input/output handling(Controller) and the data representing part(Model). By doing this your application remains structured, can be tested in separate parts and is easier to maintain or change.

1.4 References

http://www.json.org/

http://www.sbml.org/

1.5 Overview

So this chapter was all about what we wanted to achieve globally. We want to create a modeling environment for biologists and genetics experts to model BioBricks. These BioBricks represent genetically modified cells or genes which act as logical gates. The Definitions section explains some of the major concepts. The full architectural design will be explained in detail how we intend to implement everything.

2 Proposed software architecture

2.1 Overview

In this chapter we will discuss the exact architectural design of our application. We will go into detail about how is system is build up. There will be a short explanation of every sub-system. This will all be done in chapter

3.2. There is a mapping of how different parts of our application interact with each other in chapter 3.3. In 3.4 there will be an explanation of how we manage data. Where and how will we save our BioBricks? Which format will we use? These are all questions we will answer. In chapter 3.5 we will tell something about how the application handles global recourses. There we will also explain why we chose not to have a complex access structure. Concurrency, how communication is handled in our system, will be discussed in chapter 3.6. Finally we will discuss the boundaries of our system. Where does our system end and what happens in case of errors or crashes?

2.2 Subsystem Decomposition

(which sub-systems and dependencies are there between the sub-systems?)

2.2.1 Interface (API) of each sub-system

We propose the following API for the server-side of our application. Every model is represented by an object which stores the needed information. It is basically a directed graph with building blocks as nodes.

listFiles(): list of filenames - Returns a list of available models.
If there are no available files, then it returns an empty list.

getFile(fileName): model - Returns the BioBrick model stored in fileName.
If the file does not exist or if the file is corrupt, it will give an error.

putFile(fileName, model) - Stores model in a file called fileName.
If something goes wrong while saving, it will give an error.

listProteins(): list of proteins - Returns a list of proteins, including their meta-data.
With absence of information, it will give an error.

listModels(): list of models - Returns a list with a model for all available files.
Returns an empty list when there are no available files.

modelToSBML(model, fileName) - Exports model to an SBML file and saves it to a file called fileName.
Returns an error if model is not a valid model.
Returns an error if saving failed.

simulate(fileName, inputValues): outputValues - Simulates the SMBL model saved in fileName with given input values and returns the matching outputvalues.
Returns an error if the simulation of the file failed.

validate(model): boolean - Validates the model and returns true if model has everything defined to simulate it, and returns false otherwise.

For the client-side of our application, we will not follow a predefined API. As the client will only implement the view part of the model, view controller (MVC) system, it will only reuse the API of the server and not implement an API of its own. Messages between server and client will be passed using JSON objects.

Figure 1: Examples of JSON responses

Generic response:

```
{
  "success": boolean,
  "message": string,
  "data": object
}
```

Response of getFiles():

```
{
  "success": true,
  "message": "",
  "data": [
    "example1.syn",
    "d-flipflop.syn",
    "xor-gate.syn"
  ]
}
```

Response for getFile(fileName) for a non-existing file:

```
{
  "success": false,
  "message": "Requested file not found.",
  "data": {}
}
```

2.3 Hardware/Software Mapping

Our architecture is composed of two major elements, the client implemented as a web application and the server, implemented in Java. The browser has to be a recent version of one of the major browser[1], the supporting system is not relevant. The server can be run on the same machine, which is likely to be the same machine during the development stage.

[1] We'll test in Google Chrome and Firefox

2.4 Persistent Data Management

The application will persistently store data serverside on the disk. A database seems overkill for the rudimentary data storage that is required, and the already existent and mature text formats are good candidates to save our data in. The application will store data using the following files:

1. **.xml (SBML)**
   A `text/xml` file in the Systems Biology Markup Language schema that describes the abstract model of the biological system. Within the domain of our application it will primarily be used to describe the biological processes of a modeled circuit/BioBrick.
2. **.sym (JSON)**
   A `text/json` file in our made up extension that embodies the entire modeled circuit/BioBrick in JavaScript Object Notation. The object structure has to comply with that of the application to be read.

2.5 Global Resource Handling and Access Control

The application will be controlled through a web interface. As is dictated in the requirements, the application will only work will a single user at a single point in time, and there is no access control implemented. A browser will connect to the server and without authorization will be able to access the functionality. The decision on this was made because the focus should be on implementing the required functionality. We agreed that security and multi-user operation would be a nice-to-have, but to ambitious with the current goals in mind.

This means that in theory anyone who has access to the network where the server runs could operate the application, so users of the application should be aware that they do not work on and/or save sensitive information.

2.6 Concurrency

As is described in the previous section, our system will not support multi-user operation, so operation is only guaranteed for one client-server connection at the same time. This reduces concurrency issues, but leaves some scenarios to be thought about. Because user interaction is a dominant part of our application, concurrency problems may not hinder the workflow or be very problematic. This implies multi-threading, and the following subsystems will have to run on a separate thread with concurrency handling:

1. **Main controller**
   The main controller will have to be able to handle calls from multiple subsystems at the same time. For example the user must still be able to interact with the server when the server is reading/writing a lengthy XML file.
2. **HTTP request handler**
   See next item.
3. **Web server**
   These two subsystem must be able to terminate when hung in the processing of a HTTP request or serving HTTP to a client. Therefore it must be isolated in a separate thread. Also if we decide to implement multi-user support this would be a lot easier to implement as we could easily start multiple of these threads.
4. **XML parser/writer**
   This subsystem must also be able to terminate when hung without crashing the rest of the server.
5. **Simulation math**
   This subsystem must also be able to operate independently of the rest of the server systems so in case lengthy simulations are performed, the server can still interact with the client and the simulation can be aborted.
6. **GUI**
   This is essentially the browser, and in this sense should also be considered a separate thread. This has the benefits as described in the system decomposition. Because we only support recent versions of the major browsers concurrency problems with the GUI are already covered.

2.7 Boundary Conditions

(how is the system started and stopped, what happens in case of a system crash)