Final Report

Dynamic Visualization of State Transition Systems

Alex Aalbertsberg (s1008129)

February 6, 2015

Contents

1 Introduction

In the current day and age, model-based software development is a technique that is often used to test the requirements and functionalities of a software system before physically building the system. These models are often not very readable to the tester and will not provide a clear overview of object states. The purpose behind this project is to create a way for users to have a visual overview of the software system's state while running simulation. This will provide a clear picture of the way in which the system operates.

This document will describe a software system design for an application that can dynamically receive information from applications that generate state diagrams about transitions that take place during a simulation. The application will take the information about these transitions and visually represent them, so as to create a more clear full picture of the situation for the end user.

The system has been set up in a modular way, so that expansion by adding additional functionality is not a difficult task. How this modular setup has been established will be described further in the report.

2 Requirements

The first stage of the project was to establish proper boundaries for the project. This section lists the requirements that were set for the final product. These requirements were established over several review sessions.

The final product of this project should:

1. Be able to create an initial visual representation of a state diagram from the tool that generated said diagram.
2. Be able to receive information about state transition steps from any state diagram tool.
3. Be able to interpret the received information and translate the abstract syntax that is received to a graphical representation which is also known as the concrete syntax.
4. Support a preset schema of commands and respective parameters that allows client applications to perform GUI operations according to what happens in the state transitions.
5. Update the visual representation (concrete syntax) according to the transition information that has been received.

3 Technology

3.1 Design Choices

This section will describe the technology that has been used to create the final product. The decisions to utilize the technology listed below were made during the early stages of the project.

First, we needed to decide what programming language to use. The programming languages that I was most familiar with were C# and Java, so we decided to focus mainly on those languages. Below you will find a table with characteristics for each programming languages pertaining to the requirements of the project.

Criteria	Java	C#
GUI Support	Yes, in Java 8 (JavaFX)	Yes
Networking Support	Yes	Yes
Multi-OS	Yes	Limited*

Table 1: Comparison of Programming Languages

Based on the above criteria, the decision was made to write the application in Java 8 and uses the native library JavaFX for user interfacing. Java has been chosen in order to maintain as much scalability as possible.

Criteria	JSON	CSV	XML
Overhead	Small	Small	Large
Language Support	Open-source	None	In-built
Parseability	Easy	Medium	Easy

Table 2: Comparison of Transport Protocols

Based on the above criteria, the decision was made to let communication take place by sending JSON objects over an established TCP connection. XML's large overhead and no support existing for parsing CSV data left JSON to be the best option. The application therefore adopts the client-server paradigm as its communication protocol. This means that client applications (i.e. the application simulating the state diagram) will connect to the server application and send JSON objects containing information about the steps that have been taken, so that the server application may perform actions that correspond with the received information. You will find a description of currently supported commands and their parameters below. The library that

has been used for JSON is the one from www.json.org. Their source code for supporting JSON is freely available on their website.

3.2 Application Design Specifics

The application consists of several classes, which have been listed below. This design attempts to maintain a certain level of abstraction, so that expanding upon existing functionality does not require a lot of work.

1. The Main class. This class initializes the GUI and TCP Server, so that commands may be interpreted and updated on the user interface.

2. The CommandProcessor class. This class deciphers the JSON data sent to the TCP server, interprets and then executes the specified command.

3. The AbsCommand (abstract) class. This class is a framework for the implementation of any command. It only contains a constructor that requires a JSONObject, and a performCommand method that will execute whatever the JSON data tells it to do.

4. All classes that specify a certain command derive from the AbsCommand class.

5. The SocketListener class. This class listens for incoming data on a certain port and sends the received information through to the CommandProcessor.

6. The JSONObject class. This class encapsulates an associative set of objects that can easily be sent over a TCP connection.

The client application will have to connect to the server application's IP address and port number on which it is listening. Subsequently, the client will be able to send JSONObjects over this established connection, which the server application will parse. The parsed parameters should then lead to a desired result on the GUI.

3.3 Commands and Parameters

A JSONObject should contain the command key with a valid command as its value. The commands and corresponding parameters that are currently supported by the application are listed below.

1. init. This command allows the user to set initial GUI/application values.

   (a) GUI width. ("stageWidth", double)

(b) GUI height. ("stageHeight", double)

```
JSONObject obj = new JSONObject();
obj.set(``command'', ``init'');
obj.set(``stageWidth'', 500);
obj.set(``stageHeight'', 500);
```

2. set. This command allows the user to create elements on the GUI.

   (a) Identifier. ("id", String)
   (b) X-axis position. ("posx", int)
   (c) Y-axis position. ("posy", int)
   (d) Color. ("colorCode", String)
   (e) Shape. ("shape", String (Supported values: "circle" or "rectangle"))
   (f) Image. ("image", Base64 String)

```
JSONObject obj = new JSONObject();
obj.set(``command'',``set'');
obj.set(``id'',``wolf'');
obj.set(``posx'', 100);
obj.set(``posy'', 100);
obj.set(``image'',``wolf.jpg'');
```

3. move. This command allows the user to move existing GUI elements.

   (a) Identifier of the element to move. ("id", String)
   (b) New X-axis position. ("posx", int)
   (c) New Y-axis position. ("posy", int)

```
JSONObject obj = new JSONObject();
obj.set(``command'', ``move'');
obj.set(``id'', ``wolf'');
obj.set(``posx'', 200);
obj.set(``posy'', 200);
```

4. remove. This command allows the user to remove an existing GUI element.

   (a) Identifier of the element to remove. If the specified identifier does not exist, this command does not have an effect. ("id", String)

```
JSONObject obj = new JSONObject();
obj.set(``command'', ``remove'');
obj.set(``id'', ``goat'');
```

4 Test results

For the sake of properly testing the final product, a testing method has been defined. We will check the functionalities of the main program with a simulation of the wolf-goat-cabbage problem, as described below.

4.1 Wolf-Goat-Cabbage Problem

This problem deals with the following: A man is standing on the bank of a river with a wolf, a goat and a cabbage. The man wants to cross the river with all three of them. However, his boat will only allow him to take one of the three with him at a time. The other two will have to wait. When the man is not there, the wolf is capable of eating the goat, and the goat is capable of eating the cabbage.

For this problem, both a success and a failure scenario have been created. These scenarios test the functionality of each individual command and operate properly. Of course, the success scenario first places the goat on the other bank, as the wolf and the cabbage share no interaction that would hinder progress. The man could then take either the wolf or the cabbage to the opposite bank (it does not matter which one) and takes the goat back to the first bank. Then, the man will take the wolf or the cabbage (whichever he did not take with him previously). Now, the goat will be the only one left on the first bank, allowing a simple cross to complete the problem.

The failure scenario moves the cabbage over to the other bank first, causing the wolf to "eat" the goat. The goat is then removed from the GUI.

5 Conclusions

In this section, we will check our final product by the standards of the requirements that were set at the start of the project:

Requirement 1: Be able to create an initial visual representation of a state diagram from the tool that generated said diagram.
Result: The init and set commands provide functionality for an external application to set-up the initial GUI and instantiate objects that require a visual representation.

Requirement 2: Be able to receive information about state transition steps from any state diagram tool.
Result: The application listens for a TCP connection on a certain port. Once a client application connects, that application may then send JSON objects over the connection.

Requirement 3: Be able to interpret the received information and translate the abstract syntax that is received to a graphical representation which is also known as the concrete syntax.
Result: The application reconstructs the JSON object it receives over the TCP connection and then translates the information contained to operations on the graphical interface.

Requirement 4: Support a preset schema of commands and respective parameters that allows client applications to perform GUI operations according to what happens in the state transitions.
Result: A set of commands and respective parameters has been defined, as well as the commands' respective operations on the graphical user interface.

Requirement 5: Update the visual representation (concrete syntax) according to the transition information that has been received.
Result: Each command has a corresponding effect on the state of the graphical user interface.

All requirements set at the start of the project have been met.

6 Related work

van Ham, F., van de Wetering, H. & van Wijk, J.J. (n.d.). *Visualization of State Transition Graphs.*
Retrieved from http://www.win.tue.nl/w̃stahw/papers/infovis2001_2.pdf

Diethelm, I., Jubeh, R., Koch, A. & Zndorf, A. (n.d.). *WhiteSocks - A simple GUI Framework for Fujaba.*
Retrieved from http://www.fujaba.de/fileadmin/Informatik/Fujaba/Resources/Publications/Fujaba_Days/FD07Proceedings.pdf#page=38

Belinfante, A. (n.d.). *JTorX: a Tool for On-Line Model-Driven Test Derivation and Execution.*
Retrieved from http://eprints.eemcs.utwente.nl/17751/01/main.pdf

Belinfante, A. (n.d.). *Documentatie TorXviz.*
Retrieved from http://fmt.cs.utwente.nl/tools/torxviz/

Magee, J., Pryce, N., Giannakopoulou, D. & Kramer, J. (n.d.) *Graphical Animation of Behavior Models.*
Retrieved from http://ti.arc.nasa.gov/m/profile/dimitra/publications/icse00.pdf