Training Report
Realisation of a tool for manipulating and teaching automata
(Draft)

Raphaël Jakse

August 2013

Contents

Introduction

Back to November 2012, I was developing a library to manipulate sets in the D language programming, partly for fun, partly for training myself to the D language programming. This needed advanced knowledge in domains like templates, classes, hash tables, which I was not really used to at that time. This was during the third semester of the computer science Licence at Université Joseph Fourier, in which a unit about automata and languages is taught. At some point, I needed to assimilate some algorithms and definitions of the lesson, mainly determinization and minimization and that gave me an occasion to use my sets library for something useful. I didn't wanted to use plain lists because I wanted to remain close to the definitions and algorithms of the lesson, which were written with sets. Should I write something, I wanted it clean and generic, so I wrote a class for manipulating automata in D, which would support any kind of states or symbols. States could theoretically be numbers, characters, strings, sets, user-defined types or even automata, and so could symbols. As reading textual output is not the easiest way to read automata, I also wrote a piece of code responsible for describing an automaton in the DOT language so Graphviz could draw a beautiful picture of the automaton. With helping other students learn their lesson in mind, I wrote a quick and dirty web interface to use the tool. You entered the automaton with the keyboard, assisted by a basic and crappy interface, chose the algorithm to apply and got a picture representing the resulting automaton.

Then, while I asked Ylies Falcone, who was teaching this lesson, some questions by email during holidays before the exam, he offered to take me in internship. The goal was to make a tool for helping students to learn automata by adding an interactive aspect to the lesson. This included drawing an automata, determinizing / minimizing it and more generally applying any algorithm of the lesson on it, run a word on it and write custom algorithms and test them on drawn automata (because that's how I myself trained to assimilate determinization and minimization).

1 Initial technical choices

Originally, initial choices were done quite simply: I liked the D programming language, was eager to master it a little bit better and found it appropriate for writing my sets library because of its speed, its way of handling types and templates and its simplicity. I knew I could write a decent graphical user interface if I wanted to because of the existence of QtD and GtkD, support major operating systems and use any C existing library because of D's proximity to the C programming language. What I didn't have in mind at that time is using the application from the web. A native application represented full performances and strongly typed programming. Usage on the Web was still possible by quickly writing a PHP page that would call a command line version of the tool. Moreover, letting the user write algorithms him/herself in any script language could be done quite simply given that most languages which can be integrated in a program (e.g. Lua) have bindings in C.

So, before the internship, I tried to learn how to use QtD, downloaded it and compiled it. During the first week of my stage, I quickly wrote a simple interface in QtD to enter an automaton, applying a predefined algorithm on it, and display the result in a textual form. I then managed to integrate Graphviz directly in the program so a more visual way of displaying the result could be used. During this week, I also quite improved the automata class so it was less buggy, more generic, more beautiful. The possibility to use automata with no type constraints on states and symbol was added and the sets library also got some improvement for the occasion.

At that point, several difficulties peeped out:

- It was impossible to find any help on usage and specificities of QtD, the documentation is sparse, nobody use it. Thankfully, Qt itself has a great documentation and most things are the same in D and in C++ and usage of Qt is widely spread.
- Working on templates and types is not easy, especially when you want to handle objects of different types inside a same set in a strongly typed language. Fortunately, D is quite intuitive and the documentation of its standard library, Phobos, is nice.
- Interfacing a C library (here, Graphviz) to a D program was something I never did before, so I had to figure out how to do it. In my case, the trick was to write a small C function that took a string containing some DOT code as argument and returning a string containing the SVG generated by Graphviz. Then, calling this C function from D code is nearly as easy as calling it from C code.

2 Drawing an automaton

Then, it became rapidly necessary to be able to draw automata in a intuitive way, writing code being boring and off-putting for many people. Given that I already was using SVG with Graphviz to draw automata, and that SVG is a vector image format being easily manipulated dynamically, I found natural to work with SVG. With consistency in mind, I adopted a representation close to Graphviz's one thus importing an SVG generated by Graphviz and deducing an automaton from it is completely supported. That's great because you can ask Graphviz to make a rendering of an automaton, and then modifying this rendering with the mouse.

So something handling SVG in a dynamic fashion and handling mouse / keyboard interactions was needed and a web-compatible interface would be a great plus. What comes rapidly in mind is a browser, the dynamic part being Javascript. Qt comes with a component which lets you show some Web contents easily using the Webkit Web rendering engine and let this contents and your application interact together.

So I started the interface for drawing automata: the Automata Designer. It consisted in a very basic xhtml page and a Javascript managing everything. This interface and the program could interact together: the program could push the code of an automaton, the interface could ask the program for the Graphviz rendering of this automaton and the program could ask the code and the SVG picture of the automaton drawn by the user. The difficulties at that time were:

- handling these interactions effectively and in a intuitive way to the user, which would not know that the Automata Designer is technically separated from the rest of the program.
- geometric difficulties: handling Bezier curves given by graphviz, managing the movement of nodes / transitions, etc.
- thinking of how to make the interface independant, so it would be usable outside the program.
- Remaining "intuitive".

3 "Webify" everything

While I was designing the Automata Designer, two concerns were showing:

- I had to thing about a way to let the user write algorithms in a easy and intuitive language
- The program should be easily accessible to the user.

Ideally, the program was runnable from the Web, the user not having to install anything, and could still draw automata and write custom algorithms. At that point, I felt I was in a dead end. On one side, we had to be able to render automata automatically with something like Graphviz, writen in C, and to integrate a scripting language like Lua, also writen in C, so a native application was needed, but an native application is not runnable from the Web: it has to be installed by the user. Being runnable from the Web is a great plus, because it gives possibility to potentially interested people to quickly test your application, and to students to use your thing without being worried to install it.

Ylies suggested Java Web Start for the running from the Web side of the problem, and pointed me to dedicated languages for the algorithmic side of the problem.

Aside from Java Web Start indeed requiring Java and Java Web Start to work, I wondered how to use Graphviz and the Automata Designer from Java, which I didn't know, and how to integrate a scripting language in a Java application.

After reading about dedicated languages and dedicated languages for automata, it turned out that:

1. To use a dedicated language for automata, I would have to write my own compiler / interpreter / whatever because roughly nothing exists
2. Successful dedicated languages tend to become generalist because people need features anyway.

The best option then seemed to use a generalist language and possibly extend it to fit our needs. What seemed to be great was to have good error reporting, and (let's be crazy), have a step by step debugger. Again, Javascript showed itself as a good candidate. Good interpreters, good JIT-compilers, good debuggers, easy to integrate, C-like syntax but weak typing unfortunately. So I started porting the sets and the automata library to Javascript. This was done very quickly, yet the sets library in Javascript is weaker than the D version, although improvable and acceptable for what we need for the moment.

I could port the determinization and minimization algorithms to Javascript. However, the resulting code didn't look well: no foreach, and obligation to constantly convert sets to Arrays and vice-versa, due to Javascript structures working on arrays and automata definition/algorithms being based on sets. The need for writing a new language was becoming important but no way to write an interpreter for a new language:

- this would have resulted in a too much limited language, and in a new language to be grown.
- this would be time-wasting.

So extending Javascript to fit what we needed was a solution that seemed a good compromise. Javascript to which we add a foreach structure is mostly what we were seeking. The idea was to parse a Javascript with foreaches script and to convert it into plain javascript, keeping the same number of line to get errors at the right lines. So I started to write this converter in Javascript, to fit the need for a Web-compatible application.

At that time, we came to a point where we had quite any part of the program writen in Javascript, so I could begin a Web-only version of the program. What most notably missed to this version was Graphviz, or any means to place states automatically to present result automaton.

4 Web-only version: how to draw automata automatically

Dependency to a web server with admin access then appeared inevitable. The rendering of automata would be done server-side by a CGI[1] application calling Graphviz. Each time an automaton would need to be rendered, its code would be sent to the server which would respond with the corresponding SVG picture. That seemed less than ideal:

- The independent character of the application was vanishing. People can't use it off-line if they don't install a web server with CGI and graphviz installed on their machine.
- latencies are introduced.
- A need for paying a dedicated server with admin access just for rendering automata is also introduced.
- Scalability is compromised (what if a lot of users are using the application at the same time ?).

So I searched for clones of Graphviz in Javascript. I first found Liviz.js (JSViz). It seemed awesome, you give it some Dot code, it draws it and can draw it progressively with beautiful animations, etc. There was no an ounce of documentation, so I glanced at the code. I was impressed by its apparent quality, however I never figured out how to use it. What I wanted is just something like svgVersionOfTheGraph = get_svg_from_dot(dotVersionOfTheGraph). Didn't found this. The code was incredibly clean but not commented at all and split into several files. I also started realizing that something generating Graphviz-compatible SVG would be great. I thought I would never find what I needed... until I discovered graphviz.js and viz.js which are projects that compile Graphviz into Javascript thanks to Mozilla's Emscripten project and make it usable easily. As a result, after bringing minor fixes to *graphviz.js* or *viz.js*, the real Graphviz can be used flawlessly client-side, thus wiping out any dependency to a server.

5 Extending a programming language

In our case, extending a programming language means:

- Extending the "standard library": adding a class to manipulate sets and a class to manipulate automata.
- Modifying the grammar of the language: adding features like set manipulations and iteration.

To do this, we basically need a function which takes a file written in our programming language which gives the corresponding pure Javascript code, with the following constraints:

- n^{th} line of the generated code must correspond to the n^{th} line of the input code, for accuracy in error reporting
- The generated code must be identical to the input code if the input code is pure Javascript.

To transform the input code into pure Javascript, regular expressions come in mind, as transformations seem quite simple:

Input code	**Generated code (simplified)**
`foreach(i in object) { ... }`	`object.forEach(function(i){...})`
`varname : Type`	`var varname = new Type`

However, this would be naive: This transformations must not be done inside string literals, *object* in foreach can contain parenthesis, the variable declaration can hold a initialization value, set literals look very similar to blocks of codes, foreach can be nested so a need to match curly brackets appears, break and return statement must be transformed inside foreach loop, etc. Manipulated languages are not *regular*[2] and transformations are not that trivial eventually. As a consequence, another method needs to be used, the code must be analyzed more subtly.

[1]Common Gateway Interface: CGI is a standard method for running and getting the standard output of a script or an application on a web server from an HTTP request.

[2]a language is regular iff it can be described with a regular expression. Rules like "there is the same number of opening and closing parenthesis make a language not regular.

It turned out that the needed flexibility for extending the language comfortably required to recognize each instruction and expression recursively, which rapidly leads to reading corresponding parts of the ECMAScript documentation in order to know how to do it. Special cases and implicit semicolons (which are permitted in Javascript) need to be handled, which complicates things a bit.

A project like Jison, which is a parser like Bison might have been used[3]. However, handwriting the parser was chosen in order to keep whitespace characters intact, to have full control over optional semicolons (if a semicolon was not written by the programmer, the semicolon should not appear in the generated code) and to handle ambiguities between regular expression tokens (which begin with /) and division operators (/, /=) easily. Handwriting the parser also seemed to be a more efficient and straightforward solution here because transformations can be made without generating any abstract syntax tree. Moreover, this let write a quite flexible parser, validation being delegated to the actual Javascript engine, which already has a good error reporting system.

6 Features and comparison with others automata tools

Writing a tool for automata manipulation without looking to already existent projects would be a mistake: a tool with the same goals and the same expectations could already exist, making the creation of a new tool from scratch completely pointless. Contributing to such a project instead of beginning a new one would probably be a better approach.

As a consequence, it is essential to position the project among the others tools of the domain and tel what is new.

Like most tools of the domain, the tool gives the ability to the user to draw automata. However, unlike its friends, the user has more freedom in *choosing the shape of the transitions*, though Visual Automata Simulator is great for this, and thanks to the tight integration of Graphviz, automata can be *(re)drawn automatically* and have a *familiar look*.

Like most others tools, the program comes with basic common algorithms related to automata. However, it comes with *far more algorithms* than its friends and *lets the user write its own algorithms easily*, providing a language close to Javascript with sets as first-class citizens, which makes it suitable for manipulation of automata.

The program is designed with the user in mind: everything is thought to be the more pleasant and natural possible, appearance not being set aside. Ugliness and unfriendliness are bugs. An example of this is the graphical execution of a word: current states are seen in yellow, transitions being taken right now are brown and current final states are green. If a word runs out of the automaton, its states are drawn in red. The execution can be made *step by step* or not, and animations are designed to be beautiful and to ease the visualization of the execution.

What also make the program stand out is the Quiz Feature: it gives the ability to teachers and students to *write quiz for students* and these quizzes are run by the tool, using its capabilities to manipulate automata. Questions of the quiz can be mere multiple choices questions as well as asking the user to write automata or regular expressions.

Another thing that can be said is that unlike others tools, this one is written with web technologies, which makes the program usable without any installation and will make the port on tablets easy. Thanks to web technologies, the program should work on any desktop operating system, provided a recent browser is installed, and the support for mobile operating systems should follow quickly.

In conclusion

Though it was eventually not possible to use the program in concrete applications like real robots, and to do some research during the internship due to a lack of time, which is sad, I really enjoyed it.

[3]See http://cjihrig.com/blog/creating-a-javascript-parser/ for an implementation of a Javascript parser with Jison

What was great in this internship was to reach the computer science research community, to discover what a computer science lab looks like and how people of the lab work. I also really enjoyed working on a project I started myself months before and defining a great part of the internship subject with my supervisor.

I really enjoyed the flexibility of the internship, which is ”come when you want, leave when you want, we are not watching you all the time”. I still work even if nobody is asking me to work, I actually work better if nobody is asking me anything. Yliès widely succeeded in giving me directions, suggesting what would be great, listening to my expectations, instead of just asking me to do defined things. I am all the more eager to listen and actually fill the expectations and do the work. Not having precise schedule yet having a global view of the progression what also good. That’s how I seem to work, and that’s how I would like to work.

This internship strengthened what I felt: research and how people work in this world is probably what suits me the best. It seems you do your things alone in peace, yet you meet your fellows to exchange ideas, broad each one’s horizons, open right or original directions, close wrong directions, progress, go forward. It increased my confidence on my ability to work even if no explicit constraint is imposed. You work alone and together at the same time to discover new and build great things, and that is awesome.

Another great thing is I know that my program will be used to teach automata and this is amazing. I hope it will make automata learning more enjoyable or, at least, easier.