

FATKID: A Finite Automata Toolkit

A THESIS PRESENTED IN PARTIAL FULFILMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
MASTER OF SCIENCE
AT THE UNIVERSITY OF STELLENBOSCH



By

Nicolaas Frederick Huysamen

November 2012

Supervised by: Dr. Jaco Geldenhuys

Declaration

By submitting this thesis/dissertation electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

December 2012

Summary

This thesis presents the FATKID Finite Automata Toolkit. While a lot of toolkits currently exist which can manipulate and process finite state automata, this toolkit was designed to effectively and efficiently generate, manipulate and process large numbers of finite automata by distributing the workflow across machines and running the computations in parallel. Other toolkits do not currently provide this functionality. We show that this framework is user-friendly and extremely extensible. Furthermore we show that the system effectively distributes the work to reduce computation time.

Afrikaans Summary

In hierdie tesis bespreek ons die FATKID Eindige Automaat Toestel. Alhoewel daar reeds toestelle bestaan wat automate kan genereer, manipuleer, en bewerkings daarmee kan uitvoer, is daar egter geen toestelle wat dit op die skaal kan doen wat ons vereis deur die proses te versprei na 'n aantal nodes nie. Ons vereis 'n stelsel wat mew baie groot aantalle automate werk. Die stelsel moet dan die gewensde prosesse in 'n verspreide omgewing, en in parallel uitvoer om verwerkingstyd te verminder. Ons sal wys dat ons stelsel nie net hierdie doel bereik nie, maar ook dat dit gebruikers-vriendelik is en maklik om uit te brei.

Acknowledgements

Some acknowledgements are in order:

- Firstly, to the University of Stellenbosch, who has seen the entirety of my tertiary education, and to whom I owe thanks for giving me knowledge.
- To Jaco, my supervisor, for his guidance and friendship.
- To my parents, Derick and Estha, who provided me with the means to education.
- Most importantly, to my wife, Shareé, who's patience and love kept me going.

For Shareé

Contents

1	Introduction	1
1.1	Design Goals	2
1.2	Requirements	3
1.3	Thesis Outline	4
2	Related Work	6
2.1	The Automata Standard Template Library	7
2.2	Carmel	8
2.3	dk.brics.automaton	11
2.4	FIRE engine	12
2.5	Grail+	12
2.6	The Jacaranda Framework	14
2.7	The MIT FST Toolkit	15
2.8	Tiburon	15
3	Plugin Frameworks	17
3.1	Introduction	17
3.1.1	Java as Implementation Language	18
3.2	Plugin Framework Comparison	21
3.2.1	Eclipse Rich Client Platform	21
3.2.2	Netbeans Platform	22
3.2.3	Java Simple Plugin Framework	24

<i>CONTENTS</i>	viii
3.3 Plugin Framework Choice	25
4 Interface and Architecture	26
4.1 FATKID Modular Architecture	26
4.1.1 FATKID Base	27
4.1.2 FATKID Library	29
4.1.3 FATKID Components	32
4.1.4 FATKID Visual Editor	36
4.1.5 FATKID Engines	37
4.1.6 FATKID Utility Libraries	37
4.1.7 jregula	39
4.2 Extending the Framework	41
4.2.1 Creating a Plugin	41
4.2.2 Creating a Component	42
4.3 Examples of Components	42
4.3.1 Random Generator	42
4.3.2 Minimisation Filter	43
4.3.3 Unique Counter	43
4.4 Filter Composition	44
4.4.1 Merge	44
4.4.2 Split	45
4.4.3 Execute	46
4.4.4 Batch Mode	47
5 Concurrent Framework	48
5.1 Distribution System	49
5.1.1 Search for Available Engine Nodes	49
5.1.2 Loading and Pre-analysis of the Workflow	50
5.1.3 Selecting Output Servers	50
5.1.4 Distribution of Workflow to Nodes	51

<i>CONTENTS</i>	ix
5.1.5 Wait for Engine Nodes to Complete	51
5.2 Concurrent Execution Framework	52
5.2.1 Receive Workflow from Master	52
5.2.2 Analysis and Reconstruction of Workflow	52
5.2.3 Generation of Parallel Execution Pools and Pipeline	52
5.2.4 Execution of the Pipeline	53
5.2.5 Reporting Back to the Master	56
5.3 Example 1	56
5.4 Example 2	61
6 Evaluation	64
6.1 User-friendliness and Accessibility	64
6.2 Extensibility	65
6.3 Performance	65
6.3.1 Experiment Setup	65
6.3.2 Experiment 1	66
6.3.3 Experiment 2	68
6.3.4 Experiment 3	69
6.3.5 Experiment 4	71
6.3.6 Experiment 5	73
6.3.7 Linear Performance and Speedup	75
6.3.8 Analysis of Experimental Results	77
6.3.9 Performance Impact of Batch Sizes	77
7 Conclusion	83
7.1 Future Work	83
7.2 Proposed Future Architecture	85
Appendix A Source Code	89
A.1 DFA Minimization Filter	90
A.2 DFA Generator	92

<i>CONTENTS</i>	x
A.3 DFA Unique Counter	97
A.4 Parallel Execution Pipeline Construction	100
A.5 FatkidPlugin	101
A.5.1 FatkidEditor	102
A.5.2 FatkidEngine	103
A.6 FatkidComponent	104
A.6.1 SourceComponent	106
A.6.2 SinkComponent	108
A.6.3 SourceSinkComponent	110
Appendix B Experimental Data	112

List of Tables

6.1	Experiment Generator Sizes	66
6.2	Experiment 1 – Average Results	67
6.3	Experiment 2 – Average Results	68
6.4	Experiment 3 – Average Results	70
6.5	Experiment 4 – Average Results	72
6.6	Experiment 5 – Average Results	74
6.7	Experiment 5 – Batch Size Difference on Large Set	78
6.8	Experiment 3 – Batch Size Difference on Large Set	80
B.1	Experiment 1 – Small	113
B.2	Experiment 1 – Medium	113
B.3	Experiment 1 – Large	114
B.4	Experiment 2 – Small	114
B.5	Experiment 2 – Medium	115
B.6	Experiment 2 – Large	115
B.7	Experiment 3 – Small	116
B.8	Experiment 3 – Medium	116
B.9	Experiment 3 – Large	117
B.10	Experiment 3 – Batch Size Difference on Medium Set	117
B.11	Experiment 4 – Small	118
B.12	Experiment 4 – Medium	118
B.13	Experiment 4 – Large	119

LIST OF TABLES

xii

B.14 Experiment 5 – Small	119
B.15 Experiment 5 – Medium	120
B.16 Experiment 5 – Large	120
B.17 Experiment 5 – Batch Size Difference on Large Set	121

List of Figures

2.1	Carmel Example Automaton	10
4.1	FATKID Architecture Overview	27
4.2	FATKID Visual Editor	36
4.3	FATKID Equal Distribution Engine UI	38
4.4	<i>jregula</i> Example Automata	41
5.1	Example 1 – Workflow Overview	56
5.2	Example 1 – Analysis Step 1, Source Components	57
5.3	Example 1 – Execution Pipeline After Step 1	57
5.4	Example 1 – Identifying Second Execution Pool	59
5.5	Example 1 – Execution Pipeline After Step 2	59
5.6	Example 1 – Identifying Third Execution Pool	59
5.7	Example 1 – Execution Pipeline After Step 3	60
5.8	Example 1 – Identifying Final Execution Pool	60
5.9	Example 1 – Completed Execution Pipeline	60
5.10	Example 2 Workflow Overview	61
5.11	Example 2 – Identifying Source Components	61
5.12	Example 2 – Second Iteration, Excluding Component	62
5.13	Example 2 – Third Iteration	62
5.14	Example 2 – Identifying Final Execution Pool	63
6.1	Experiment 1	66

LIST OF FIGURES

xiv

6.2	Experiment 1 – Results Comparison Chart	67
6.3	Experiment 2	68
6.4	Experiment 2 – Results Comparison Chart	69
6.5	Experiment 3	70
6.6	Experiment 3 – Results Comparison Chart	71
6.7	Experiment 4	72
6.8	Experiment 4 – Results Comparison Chart	73
6.9	Experiment 5	74
6.10	Experiment 5 – Results Comparison Chart	75
6.11	Experiment 5 – Batch Size Difference Results	79
6.12	Experiment 3 – Batch Size Difference Results	81
7.1	Network-Aware Queue Implementation	88

List of Algorithms

1	Generate Random DFA	33
2	Parallel Execution Pipeline Generation	54

Chapter 1

Introduction

There are many different, mostly academic, tools for manipulating finite automata. One such online list¹ contains more than 60 of these tools. Even though not one of them accomplishes the goals that we set out to achieve with FATKID, we feel that discussing at least some of them is merited.

Bearing in mind that numerous such tools are available, it raises the question of why we want to create another such tool. While existing tools are often very powerful, flexible, stable, and widely used, we were unable to find one that offers the functionality we need for our empirical research in finite automata and language theory. Specifically, none of the toolkits provides a framework in which large numbers of finite automata can be generated, manipulated and processed in an efficient and distributed way.

Some of the existing tools do provide the ability to be extended. This might raise the question why we do not simply extend one of these to provide the required functionality? Our reasoning here is that while extending one of the existing tools could work, the nature of distributed systems are particularly sensitive to the software architecture and designing the framework to be distributed from the ground up results in better performance and greater stability. Also, most of these tools are written in either C or

¹kitwiki.csc.fi/twiki/bin/view/KitWiki/FsmReg

C++, and as we discuss in Section 3.1.1 we have decided to use Java as our implementation language.

1.1 Design Goals

There are many properties that good software should have. We have identified that the following goals are especially desirable and we will use them as a guideline in our work:

- E** *Extensibility*: Users must be able to extend the tool by developing plugins, without detailed knowledge of the underlying architecture, while still gaining the benefits of the distributed execution.
- P** *Performance*: The tool must be able to perform experiments on large numbers of automata (in the order of millions) and reduce the execution time by adding additional worker nodes into the distribution network.
- U** *User-friendliness*: Users must be able to easily use the tool and be able to create complex workflows in a visual environment.
- A** *Accessibility*: The tool must be open-sourced and well-documented.

We regard all four of these attributes as absolutely essential. A tool with an intuitive interface would soon fall into disuse if the performance was poor, and extensibility is important for ensuring that a tool remains useful beyond its initial deployment. While it is important not to optimise code too early on in the application lifecycle, performance is at the heart of the tool's stated purpose of handling large sets of automata. With this in mind, we have also developed the *jregula* automata library which we discuss in Section 4.1.7. While this library may initially have limited functionality, its design is such that it can manipulate automata efficiently with a minimal

memory footprint, and can easily be extended to cater to the users' specific needs.

Most importantly, we want to take care to craft a robust design that would not require later re-engineering. Furthermore, in our experience, good documentation is essential for developers and users alike. Bad documentation is a hallmark of both academic and industrial projects, and is a greater project-killer than many other flaws. This may all seem very generic (and perhaps it is), but we could point out that while other attributes such as portability are also important, we do not regard them as essential (even though our implementation language choice does make the system extremely portable). We regard *correctness* as one of the least important attributes: we believe in vigorous testing, and in our experience we have found that it is generally easy to spot and correct low-level bugs, as long as the overall design is solid. A more general guiding principle that we try to adhere to at all levels of design and implementation, is the universally wise advice:

S *Simplicity*: Make it as simple as possible.

As we discuss in Sections 4.1.2 and 4.2, it is easy for users to create their own components and plugins² to work with the framework. We refer to the goals above (**U**, **P**, **E**, **A**, **S**) throughout this paper, and use these abbreviations as a short-hand.

1.2 Requirements

While keeping the five properties discussed above in mind, we now list the requirements that we have identified as essential for FATKID.

²Throughout this thesis we will refer to *plugins* and *components*. A plugin is an extension to the framework in terms of an implementation of an engine or editor, while components refer to the implementation of components; the elements that form the activities in a workflow.

- Users must have an intuitive graphical interface where they can create complex workflows using a drag-and-drop type interface.
- Developers must be able to easily write their own plugins and components to work with the toolkit, have the plugins appear in the GUI and the components in the editor, and seamlessly integrate with the system as a whole.
- Users must have their workflows executed in a distributed environment without any in-depth knowledge of how distributed systems work.
- Users must have the system execute the workflows with high performance and scale well when extra worker nodes are added into the distribution network.

1.3 Thesis Outline

Chapter 2: Related Work explores some of the automata toolkits and libraries that are currently available. We briefly discuss each of the selected tools in terms of their strengths and weaknesses with regards to the design goals we have set out. We also mention if and how some of these libraries can be used to integrate with FATKID in the form of components.

In *Chapter 3: Plugin Frameworks* we look at possible frameworks that will enable FATKID to be an easily extensible framework. We briefly discuss the strengths and weaknesses of each, and argue why we chose to go with the Java Simple Plugin Framework (JSPF).

Chapter 4: Interface and Architecture forms the first core part of this thesis. We begin to discuss how the FATKID framework consists of multiple components that interact with each other to form a distributed, concurrently executing framework.

Chapter 5: Concurrent Framework, together with Chapter 4, forms the core of the thesis. In this chapter we discuss how the distribution engine generates the most efficient way in which to execute workflow steps in parallel without them interfering with one another, and how the framework enables the distribution of workflows by abstracting most of the technical workings from the user.

Chapter 6: Evaluation presents the results of the experiments conducted to show the performance of the system with variable numbers of worker nodes in the distribution network. We also evaluate each of the design goals that we set out.

Lastly, in *Chapter 7: Conclusion*, summaries and conclusions are presented, along with possible future work that can be done on the framework.

Chapter 2

Related Work

As we have mentioned in the introduction, numerous automata toolkits do exist already. During our literature study of these existing toolkits, we have come to the conclusion that there are currently no toolkits that can achieve exactly what we require from FATKID. None of the toolkits provide the functionality to generate large amounts of automata, manipulate them, and do various calculations with them, in a distributed environment. In fact, as we will discuss in more detail below, most of the toolkits are designed to handle only single or a very small number of automata at a time.

Another observation is that even though some of the existing toolkits are indeed extensible, none of them are extensible to the degree that we would deem sufficient for what we envision. The most prominent shortcoming is that extending these tools requires a commitment to their internal automaton representation. Secondly, we are of the strong opinion that extending existing tools which were not specifically designed to be distributed to be used in such a way, will result in sub-optimum results. The purpose of the FATKID toolkit is to give the plugin developer complete freedom over the implementation of the automaton representation and workings, even though we provide an automaton implementation library with basic manipulation capabilities. In fact, by using the Java Native Interface (JNI) the developer

can use C or C++ for his automata implementation. In the following sections, we will give very brief overviews of some of the finite state libraries and toolkits that are available currently.

2.1 The Automata Standard Template Library

The Automata Standard Template Library (ASTL) [5, 6, 7] is a framework originally developed by Vincent Le Maout at the University of Marne la Vallée. It is implemented in C++ and provides a large amount of generic components for the manipulation of automata. The components available in the framework follow the Standard Template Library (STL)¹ approach of component design. In short, this means that the aim of ASTL is to provide generic algorithms for automata, written once, for all types of automata.

It should be noted that template (or generic) programming should not be confused with the concept of inheritance programming, which is a fundamental concept of object orientation. As stated in the ASTL documentation [7], while objects and encapsulation are used, inheritance on the other hand is very rarely used. The design centres around template manipulation, which offers an increase in efficiency.

Features

The library provides implementations for both DFA and NFA type automata, as well as a number of the fundamental algorithms for the manipulation of automata. The full list of algorithms that the library supports can be seen in the ASTL documentation [7]. The algorithms available for

¹The Standard Template Library (STL) [8], available in standard C++, offers a collection of container classes, algorithms, and iterators that provide most of the basic data structures and algorithms in modern computer science.

the manipulation of DFA include acyclic minimisation², DFA duplication using either breadth first search or depth first search, computation of the language recognised by the DFA, determining if a word is accepted by the DFA, writing of the DFA to file in either ASCII, binary or VCG³ format, computing the union, intersection and symmetric differences of DFAs and counting the number of words recognised by a DFA. Only two algorithms are available for the manipulation of NFAs. These are determinisation of the NFA and writing the NFA to file in ASCII format.

Limitations with Regards to Our Requirements

The goal of the ASTL project was to build a generic template automata library. It does not provide a framework for the automatic generation and manipulation of automata in a distributed environment, and therefore lacks the required functionality that we want from FATKID. It also falls short since it provides implementations of automata, and thereby binds the user to this implementation when using it.

This however does not mean that it cannot be used with FATKID. As was mentioned earlier in this chapter, and should be made clear after reading Chapter 3, by using JNI it is possible to build components that use ASTL, despite FATKID it being written in Java.

2.2 Carmel

Carmel is a finite state transducer package [10, 11]. It was written by Jonathan Graehl at the University of Southern California.

²A precondition to this algorithm is that the DFA must be acyclic, meaning that no path starting from state q_i is allowed to encounter state q_i again.

³The Visualization of Compiler Graphs (VCG) [9] tool reads a textual specification of a graph in VCG format, and displays it in a visual format.

Features

Carmel is able to do various operations on finite state acceptors and finite state transducers, with or without weighted transitions. It also supports finding the k -most likely strings that are accepted by a given finite state acceptor, as well as forward and backward training (as can be found in hidden Markov models).

Limitations with Regards to Our Requirements

During the investigation of Carmel, it became apparent that, while it is extraordinarily good with the handling of finite state acceptors and transducers, it does not fit the needs set out by our requirements perfectly. The first shortcoming identified is the fact that Carmel typically handles single finite state acceptors or transducers at a time. Input is defined either by specifying an input file or receiving input from another application with the use of piping⁴. The classic example of an input file, as can be seen at [10], is included below. The resulting finite state machine can be seen in Figure 2.1. This type of finite state creation is very verbose and would make some of the desired functionality required from FATKID tedious and potentially sub-optimal, for example the generation of a large number of random finite state machines.

⁴Piping here refers to the standard UNIX pipe (‘|’) command whereby output from one application can be used as input to another in a single command. Similar functionality exists on most modern operating systems.

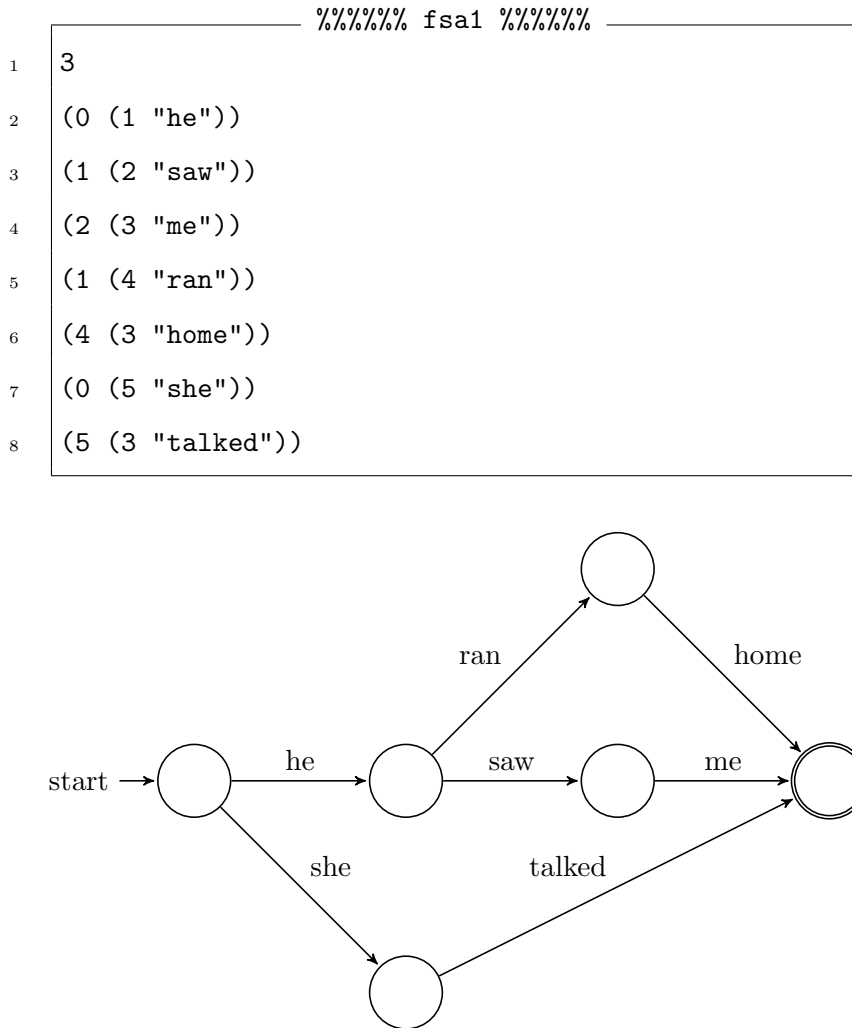


Figure 2.1: Carmel Example Automaton

Carmel is a popular and powerful package, but it is clearly not very well suited to our needs. It does not support traditional automata in an efficient way and contains little direct support for the type of integration and extension that would be required to achieve our requirements. From the documentation that was available, it seems that the internal components can not be used in other projects. This means that even by using technologies such as JNI, Carmel will not be able to integrate with FATKID components.

2.3 dk.brics.automaton

The `dk.brics.automaton` [12] package is a finite state automata library written in Java by Anders Møller at the Aarhus University.

Features

The library has support for the standard operations that can be performed on regular expressions, which include concatenation, union, and Kleene star. It also supports some of the more non-standard operations such as calculating the intersections and complements of languages. It also boasts to be one of the fastest automaton/regular expression packages [12].

Limitations with Regards to Our Requirements

The first notable shortcoming of the `dk.brics.automaton` library is that by its own definition it is not a framework for the generation and manipulation of large numbers of automata. Rather it is a library that provides finite state automata implementations and algorithms for the manipulation thereof. However, the desired method of generating automata seems to be converting them from regular expressions, and that its true power lies in that usage. The use of regular expressions makes it difficult to generate random automata in a structural way.

It is however possible to programmatically create automata with this library by using the available API methods, and we have built a few small components that worked perfectly with the FATKID framework, showcasing the ease with which external libraries can be incorporated. Therefore, while this library by itself does not meet our requirements, it is a possible candidate for implementing a set of components to be used in with the FATKID framework. Extension of this library to include our specific requirements however will bind the user to use the implementations provided by the library, which is not desired.

2.4 FIRE engine

The FIRE engine is an excellent C++ class library developed by Bruce Watson at the Eindhoven University of Technology. It implements finite automata and regular expression algorithms [13].

Features

The FIRE engine consists solely of a class library that provides classes and algorithms that can be used in other applications that require work with finite automata or regular expressions. No interactive user interface is provided. It states that it provides algorithms for almost all of the algorithms presented in the taxonomies of finite automata algorithms [14].

Limitations with Regards to Our Requirements

The most notable shortcoming in the FIRE engine library is the fact that it consists solely of a class library. By itself it provides no further functionality other than providing implementations of finite automata and algorithms. This makes it a very good candidate to be used in designing components for the FATKID framework. By using the JNI technology to link Java with C/C++ code, one should be able to build a robust set of components.

2.5 Grail+

Grail+ is a set of 54 command-line applications [15, 16] developed at the Department of Computer Science at the University of Western Ontario. It provides a way of generating and manipulating finite state machines, regular expressions, and finite languages. All the applications are written in such a way that it defines a standard format for all input and output, thereby enabling the conversion from one form to another.

Features

Grail+ is probably the toolkit that comes the closest to what we expect from FATKID in that it is possible to build workflows by combining multiple operations that send data to each other. As mentioned previously however, Grail+ only provides a set of *command-line* applications, which makes it less user-friendly, especially when building large complex workflows. The various applications work together by piping the output from one to another, or receiving input from files. An example of combining more than one of these applications can be seen below. The example takes a regular expression, transforms it into a non-deterministic finite automata, determinises the automata, and converts it back into a regular expression. This example, and many others, can be seen in the *User's Guide to Grail* [16].

```
% echo "(a+b)*(abc)" | retofm | fmdeterm | fmtore  
(aa*b+bb*aa*b)(aa*b+bb*aa*b)*c
```

While this approach is familiar to anyone who is used to the operation of UNIX shells, it can be very difficult to learn and may make it difficult to design complex workflows. Unfortunately, Grail+ has no support for executing the workflows in a distributed way. Concurrent processing could potentially be achieved by a set of complex shell scripts that make use of Grail+ tools, but this approach is error-prone, hard to manipulate, and requires a significant amount of human involvement. Furthermore, assuming that a concurrent system such as described existed for Grail+, the constant conversion of automata between their internal representation (during code execution) and their external representation (textual format output to the shell) will result in sub-optimal performance.

It should also be noted that Grail+ can be used as a C++ class library in other applications, rather than a set of command-line applications.

Limitations with Regards to Our Requirements

Grail+ is extensible: filters can be added and existing components can be extended. But having to build a concurrent and distributed system from a framework that was never designed to be used in this way would have most probably resulted in an unstable and sub-optimal solution. Also, as we discuss in Section 3.1.1, we decided that Java was a more appropriate language for implementation. There is however no reason why using Grail+ as a C++ class library to build components, in a similar way as discussed for the FIRE engine, cannot work.

As a last note, the last release of Grail+ was a preview release in 2002. It would seem that this project is no longer being maintained.

2.6 The Jacaranda Framework

The Jacaranda framework is a customisable finite state machine package implemented in Java [17]. It was written by Sandro Pedrazzini.

Features

The Jacaranda framework aims to provide an easy and customisable design for the implementation of different finite state machines. It boasts that it differs from other existing frameworks in that it offers customisable parts through which users can change various aspects of the finite state machine.

Limitations with Regards to Our Requirements

As with most of the other libraries, Jacaranda only offers a finite state machine implementation. Also, input is read from files and as such makes it difficult to use to produce high volumes of automata.

2.7 The MIT FST Toolkit

The MIT Finite-State Transducer Toolkit is a collection of command-line tools (and their associated C++ APIs) for manipulating finite state transducers and finite-state acceptors [18, 19]. It was designed as an educational tool, but is flexible and efficient enough to be used in production level applications.

Features

The MIT FST Toolkit provides functionality for the construction, combination, and training of weighted finite state acceptors and transducers. The toolkit was designed with three goals in mind: ease of use, flexibility, and efficiency. As with the Grail+ framework discussed in Section 2.5, the MIT FST command-line tools also provide a universal representation of the finite state acceptors and transducers so that multiple operations can be combined by using the UNIX shell pipe feature. It is recommended to use the C++ API to integrate it with larger systems.

Limitations with Regards to Our Requirements

The MIT FST toolkit is extensible in exactly the same way as Grail+: users can add filters and change or add new command-line tools. However, the same objections apply here. The system was not designed with distribution and concurrent execution in mind; retrofitting it to operate in this way may very well result in a poor architectural design.

2.8 Tiburon

Tiburon is a tree transducer package written by Jonathan May at the University of Southern California [20, 21].

Features

Tiburon was designed to handle weighted regular tree grammars, context-free grammars, and both tree-to-tree and tree-to-string transducers. It can perform composition, intersection, application, determinisation, inside/outside training, pruning, return k-most likely trees, Viterbi derivations, and many other useful things [20, 21, 22].

Limitations with Regards to Our Requirements

As with many of the other frameworks, Tiburon was not designed to work with a large number of inputs, nor to do calculations in a distributed way.

Chapter 3

Plugin Frameworks

A software framework is defined as a software abstraction layer that provides generic functionality to a developer. This functionality can then be selectively changed by the developer with their own custom code, and through this action, the framework becomes application specific [30]. There are various types of categories that software frameworks fall under, including support software, compilers, and toolkits. All types provide an environment for easier and faster software development, and through the re-use of the tested framework components, applications become more stable. This chapter will focus on the frameworks that we investigated to realise our goal of extensibility by enabling the ability to write custom plugins for the FATKID framework.

3.1 Introduction

Before we start our discussion on the plugin frameworks, we feel that we need to clarify the differences between frameworks and libraries, as these are often, and incorrectly, used synonymously. Software frameworks contain key distinguishing features that separate them from normal libraries:

- *inversion of control* - In normal libraries or user applications, the pro-

gram flow is usually dictated by the caller (i.e., the user specific code calling the library code). In a framework however, the flow is dictated by the framework itself.

- *default behaviour* - A framework should always define default behaviour. Not only that, the default behaviour should also be useful and useable.
- *extensibility* - A framework provides extension points where developers can selectively add custom code to refine the functionality of the framework. In contrast, a library provides a service to the developer. While some libraries do provide a mechanism of extension, it typically only enables extension in terms of adding new parts, not refinement the existing functionality.
- *non-modifiable framework code* - In general, the actual implementation code of the framework itself is not allowed to be changed. Any specific changes that the developer wants to make needs to be done via the extension points mentioned in the previous point.

3.1.1 Java as Implementation Language

While languages such as C and C++ are traditionally a much more popular choice when it comes to the implementation of automaton libraries and toolkits, especially in the academic community, we decided to implement ours in Java. In the following sections, we will defend our decision to use Java by discussing a few key points.

Platform Independence

Platform independence is one the most widely used reasons why software developers choose interpreted languages like Java as their programming language of choice. Even though there are various other languages that can also boast the feature of platform independence, while some contain at least some

degree of it, not many of them are as established, stable, well-supported, and with a large number of 3rd party libraries such as Java.

For FATKID, platform independence is vitally important. The nodes in the distributed system are not necessarily homogeneous, and as such the overhead of having to either implement and/or compile different versions of components for each of the platforms that Java already support, is still not even close to the overhead and inconvenience to the user of the system who has to set up these different versions in a non-homogeneous environment.

Performance

It might seem strange at first that we list performance as a factor during the process of choosing Java as our implementation language. Probably even more so when compared to languages like C and C++. During the earlier years of Java it was rightfully considered to be much slower than compiled languages, since it is a 2 step interpreted language¹ itself. However, with the introduction of the Java Just-in-time (JIT) compiler and HotSpot optimisations, this is not longer necessarily the case [23, 24]. Even though the Java compiler still compiles the source code to platform independent bytecode which is then interpreted by the Java Virtual Machine (JVM), the optimisations mentioned above brings the execution time to near-native speeds, and in some scenarios even faster than the traditional compiled languages.

Just-in-time compilation, also called dynamic translation, is a process where, during runtime, portions of the bytecode are compiled to native machine code and then executed rather than the bytecode. This makes

¹A 2 step interpreted language is a languages that requires 2 steps to bridge the gap between source code and execution. During the first step, the source code is compiled into bytecode. Bytecode is not executable natively on any machine. The bytecode is interpreted by a virtual machine that translates the bytecode into machine operations at runtime.

the application as a whole run at near native speeds, since the compiled portions do run at native speed. The JIT compiler is also what formed the source of what is now called HotSpot. HotSpot continually analyses the program's execution and tries to identify "hot spots" in the program flow. These hot spots are parts of the application that are frequently and/or repeatedly executed. The JIT compiler then compiles these parts to native machine code. This type of adaptive optimisation makes it possible for Java to outperform some native hand-coded C or C++ code.

The reasoning and method behind the adaptive optimisation is discussed in *The Java HotSpot Virtual Machine, v1.4.1 White Paper* [23]. It states that an interesting property of virtually any program is that it spends a vast majority of its time executing a very small portion of its code. Now, rather than compiling method by method, just in time, the Java HotSpot starts out by running the application interpreted. It then analyses the program code as it executes and tries to detect the critical parts of the code. These parts of the code are then focussed on and compiled to native code.

Ease of Extensibility

While languages like C and C++ do provide the functionality to build systems that are de-coupled by the use of dynamically linked libraries, it still involves a relatively complex procedure to do this when compared with Java. Java makes it incredibly easy with the use of frameworks like JSPF, discussed in Section 3.2.3.

User Demographic

While C and C++ used to be, and to some degree still are, very well known languages, the popularity amongst younger developers have drastically declined. With Java being taught in increasingly more schools and universities, we made the decision that to make our framework more accessible to the

largest number of people, Java would be a much better choice.

3.2 Plugin Framework Comparison

We will now continue with our comparison of the three plugin frameworks that we investigated; the Eclipse Rich Client Platform, the Netbeans Platform, and the Java Simple Plugin Framework.

3.2.1 Eclipse Rich Client Platform

The Eclipse Rich Client Platform (RCP) is a platform for the development of applications that are based on a dynamic plugin model [25]. The Eclipse IDE itself is also written using the Eclipse RCP. While this seems like a perfect fit for what we envision for FATKID, we discuss the various aspects we had to take into consideration. It should be noted that during the start of this project, the Eclipse 4 Platform was not yet released. Therefore, any mention to the Eclipse Rich Client Platform (RCP) is a reference to the 3.x (excluding 3.8) line of platform releases. The 3.0 line of Eclipse was released in 2004 and was the first version to support the re-use of the platform in stand-alone applications [26].

Advantages

Probably the biggest advantage of the Eclipse RCP is that it is a mature, stable, and widely used platform. Many big companies like IBM and Google have used this in some of their products [26]. It also has a very large community of followers, which makes getting help and guidance relatively easy.

Disadvantages

While the advantages described above does make the Eclipse RCP a good candidate for developing FATKID, it still requires the developer to include a certain number of required components from the Eclipse framework. This is quite understandable as some components of the framework are required to provide the functionality. However, this introduces an unnecessary amount of bulk to the application. While it might not be a problem, as it should only affect the base FATKID module, there are a few other points that steered us away from this platform. The most important and relevant of these is that one of the design goals of the project was to make it as easy as possible for developers to extend the framework. If the Eclipse RCP was used, developers would have had to spend time learning how the Eclipse RCP works and how to write extensions for it. This is not a trivial task, especially compared to a framework such as JSPF that we discuss in Section 3.2.3.

Conclusion

For our use-case, using the Eclipse RCP platform is overkill. The complexity required to implement even the simplest plugin is too high when compared to other plugin frameworks. This goes against our design goal of *simplicity*.

3.2.2 Netbeans Platform

The Netbeans Platform is a generic platform for the development of Swing applications [27]. As with the Eclipse RCP and the Eclipse IDE, the Netbeans IDE was built using the Netbeans Platform. Even though your application does not strictly need to look anything like an IDE when using the platform, certain elements do pull through in the general use of the platform.

Advantages

As with the Eclipse RCP, the Netbeans Platform is also mature, stable and flexible. It has a big community of followers and the documentation on the Netbeans website is complete. The platform does eliminate the need for a lot of boilerplate code, such as saving state and connecting actions to menu items [27] which would normally be left to the developer to implement each time. Lastly, similar to the Eclipse RCP, implementing the simplest plugin has a much higher complexity than necessarily required.

Disadvantages

The Netbeans platform is in most aspects very similar to the Eclipse RCP. Therefore, the same objections we discussed previously apply the Netbeans Platform too.

Netbeans Visual Library

One special mention regarding the Netbeans Platform should be made, and that is regarding the Netbeans Visual Library which forms part of the platform. By investigating the package structure of the Netbeans Platform, and because of its modular design, it is possible to use some of the sub-components, such as the Netbeans Visual Library, with minimal inclusion of the other modules. This is also exactly what we did during the development of the FATKID Visual Editor. The Netbeans Visual Library provides a framework for creating visually interactive systems. It provides all the functionality that we require in the FATKID Visual Editor, including the creation of widgets, connections, and the interaction between them.

Conclusion

We feel that the Netbeans Platform, as does the Eclipse RCP framework, provides unnecessary bulk and complexity to the project. It also ties plugin

developers into using the Netbeans Platform when writing their own plugins.

3.2.3 Java Simple Plugin Framework

The Java Simple Plugin Framework (JSPF) is a framework that enables the rapid development of modular small to medium sized systems [28]. We have found that the JSPF framework really holds up to that claim. It is small in comparison to the other frameworks discussed, requires no configuration and works perfectly without the need to include any other components.

Advantages

The biggest advantage of JSPF is the fact that it does not tie the plugin developer into any specific technology or framework (except Java). The only thing required to convert an ordinary Java class into a plugin, is annotating the class with a simple annotation provided by the JSPF framework. After this, the FATKID framework (using the JSPF plugin manager) automatically picks up and loads that class into its classpath.

Disadvantages

In all honesty, we have not found a single disadvantage while using the JSPF framework. The framework is complete, stable, and has not once given problems during the development of FATKID. The framework is also still under active development with features being added continually.

Conclusion

JSPF is a very small and lightweight framework for developing plugins. It can easily be extracted into the FATKID library, making it possible for developers to create plugins for FATKID without having to import any external libraries, except the FATKID library.

3.3 Plugin Framework Choice

After taking into consideration everything that we have investigated regarding plugin frameworks, we decided that using the JSPF framework would fit our requirements the best. Not only does it considerably reduce the complexity for developers to write plugins, it is also very small and lightweight, keeping the application small and responsive.

Chapter 4

Interface and Architecture

The FATKID framework consists of a number of loosely coupled libraries and plugins that work together to provide the required functionality. This approach to application design makes for high levels of code re-use. It also enables easier testing of the separate modules. In the following sections we describe the modular architecture of the FATKID framework, how the framework can be extended, and examples of extensions.

4.1 FATKID Modular Architecture

The FATKID framework architecture and user interface are based on what is sometimes referred to as a pipeline or pipes-and-filters design pattern. The interface allows users to “compose” scenarios by creating and connecting instances from a palette of filters. Each filter has (1) an arbitrary number of inputs, (2) an arbitrary number of outputs, and (3) a set of configuration settings. In accordance with **S** (simplicity), each filter is as simple as possible. This makes it easier to implement correctly and test thoroughly. In general, it also makes the automatic parallelisation of the experiments much easier.

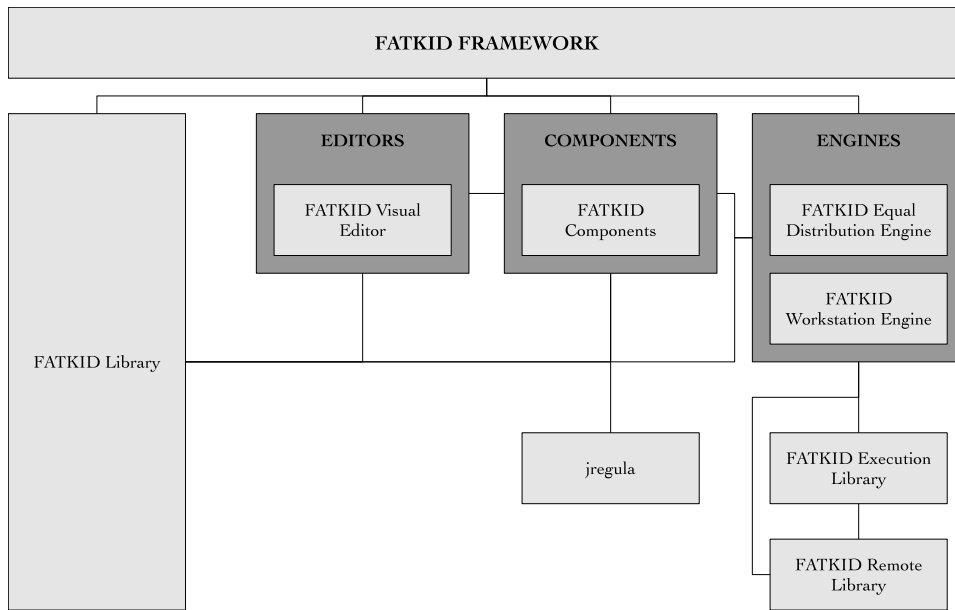


Figure 4.1: FATKID Architecture Overview

4.1.1 FATKID Base

The base FATKID framework is a thin client that makes use of the various modules and plugins to provide a feature-rich environment. An overview of the modular design of the framework can be seen in Figure 4.1. Before we begin describing the architecture of the FATKID framework, it should also be mentioned that FATKID makes use of an event bus system to enable some of its functionality. We give a quick overview of the event bus system so that the context is defined for the rest of the descriptions.

The event bus is a very important aspect of the FATKID framework. It allows any and all plugins and components to be able to communicate with each other by firing events onto the event bus. They can also receive events by registering themselves on the event bus for specific events. The base framework initialises the application level event bus on startup. The event bus is a singleton object available to all plugins and components. An example use of the event bus follows. We show how a component can register

itself on the event bus to listen for the `ComponentsLoadedEvent`. This event is fired as soon as FATKID has loaded all the components into the classpath. The event itself also contains a collection of all the components, and as such any object that receives this event will have access to the collection of components.

```
EventBus.getInstance()  
    .registerEventHandler(  
        this, ComponentsLoadedEvent.class);
```

This requires that the component calling this function implements the `EventHandler` interface provided by the FATKID library. Now, whenever any component in the system fires a `ComponentsLoadedEvent`, the object will be notified. Similarly, it is just as easy to fire an event onto the event bus. For example, the code for firing an event that will update the string being displayed in the information bar at the bottom of the application frame can be found below. The second argument to the `UpdateInformationBarEvent` constructor states if the message is an informational message, or an error message. A value of `true` indicates an error and will result in the message being displayed in red.

```
EventBus.getInstance()  
    .fireEvent(new UpdateInformationBarEvent(  
        "Cannot create connection, cycle detected",  
        true  
    ));
```

As a framework, the base FATKID application has no knowledge of how to build, distribute or execute anything. Rather, it depends on the various plugins and components to do this. During startup, FATKID inspects the `plugins` folder and loads any plugins and components that are present. These are registered in the application's internal plugin registry for easy access. The framework also fires a `PluginsLoadedEvent` after all the plugins are loaded, and a `ComponentsLoadedEvent` after the components are loaded.

If the correct startup script is used, the `lib` folder is added to the application classpath. This folder should include any library modules. These include the FATKID library, the execution library, and the remote library. This means that the plugins and components do not have to include the libraries in their JAR files. By sharing the libraries in this way, extensions can be significantly smaller in size.

4.1.2 FATKID Library

The FATKID library forms the core of all the system modules and plugins. It exposes the shared interfaces and classes that components and plugins require to be used in the system. It also includes an extracted version of JSPF, making it possible for developers to use it without having to link to any external library other than the FATKID library. This library is at a minimum the only library that a plugin or component developer would be required to use when developing new extensions.

Provided Interfaces

The library provides a number of interfaces which should be adhered to when developing components and plugins:

- **Plugin Interfaces**

- *FatkidPlugin* (Appendix A.5) - This is the base plugin interface that defines only three basic methods. A developer should never actually implement this interface, but rather one of the interfaces that extends this one.
- *FatkidEditor* (Appendix A.5.1) - The editor interface defines all the methods required from a plugin to be implemented as an editor plugin to integrate with the FATKID framework. It defines

a single method that returns a `JPanel` which the base framework will place and draw.

- *FatkidEngine* (Appendix A.5.2) - This interface is for all intents and purposes identical to the editor interface. The reason for having them separate is so that the framework can distinguish between the two at runtime and place them in the correct categories. An engine plugin is also required to return a `JPanel`, as engine components have graphical user interfaces, as can be seen in Figure 4.3.

- **Component Interfaces**

- *FatkidComponent* (Appendix A.6) - The `FatkidComponent` interface defines the contract for a component in the workflow of an editor. This interface, even though defining a plugin in the system, should not be confused with the `FatkidPlugin` interface. They are handled differently in the system. The `FatkidPlugin` interface should be used for functional components in the system (e.g., editors, engines), while the `FatkidComponent` interface should only be used to define components in the workflow of an editor.
- *SinkComponent* (Appendix A.6.2) - The `SinkComponent` defines the interface for a component in the FATKID workflow that acts only as a sink (a component that accepts input).
- *SourceComponent* (Appendix A.6.1) - The `SourceComponent` defines the interface for a component in the FATKID workflow that acts only as a source (a component that generates output).
- *SourceSinkComponent* (Appendix A.6.3) - The `SourceSinkComponent` defines the interface for a component in the FATKID

workflow that acts as both a source (a component that generates output) and a sink (a component that accepts input).

- **Component Abstract Classes** - The library does provide three abstract classes (*AbstractSinkComponent*, *AbstractSourceComponent* and *AbstractSourceSinkComponent*) which implement their respective interface counterparts, but does some property management for you automatically. These are convenience classes that eliminate the need to write some boilerplate code¹.
- **Property Classes and Interfaces**
 - *Property* (class) - The *Property* class defines a single property that can be set on a component. This provides a mechanism to the user to specify and change the behaviour of components in different stages in the workflow (e.g., setting the number of items to generate). To make the property as generic as possible, all property values are strings, and the user should therefore define parsing functions in his components to convert the values to the desired types.
 - *PropertyValidator* (interface) - The implementation of a property validator should validate if the string representation of a value is valid for the containing property. This is used in the FATKID editor when users are editing properties set on components. Invalid values should not be accepted.
 - *PropertyDistributor* (interface) - The implementation of a property distributor should implement the logic of how this specific property value should be distributed (divided) between the nodes in the computational grid. In the absence of a property distrib-

¹Boilerplate code is code that typically has to be included in lot of different places in source code, but with no or very little alteration.

utor, the property will be distributed as is to all nodes. We speculate that this feature will very rarely be used, as the default behaviour of distributing the values of properties as duplicates is probably the desired use-case most of the time.

Workflow Utilities

The FATKID library provides a generic Java representation of what a workflow should look like in the system. Even though it is not required that our version be used, it does provide a single implementation that should fit most needs. Included is also a utility class that knows how to serialise and de-serialise the workflow to file.

4.1.3 FATKID Components

The FATKID components library is a collection of various components that demonstrate the functionality and usage of the FATKID framework. All of the components are built using *jregula*, which is discussed in Section 4.1.7. We will briefly discuss each of the provided components.

DFA Generator

The DFA generator implements a very simple random DFA generator. It produces a list of randomly generated DFAs, constructed using the property values as set by the user. It supports the following properties:

- *Number of DFAs* — This property indicates the number of random DFAs to generate.
- *Number of States* — The number of states that should be present in each DFA generated.
- *Number of Symbols* — The number of symbols in the alphabet of the DFA.

- *Number of Accepting States* — The number of states that should be considered accepting.
- *Transition Probability* — The probability that a state will have a transition on a specific alphabet symbol.

The algorithm for generating a single random DFA is presented as Algorithm 1. The S parameter denotes the (pre-computed) set of states, the l parameter denotes the size of the alphabet, the f parameter denotes the number of final states, and the δ parameter is the transition probability.

Algorithm 1 Generate Random DFA

```

1: procedure GENERATERANDOMDFA( $S, l, f, \delta$ )
2:   def  $I \leftarrow \text{nil}$  ▷ Initial state
3:   def  $A \leftarrow A \subseteq S$  and  $A_{size} = f$  ▷ Set of accepting states
4:   def  $T \leftarrow \emptyset$  ▷ Set of transitions
5:   def  $\lambda \leftarrow \emptyset$  ▷ The alphabet
6:   for symbol  $\sigma \in \{1..l\}$  do
7:      $\lambda \leftarrow \lambda \cup \sigma$ 
8:   end for
9:   for state  $s \in S$  do
10:    for symbol  $\sigma \in \lambda$  do
11:      def  $d \leftarrow U(0, 1)$ 
12:      if  $d \leq \delta$  then
13:        def  $t \leftarrow S_i$  where  $i = U(0, S_{size})$ 
14:         $T \leftarrow T \cup (s \times \sigma \rightarrow t)$ 
15:      end if
16:    end for
17:  end for
18: end procedure

```

DFA Minimiser

The DFA minimiser component implements a *source-sink* component that takes as input a list of DFAs, minimises them, and outputs the list of minimised DFAs in the same order as they arrived. It currently supports two algorithms of minimisation, namely the Hopcroft algorithm [1, 2] and the Brzozowski algorithm [3].

DFA Normaliser

The DFA normaliser takes as input a list of DFAs, renames all the states and symbols into a uniform order, and outputs the list of normalised DFAs in the same order which they arrived in.

DFA Unique Counter

The DFA unique counter takes as argument a list of DFAs and calculates the number of unique DFAs in the list. It achieves this by comparing DFAs by their structure alone, not by what language they accept. It also currently supports two methods of achieving this. The first requires an exact match to be found before two DFAs are considered structurally the same. This method however consumes quite a large amount of resources, as DFAs have to be serialised to a string and stored to be compared with all other DFAs.

The second method is a heuristic method. While it still serialises each DFA into its string representation, it only uses the hexadecimal MD5 hash of the automaton. Since this is a one-directional, non-unique algorithm, the chance does exist that two different DFAs hash to the same value, even though this chance is very small. To calculate the probability of two automata having the same hash value, we refer to a problem called the *Birthday Paradox* [32]. This problem concerns itself with calculating the probability that, given a set of n random people, some pair will have their birthday on the same day. It can easily be extended to fit the model of finding the

probability that two values in a set having the same hash value.

The MD5 hash algorithm uses 128 bits (or 32 hexadecimal characters) to store its value. By looking the probability table presented in [32], to achieve a probability of only $10^{-18}\%$ that two values have the same hash value, you will require a set of 2.6×10^{10} values. To achieve a probability of 1%, the required number of values raises to 2.6×10^{18} .

DFA Writer

The DFA writer is a very simple component that merely serialises a DFA to its string representation, and prints it out to the console. This component was developed for testing purposes and showcases the use of a sink component, but it can easily be changed to accommodate functionality such as writing the output to a file or database.

DFA Comparator

The DFA comparator component requires at least two streams of input to function properly. It then inspects the streams of input and compares the DFAs against each others' structure, in the order that they arrive.

DFA Reverser

The DFA reverser components takes as input a list of DFAs, and outputs a list of NFAs. The DFAs are reversed structurally, resulting in a NFA.

NFA Determiniser

The NFA determiniser component accepts a list of NFAs, determinises each one, and outputs a list of DFAs.

4.1.4 FATKID Visual Editor

The FATKID Visual Editor provides a graphical, easy-to-use interface for building complex workflows and setting the properties of the various components. It uses the Netbeans Visual Library extensively for the drawing and connecting of components. A screenshot of the plugin can be seen in Figure 4.2.

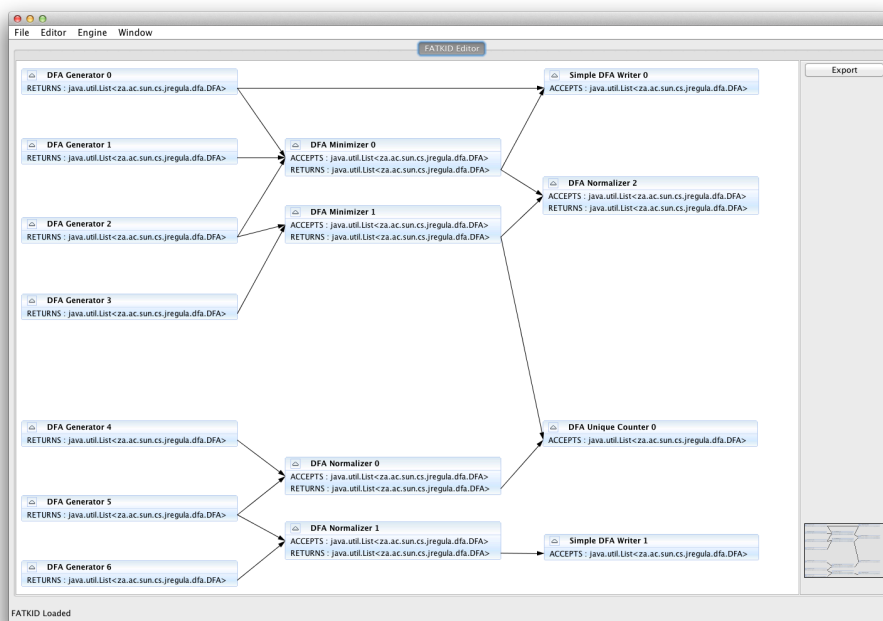


Figure 4.2: FATKID Visual Editor

During application startup, the FATKID Visual Library utilises the event bus to be notified of all the components that are available in the system. This allows any and all plugins to be automatically available in the editor. The user can then simply right-click anywhere on the editor canvas (with the exception of right-clicking on a node) to be shown a list of all the components. Selecting one will place the appropriate node on the canvas.

When right-clicking on a node, the user is presented with the choice of either editing the properties of the component, or deleting it. Selecting

properties presents a window where the user is able to change the values of the properties. The list of properties displayed in the properties editor dialog is dynamically generated at runtime. Deleting the node removes it, and all the connections to and from it, from the pipeline.

4.1.5 FATKID Engines

Workstation Engine

The FATKID Workstation engine is an engine plugin that executes the workflow on the local machine, without distributing the load. This is perfect for small workflows and shows off how the framework's modular design enables the re-use of components. Both the FATKID Workstation Engine and the FATKID Equal Distribution Engine (Section 4.1.5) makes use of the FATKID Execution Library (Section 4.1.6) to perform the core execution of the workflow.

Equal Distribution Engine

The FATKID Equal Distribution Engine is what formed the bulk of the work during plugin development. This module enables a user to execute the workflow in a distributed way, without requiring any additional knowledge of the system's workings. A screenshot of the interface can be seen in Figure 4.3. The distribution mechanism will be discussed in detail in Chapter 5.

4.1.6 FATKID Utility Libraries

Execution Library

The FATKID Execution Library is responsible for the execution of workflows, either in workstation mode, or in a distributed way. It will be discussed in detail in Chapter 5.

Remote Library

The FATKID Remote Library is a small library that abstracts the serialisation and de-serialisation of Java objects to create an easy framework for communication between the different nodes in a distributed system. It also provides a simple server class which can be used to easily create new worker nodes in the distributed network. As described during the discussion of the base FATKID module, these servers also load all libraries and components into their run-time classpaths. As such, the servers have access to all the components required to execute workflows remotely. It is up to the user to make sure that all the required components, libraries, and the correct versions of these, are available to the servers.

As will be discussed in Section 7.1, a future improvement to the system

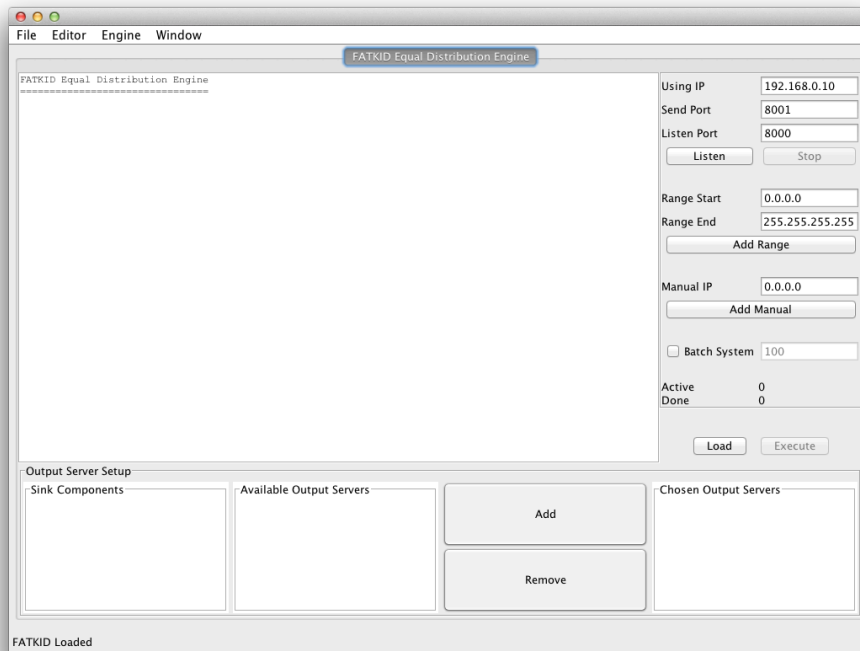


Figure 4.3: FATKID Equal Distribution Engine UI

is to have all libraries, plugins and components loaded from a central, remote location. JSPF supports the loading of plugins from remote locations using the Hypertext Transport Protocol (HTTP). This will give users the ability to maintain only a single, central repository of libraries, plugins and components, and all server nodes can automatically stay updated by accessing this central repository.

4.1.7 *jregula*

Description

jregula is a pure Java finite automaton library that we developed for the purpose of illustrating the working of the FATKID framework. It includes basic automata building and manipulation functionality. It uses the GNU Trove [29] collections library, which provides a way to use Java primitive data types in collection classes such as lists, maps and sets, rather than having to use the Java collection classes that wrap all primitive data into their object alternatives. This results in tremendous improvements in memory usage, and improves the execution time of all functions.

Features

The following features are currently included in *jregula* as we have deemed this enough to illustrate the functionality of the FATKID framework:

- DFA building using an abstraction layer,
- DFA minimisation,
- DFA normalisation,
- DFA reversal,
- NFA building using an abstraction layer, and
- NFA determinisation.

The GNU Trove Collections Framework

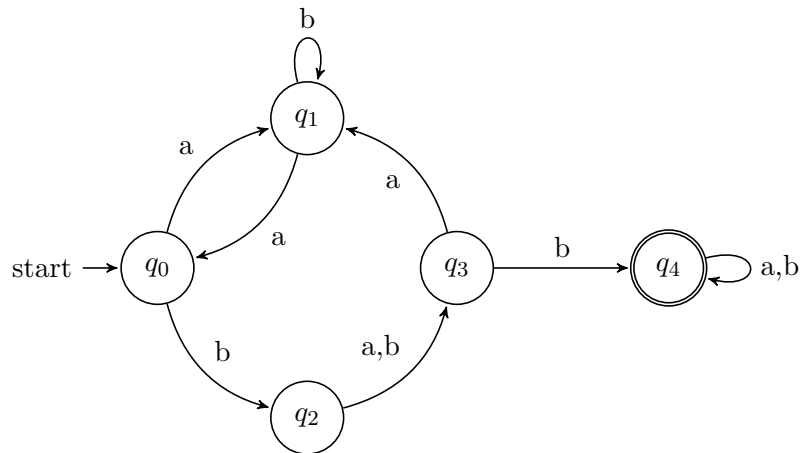
The GNU Trove collections framework provides a free, fast and lightweight implementation of the Java collections API [29]. It achieves its goals by using primitive data types wherever possible rather than wrapping primitives in objects. This typically leads to massive performance gains and lower memory usage.

Example Use

We will now show an example of how to use the DFA builder abstraction layer to build a DFA programmatically. Consider that we want to create the automata displayed in Figure 4.4. From the code example below, it should be clear that the API is very intuitive and easy to use. Also, from the definition of a DFA, each state must contain transitions on all alphabet symbols. With this in mind, it is not required to specify self-loops, as these are implied if not specified.

```
1      final DFABuilder builder = new DFABuilder();
2      builder.addStates(5);
3      builder.addSymbol("a");
4      builder.addSymbol("b");
5      builder.addTransition(0, "a", 1);
6      builder.addTransition(0, "b", 2);
7      builder.addTransition(1, "a", 0);
8      builder.addTransition(2, "a", 3);
9      builder.addTransition(2, "b", 3);
10     builder.addTransition(3, "a", 1);
11     builder.addTransition(3, "b", 4);
12     builder.setInitialState(0);
13     builder.addAcceptingState(4);
```

The resulting DFA is an immutable object. By design, no further modifications can be made to the internal structure of the DFA except by using Java reflection. The DFA requires an array of strings to be used as parameter when checking the acceptance of strings in its language. Each string element

Figure 4.4: *jregula* Example Automata

in the array represents the next symbol in the input string. The reasoning behind accepting an array, rather than just a string, is that this enables the library to do transitions on entire words, rather than single characters.

```

1     final DFA dfa = builder.compile();
2     assertEquals(dfa.acceptsInput(new String[] {
3         "a", "a", "b", "a", "b", "a"}), true);
4     assertEquals(dfa.acceptsInput(new String[] {
5         "a", "b", "b"}), false);
  
```

4.2 Extending the Framework

The ability to extend the framework as easily as possible was one of the main goals for the project. We will now continue to describe the process of developing an extension for the FATKID framework.

4.2.1 Creating a Plugin

The process of creating a plugin is as follows:

1. Create a new Java class.

2. Annotate the class with the `@PluginImplementation` annotation. This annotation is available from the FATKID library.
3. Make the class implement either `FatkidEditor` (to create an editor) or `FatkidEngine` (to create an engine).
4. Implement the necessary methods from the interface contract.
5. Compile the JAR. It is not required to include the FATKID library in the JAR, as it should reside in the `lib` folder of the application.
6. Place the JAR in the `plugins` folder of the application.

The FATKID application will then inspect the JAR during startup, load the classes annotated and make them available during the application's execution. The source code of the `FatkidPlugin` interface can be seen in Appendix A.5.1. As can be seen, there is only one method to implement. It should return a `JPanel` which the framework will display. From here the developer can include any functionality he requires.

4.2.2 Creating a Component

The process of creating a component is similar to that of creating a plugin. The only step that differs is rather than implementing the `FatkidEditor` or `FatkidEngine` interfaces, the developer should implement one of the component interfaces mentioned in Section 4.1.2.

4.3 Examples of Components

To illustrate the nature of components, we look at three examples.

4.3.1 Random Generator

The random generator is a prime example of a source component. It generates a stream (or whichever data type the developer defines) of automata

which can be used as inputs to either source-sink or sink components. To create this source, the `SourceComponent` interface needs to be implemented, and as the generic output type we define `List<DFA>`, which means that the result of the generator component's `execute` method would be a list of deterministic finite automata. The source code of our DFA generator can be seen in Appendix A.2.

4.3.2 Minimisation Filter

The minimisation filter is an example of a source-sink component. It requires that one or more data streams (from either source components or source-sink components) be connected to it. It will then minimise the automata and can provide the result as input for other components.

Source-sink components have to define input types and output types. For our minimisation component implementation, we define both the input and output types to be of type `List<DFA>`. Therefore, we output the same type of data stream as the input. This does not need to be the case though. It should be clear that a source-sink component is perfect for writing a component that transforms one data stream into another. A great example of a *transformer* type component like this is when using more than one automata implementation, and it is required to convert between implementations.

The source code for our DFA minimiser can be seen in Appendix A.1.

4.3.3 Unique Counter

As its name suggests, the Unique Counter Filter counts the number of unique instances of automata. There are various issues involved in identifying when two automata are the same. At one end of the spectrum, identity could be based on the languages that the automata recognise, while at the other end identity could be defined as purely structural. Of course, users are free to extend the tool to incorporate these extremes or anything in between, but

our default implementation is purely structural and accepts only deterministic finite automata. This is simple to implement efficiently, and, fitting with our simplicity-principle, it is easy to implement language equivalence through the use of other filters that, for example, determine, minimise, and canonise nondeterministic finite automata.

4.4 Filter Composition

Our first approach for defining the data streams between components was to define a very generic automaton interface, and all communication had to be done via this. It quickly became evident to us that this was not an optimal approach, as components might need to communicate other types of data, such as raw numbers, boolean values, etc.

Our current approach allows for much more flexibility in the types of data streams that components use to communicate. Every component now declares its own communication class during development. We then inspect this with reflection at runtime and make sure that only components that can actually talk to each other are allowed to do so. Even though this seems like a restriction, with the current framework, it is extremely easy to write a component to convert data streams. Each component that a developer writes for the framework has methods that need to be implemented, as put forth by the three interface contracts for components. We discuss the three most important ones relating to filter composition:

4.4.1 Merge

The first of the three methods we discuss is relevant to all source-sink and sink components. The purpose of this method is to define the behaviour of the component when it receives data from multiple, or a single sources.

```
public A merge(A[] inputs)
```

where *A* is the input stream object type. As can be seen from the method call, the framework will inject an array of the object type, and it is then up to the developer of the component to determine how to handle multiple streams. Each element in the input array is a single stream from a different component. Even when there is only a single input stream, the method is invoked; in this case, the component can simply return the first element of the 1-element array. The output generated from the merge function is the same as what the framework will pass as parameter to the execute function.

4.4.2 Split

As with the merge function, the split function is applicable to two types of components, namely source-sink components and source components. The purpose of this method is to dictate how the result of execution should be split amongst all the components receiving their input from the executing component.

```
public R[] split(R r, int n)
```

where *R* is the result (output) object stream type and *n* the number of streams into which it should be split. We predict that the most common case is that the implementation of this method will simply duplicate a single stream into multiple streams, so that each component receives the same information. However, the framework allows the developer to split the streams in any way to fit his implementation needs. This may be useful, for example, when a component wishes to divide its output between two components, rather than send duplicate streams.

The output generated by the split method is what the framework will inject into the components connected to the executing component (the components receiving their input from the executing component).

4.4.3 Execute

The execute method is where the actual work of the component happens. This method contains the logic of the developer's intent for the component. For each of the three types of components, the method signature differs slightly to accommodate the workflow.

```
public R execute()
```

The definition for source nodes specifies that the execution method does not require any parameters. This is correct, as source components only expose methods for generating output and cannot receive input from any other components.

```
public R execute(A a)
```

For source-sink components, we need to allow for both input and output to come into play. The method provides a mechanism via which input streams are injected by the framework. The input parameter for this execute method is never directly called. It is always the result of calling the merge method discussed previously.

```
public void execute(A a)
```

Lastly, for sink components, only input streams are required. Sink components are considered as the “end-of-execution” of a certain pipeline path, and as such never produce output for other components. Therefore, any results that need to be processed (for example writing them to disk) need to be dealt with inside the execute method. As with the execute method for source-sink components, the input is the result of the merge method.

In the case where the distributed engine is used, you can define output servers for sink components. This allows sink components from all the worker nodes, rather than executing individually in isolation on the local

machine, execute on a single node. This is useful for components such as the unique counter, where the total number of unique automata in a workflow needs to be counted, a task impossible to do in isolation.

4.4.4 Batch Mode

On the issue of performance, an important feature of the system is the ability to run in batch mode. The user can specify a batch size which will be communicated to the execution engine. The system will then not execute components completely. Rather, it will instruct source components to only produce a specific amount of data, which will then be passed through the pipeline. The system keeps track of the amount that each source component has generated, and the total number that each component should generate. It also analyses each component after execution to determine if all its sources have finished their generation, at which point it will be removed from the execution pipeline.

This batching feature enables the system to process millions of automata without using a large amount of memory. We have identified that a batch size of roughly 100 automata results in very good performance in general. This can be seen in the results we obtained, which we present in Section 6.3.9.

Chapter 5

Concurrent Framework

Our current framework implements a fair and equal distribution algorithm. At the most abstract level, it consists of two steps. Firstly, the work needs to be distributed across all the execution nodes, and secondly, the execution happening on each node also needs to run tasks in parallel without affecting the workflow. The source code that generates this pipeline of work that can be executed in parallel on each node can be seen in Section A.4. The first step of the process can be broken down into the following simple steps:

1. Search for all available nodes in the grid which can function as a worker node.
2. Load a saved workflow into memory and perform the initial analysis of it.
3. The user then optionally selects output servers for the various sink nodes in the workflow.
4. Upon starting the execution, the engine distributes the workflows to the various engine nodes.
5. The engine then waits for the nodes to finish their computation and report back to it.

During step 4, when the workflow is distributed to the various nodes, the execution engine takes over and performs calculations in isolation (except if output must be sent to other servers). The following steps take place during execution:

1. The workflow is received from the engine.
2. The workflow is deserialized back into object form and analysed for integrity.
3. A pipeline and parallel execution pools¹ are generated from the workflow.
4. The pipeline is executed (during which output is sent to external servers if specified).
5. When all work is complete, the node reports back to the engine to notify it that all work is completed.

We now discuss each of the above mentioned steps in greater detail.

5.1 Distribution System

5.1.1 Search for Available Engine Nodes

The first step in executing a distributed workflow is for the system to determine the number of worker nodes that are available to help with the execution. This is done by either specifying IPs manually, or giving a range of IPs that the application will query. The master node then iterates over these IPs and tries to open a connection to each one.

If the system cannot establish a connection in 100ms, it continues to the next IP. If a connection is successfully established, it sends a specifically

¹The term *parallel execution pool* refers to a collection, or pool, of components that can be executed simultaneously in parallel without affecting each other.

formatted object stream to the node. It is then up to the node to interpret, and if successful, respond to the server and identify itself as a worker node. The asynchronous nature of the server prevents it from blocking on each node and waiting for a response.

It is important to note that the master node does not function as a worker node, but by using the port settings correctly, it is possible to have a worker node on the same machine that the master node is running on. Once the server has iterated over all the IPs specified, and the user is happy that all the worker nodes he requires are in the active node list, he can continue to the next step, which is loading a workflow.

5.1.2 Loading and Pre-analysis of the Workflow

After at least one active worker node has been registered, the user is able to load an existing workflow into the system. A very basic pre-analysis of the workflow is done, which basically just checks that the input is correct and was produced with the same version of the libraries that is currently loaded.

5.1.3 Selecting Output Servers

Once the workflow has been successfully loaded into the system, the user can start selecting output servers. The requirement of being able to select output servers arose from the problem that not all operations can be distributed. For example, consider the scenario where a user constructs a workflow in which he counts the number of unique automata that are generated. If this were to execute in isolation, it could very well be that equivalent automata are generated on separate worker nodes. For this case, we implemented functionality through which output from components can be sent to specific worker nodes and executed remotely. This makes it possible to use components such as unique counters that count the number of unique automata in the entire workflow.

Any number of output servers can be assigned to each sink component. During execution, if a sink component has output servers specified, it will not execute in its normal fashion. Instead, it will be asked to serialise any output that it might generate. This output is then sent to all the output servers associated with the sink component. On the receiving end, the data will be assigned to a similar class component, and be executed.

This provides an easy mechanism to implement situations that are otherwise difficult in a distributed environment, such as calculating the number of unique automata, since this needs to be done centrally. If no output server is specified (as when the system is running in workstation mode), the component will simply do its calculation and produce output locally (i.e., on the node itself), without sending the output to any other node.

5.1.4 Distribution of Workflow to Nodes

Once the workflow is successfully loaded and the output servers selected, the system can start the execution process by sending the workflow to all the nodes, accompanied by some meta-information that is required by the nodes to determine their own behaviour. The meta-information includes details such as the total number of nodes in the distributed system, the address (and port number) to which the nodes must report when their execution is complete, whether the execution engine should run in batched mode or not, and if so, the size of the batches.

5.1.5 Wait for Engine Nodes to Complete

Once the system has received workflow completion responses from all the worker nodes, the execution is complete. Information such as the total execution time, and execution time of each individual node is displayed in the application.

5.2 Concurrent Execution Framework

5.2.1 Receive Workflow from Master

Each node in the distributed system that the user wants to use as a worker node must be running a small server program that waits and accepts incoming requests from either the master, or other server nodes.

Upon the receipt of a workflow execution request, the server passes the serialised form of the workflow unto the execution engine, which knows how to de-serialise and reconstruct it.

5.2.2 Analysis and Reconstruction of Workflow

Once the execution engine receives a serialised workflow from the server on which it is running, it will first do another analysis of it (similar to that done upon loading). Once successful, the execution engine will start reconstructing the workflow from its serialised form into a structure consisting of the real component objects and the relationships between them.

During construction, it also finds all the source components so that it can quickly determine where the execution pipeline starts. During this same process, it injects the new number of items that the component is required to source (which in our current equal distribution scenario is simply the required number divided by the number of worker nodes in the system). When a sink component is constructed, the engine will also look to see if any output servers are defined for that specific component and set them on the instantiated component.

5.2.3 Generation of Parallel Execution Pools and Pipeline

The next step before the engine starts its execution of the workflow is to determine the parallel execution pools and pipeline order. The engine analyses the reconstructed workflow and determines which groups of components can

be executed in parallel without influencing each other. This is done by doing an adaptation of the breadth-first-search algorithm, branching from all the source nodes simultaneously (since no source node can rely on another).

Each of these parallel execution pools are then placed into the execution pipeline in the order in which they are discovered, which maintains referential integrity.

The pseudo-code for this process is shown below. The actual source code can be seen in Section A.4. The C parameter is the collection of components loaded from the workflow, while EP is a reference to the execution pipeline that will be filled by the algorithm. N is a placeholder set to keep track of components that should be considered as potential candidates for execution during the next iteration. P is a map that keeps track of the number of times a non-source component has been referenced by a component that provides it with input. The notation $P[c]$ is used to indicate that we are referencing the value stored in the map for the key c . For any component c , we use two values during the algorithm. $c_{num_sources}$ is an integer that indicates the total number of sources that provide input to the component. In other words, it is the number of input streams connecting to the component. c_{sinks} is the set of sink components that the component provides with output data.

After completion of the algorithm, all variables local to the algorithm are lost. The reference to the execution pipeline, EP , now contains a list of execution pools, in the correct order in which they should be executed.

5.2.4 Execution of the Pipeline

After the parallel pools and execution order have been defined, the engine can start executing the workflow. The execution process on a high level can be defined as follows:

1. Instantiate an executor service
2. For each parallel execution pool in the pipeline

Algorithm 2 Parallel Execution Pipeline Generation

```

1: procedure GENERATEPARALLELEXECUTIONPIPELINE( $C, EP$ )
2:   def  $N \leftarrow \emptyset$  ▷ Next Execution Set
3:   def  $P \leftarrow \emptyset$  ▷ Processed Nodes
4:   for component  $c \in C$  do
5:     if  $c \in SourceComponent$  then
6:        $N \leftarrow N \cup c$ 
7:     end if
8:   end for
9:   if  $N \equiv \emptyset$  then return false
10:  end if
11:  while  $N \neq \emptyset$  do
12:    def  $PEP \leftarrow \emptyset$  ▷ Parallel Execution Pool
13:    def  $CES \leftarrow N$  ▷ Current Execution Set
14:     $N \leftarrow \emptyset$ 
15:    for component  $c \in CES$  do
16:      def  $ref \leftarrow P[c]$  ▷ or 0 if not exists
17:      if  $ref = c_{num\_sources}$  then
18:         $PEP \leftarrow PEP \cup c$ 
19:        for component  $s \in c_{sinks}$  do
20:           $N \leftarrow N \cup s$ 
21:           $P[s] \leftarrow P[s] + 1$ 
22:        end for
23:      else
24:         $N \leftarrow N \cup c$ 
25:      end if
26:    end for
27:     $EP \leftarrow EP \cup PEP$ 
28:  end while
29: end procedure

```

- (a) Remove the pool from the pipeline
 - (b) For each component in the pool
 - i. Determine if the component will need to execute again after this execution. This is done by inspecting all the sources connecting to this component to see if they have completed. In the case of a source node, we simply need to check if it has sourced the required number of items
 - ii. If the component must execute again, add the component to a new pool which will be placed into a new pipeline
 - iii. Load the component into the executor service for execution
 - (c) If the parallel execution pool into which components that have not yet completed have been placed is not empty, place the new pool into the new execution pipeline
3. If the new execution pipeline is not empty, swap it with the pipeline that have just finished and repeat from step 2

In step 1, the executor service that is instantiated is a normal Java `ExecutorService`. This class provides a mechanism with which an arbitrary amount of threads can be executed in parallel, but only a specific number of them at a time. The current version of the execution engine will never execute more threads simultaneously than the number of available processors of the physical hardware (or processors exposed to the Java Virtual Machine instance). In upcoming versions of FATKID, the number of threads will be configurable, giving the user more power to fine-tune the system.

The executor service keeps an internal queue of processes that need to execute. We can therefore add any number of processes to this queue. The service creates the number of threads specified, runs them in parallel and re-uses them as processes finish to execute others.

5.2.5 Reporting Back to the Master

Once the entire execution pipeline has completed and reporting server information is present in the meta-information provided in the workflow, the execution server will send a message to the specified server reporting that it has finished its execution and how long the execution took in milliseconds.

5.3 Example 1

In this section, we will show how the concurrent execution pipeline behaves

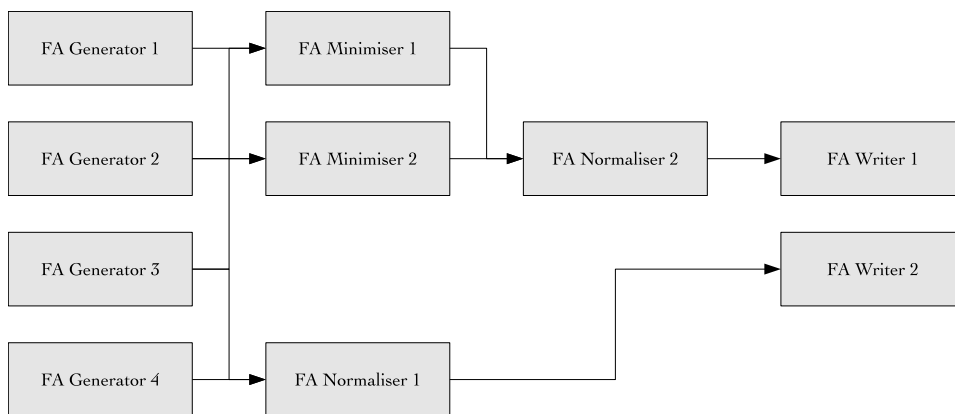


Figure 5.1: Example 1 – Workflow Overview

The example workflow can be seen in the Figure 5.1. It consists of four finite automata generators, two minimisers, two normalisers and two writers. Please note that generator here is analogous to a source, minimisers and normalisers to sink-sources and the writers to sink components.

After the framework has reconstructed the workflow into its internal storage format (Java objects), the first step is to divide the workflow into groups that can execute in parallel (parallel pools). It starts, by iterating over all the components in the workflow and identifying the source components. This can be seen in Figure 5.2

Since per definition none of the sources can receive any input, the collec-

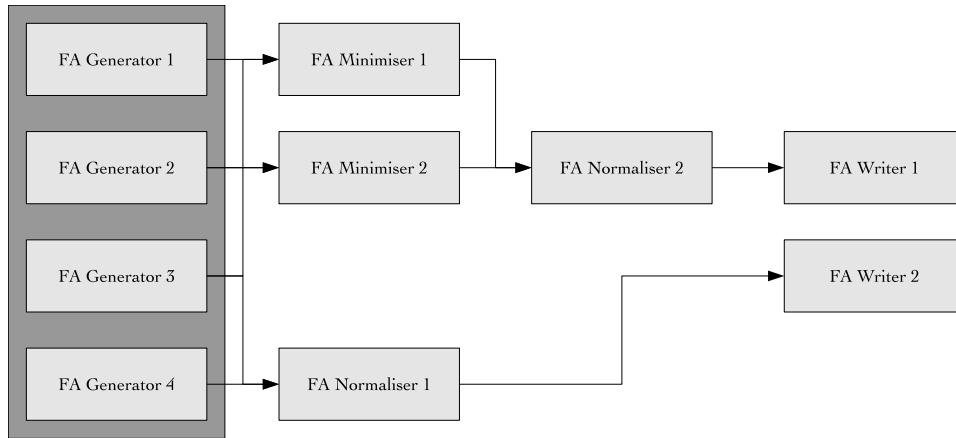


Figure 5.2: Example 1 – Analysis Step 1, Source Components

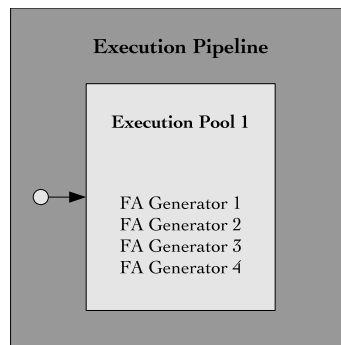


Figure 5.3: Example 1 – Execution Pipeline After Step 1

tion of sources form the first parallel pool. The framework then proceeds to look at all the components that receive input from the sources. It does this by inspecting the workflow and traversing in a breadth-first fashion from the sources. It might not be immediately apparent, but there is a second level of investigation that needs to take place once all of these components have been identified.

The framework keeps an internal registry of how many times a component has been identified as being next in line for processing. This is important because of the complex manner in which workflows can be constructed. Because any component that produces output can send its data to any number of components that can successfully accept the data, and also, any component that accepts data can receive data from an arbitrary number of sources, a component should only be allowed to execute once it has received input from all its sources.

Therefore, if a component is identified for execution, the internal registry is updated to add the new source. If the registry value for the component is not the same as the number of sources connecting to it, it is not placed in the execution pool. An example of this situation is described below in the second example.

In Figure 5.4 we can see the second set of components identified. Since all of them receive input from all their sources, they are all places into the next execution pool (Figure 5.5).

Now, following the same process as in the previous step, the next set of components (Figure 5.6) are identified, and since all of them have received all their input, they are placed into the next execution pipeline (Figure 5.7).

In the last step, only one component is left and by the same process as described previously, it is placed into the next execution pool (Figure 5.9). The framework then establishes that all components have been processed, and as such the execution pipeline is finished constructing.

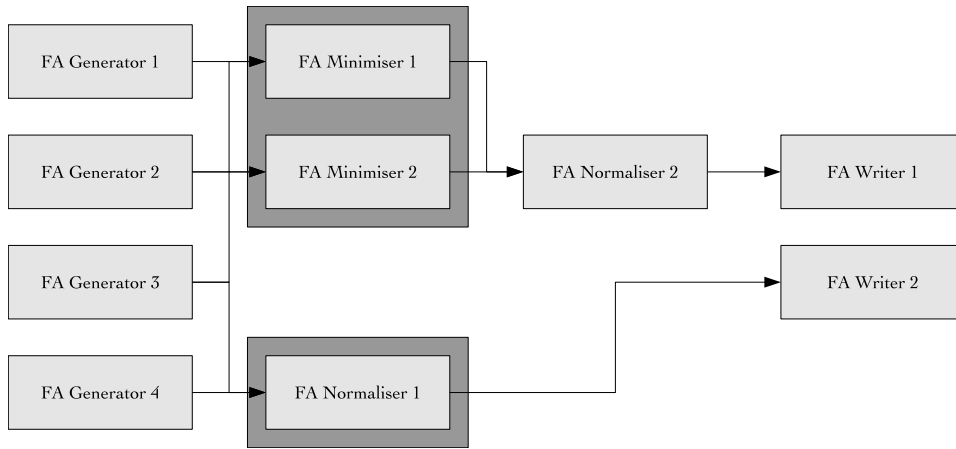


Figure 5.4: Example 1 – Identifying Second Execution Pool

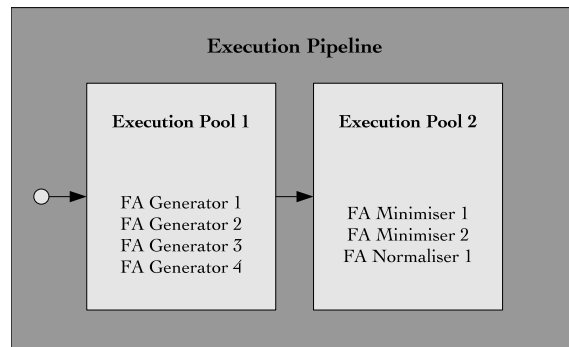


Figure 5.5: Example 1 – Execution Pipeline After Step 2

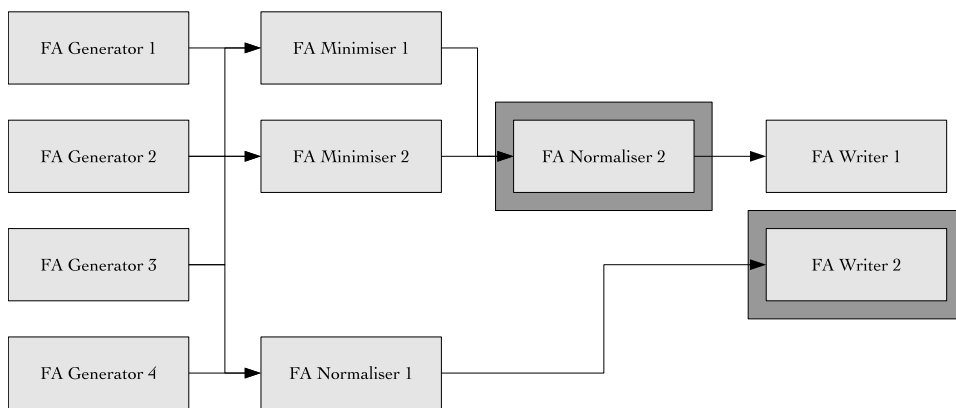


Figure 5.6: Example 1 – Identifying Third Execution Pool

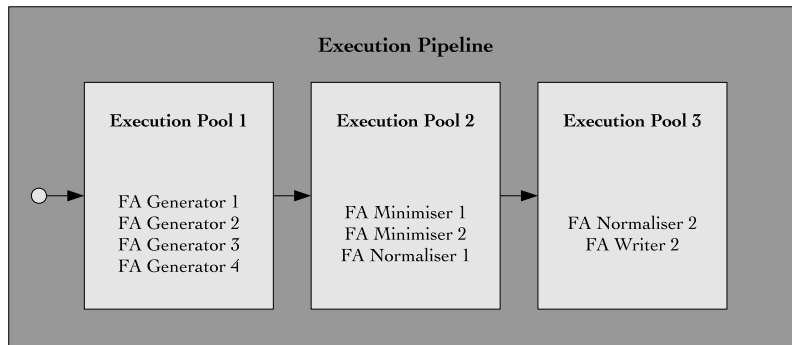


Figure 5.7: Example 1 – Execution Pipeline After Step 3

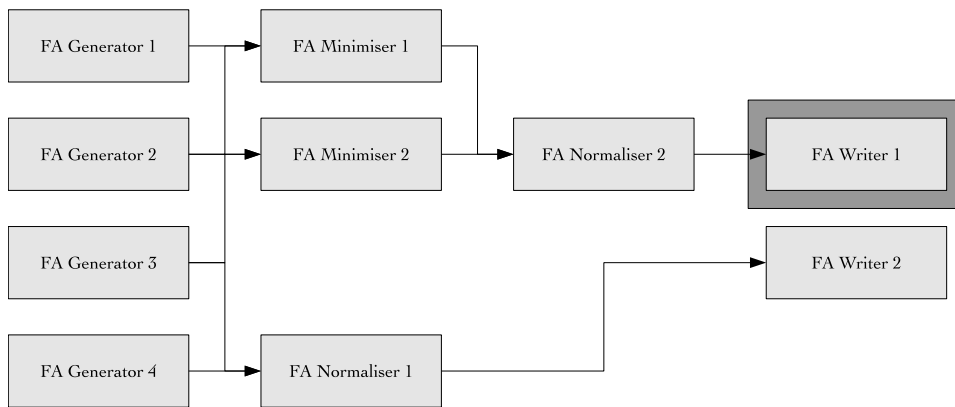


Figure 5.8: Example 1 – Identifying Final Execution Pool

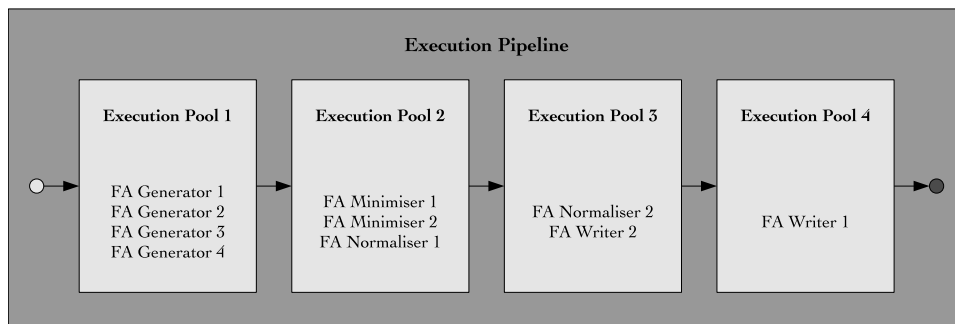


Figure 5.9: Example 1 – Completed Execution Pipeline

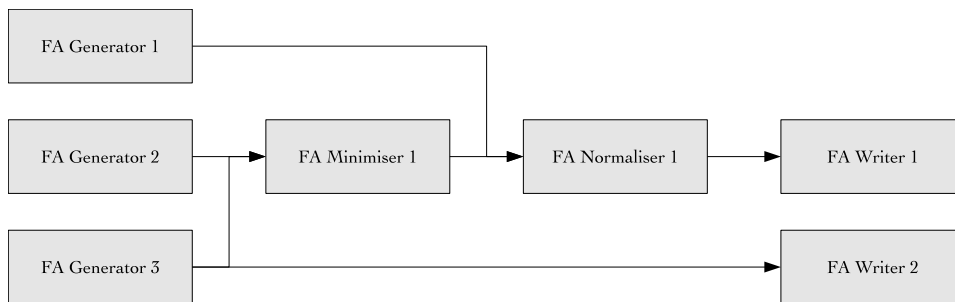


Figure 5.10: Example 2 Workflow Overview

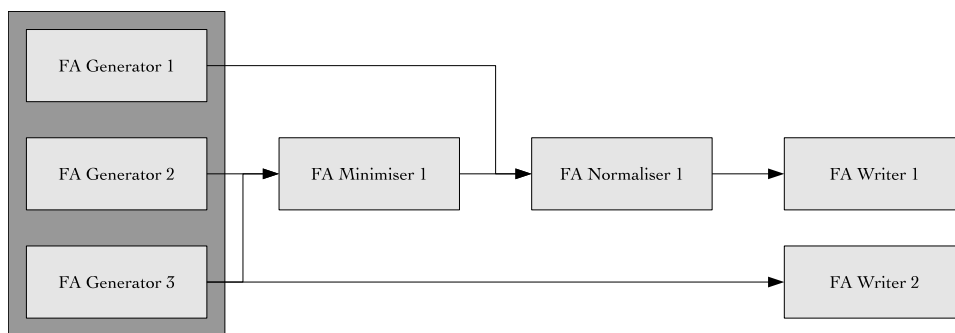


Figure 5.11: Example 2 – Identifying Source Components

5.4 Example 2

In the following example, we will show how when a component has not received all of its input, it is not placed into the current execution pool. The example (Figure 5.10) is a slightly modified version of the workflow in the previous example. We omit the figures displaying the execution pools, as was shown in the previous example, as these can easily be deduced from the workflow progression.

As mentioned before, the first step again involves identifying all the pure source components and placing them into the first execution pool.

The framework then traverses from the source components to identify the next candidates for execution. As can be seen, the *FA Normaliser 1* component is found. However, since this is the first time it is referenced,

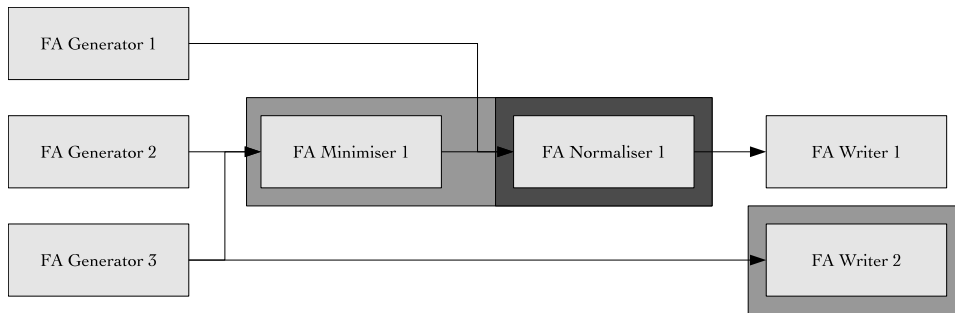


Figure 5.12: Example 2 – Second Iteration, Excluding Component

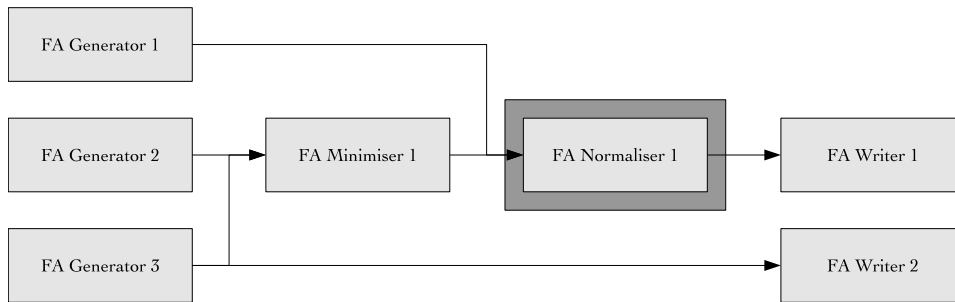


Figure 5.13: Example 2 – Third Iteration

the reference registry will have a value of 1 while the number of sources that connect to it, is 2. Therefore it is not placed into the execution pool.

In the next iteration, the *FA Normaliser 1* component is identified again (from *FA Minimiser 1*). The reference registry and number of connecting sources now correspond, and as such the component is placed into the execution pool.

Finally, as previously, the last component fills the final execution pool.

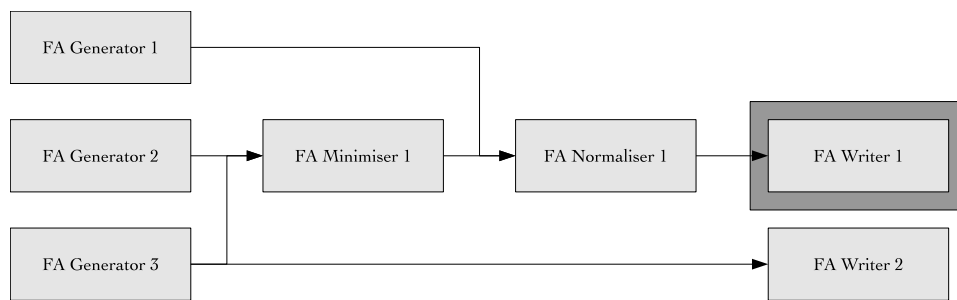


Figure 5.14: Example 2 – Identifying Final Execution Pool

Chapter 6

Evaluation

6.1 User-friendliness and Accessibility

User-friendliness and accessibility are probably the two most difficult aspects of the framework to quantify and evaluate. What one person finds user-friendly, another finds difficult to use. We therefore merely state we think that what we have produced certainly provides the user with an interface that is much easier to use than some of the other frameworks that are only command-line utilities.

With FATKID, the user is presented with a graphical user interface, where workflows can be created by adding components to the screen and manipulating them visually. It is our opinion that this simple fact already makes FATKID adhere to our design goal of user-friendliness.

As far as accessibility is concerned, the project is open-sourced, and as can be seen from the FATKID library code in Appendix A, it is very well documented and clearly explains the purpose of each method and class. The project source can be found at <http://www.cs.sun.ac.za/~fatkid>.

6.2 Extensibility

As mentioned previously, extensibility has been one of our main focus points from the inception of the project. As described in Chapter 4, the process of creating a plugin is extremely easy. Furthermore, detailed descriptions of each method that needs to be implemented can be found in the javadocs of the FATKID Library, some of which can be seen in Appendix A. We therefore feel that the goal of extensibility has been met.

As an example, consider the “DFA Unique Counter” component. After implementing the `SinkComponent` interface, the entire source code, with formatting, is approximately 140 lines of code, and took roughly 30 minutes to write. Since this is a plain Java class, it can be easily unit tested with JUnit (or whichever testing framework developers prefer). The source code can be found in Appendix A.3.

6.3 Performance

Performance is by far the easiest design goal to quantify. It is also the main design goal that we set out for ourselves. We now present the performance data of the various experiments that we ran.

6.3.1 Experiment Setup

All of the automata generated by the experiments consist of 20 states, 5 alphabet symbols, 4 accepting states, and a 25% probability of transitioning to any state other than itself, resulting in an average of 5 transitions per state. The destination state for a transition is selected uniformly from the other 19 states. From the definition of a DFA, each state must contain transitions for each alphabet symbol [1]. In the *jregula* library, all symbols for which no transition is specified, implicitly results in a self-loop transition for DFAs. This is not the case for NFAs as the definition of required transitions

for each symbol on each state does not hold for NFAs.

Table 6.1 shows the number of automata generated by *each* generator present in the respective workflows. All experiments were run using batch mode with a batch size of 100.

Experiment	Number of automata		
	Small	Medium	Large
Experiment 1	100000	500000	1000000
Experiment 2	100000	500000	1000000
Experiment 3	100000	500000	1000000
Experiment 4	500	2000	5000
Experiment 5	50000	100000	200000

Table 6.1: Experiment Generator Sizes

6.3.2 Experiment 1

The first experiment is a very simple workflow that consists of a single DFA generator that sends its output to a DFA unique counter component that counts the number of unique automata that are generated. The workflow can be seen in Figure 6.1, while the results can be seen in Table 6.2 and Figure 6.2.



Figure 6.1: Experiment 1

Size	Time in milliseconds			
	Number of nodes			
	2	4	6	8
Small	1546	975	771	605
Medium	6349	3372	2454	1907
Large	12276	6532	4514	3490

Table 6.2: Experiment 1 – Average Results

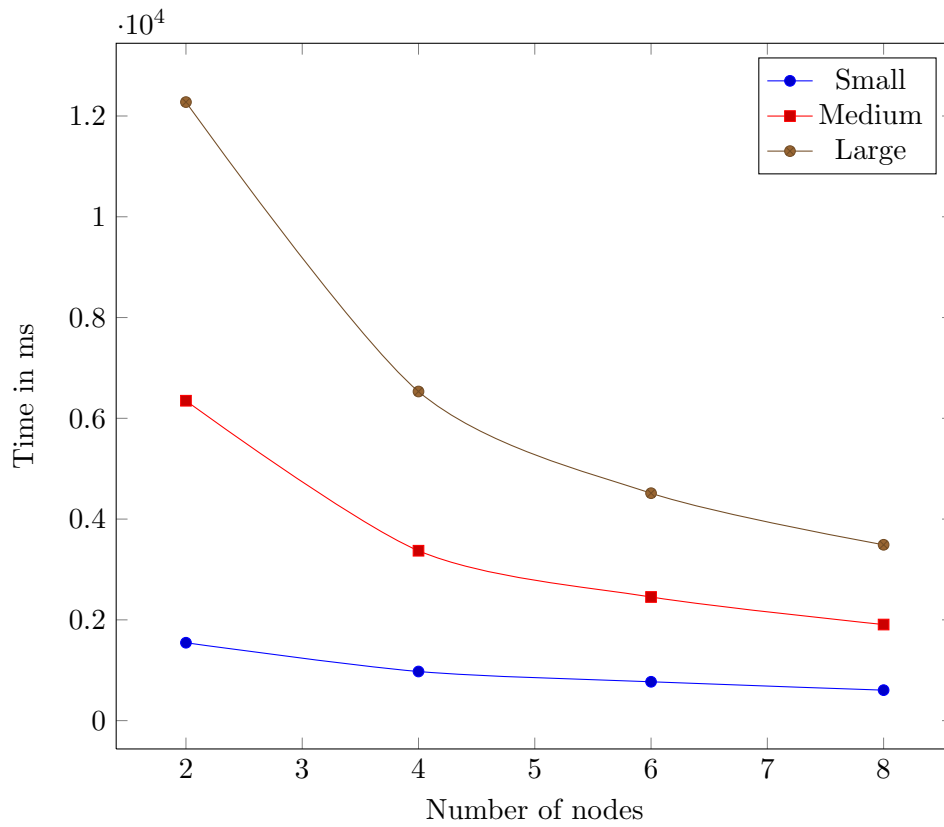


Figure 6.2: Experiment 1 – Results Comparison Chart

6.3.3 Experiment 2

The second experiment is a bit more complex than the first and involves four steps in the workflow. A generator sends its output to a minimiser, which in turn sends its output to a normaliser, and then to a unique counter, which displays the number of unique DFAs that pass through it. The workflow can be seen visually in Figure 6.3.

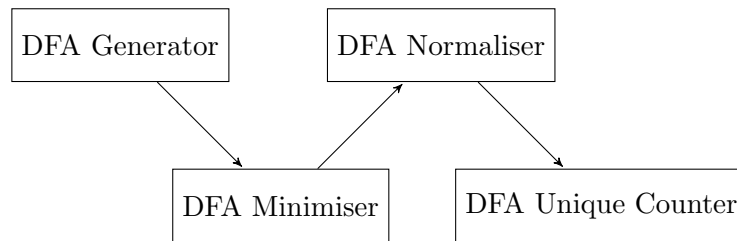


Figure 6.3: Experiment 2

Size	Time in milliseconds			
	Number of nodes			
	2	4	6	8
Small	9850	4953	3333	2520
Medium	48980	24579	16395	12302
Large	97847	49104	32712	24564

Table 6.3: Experiment 2 – Average Results

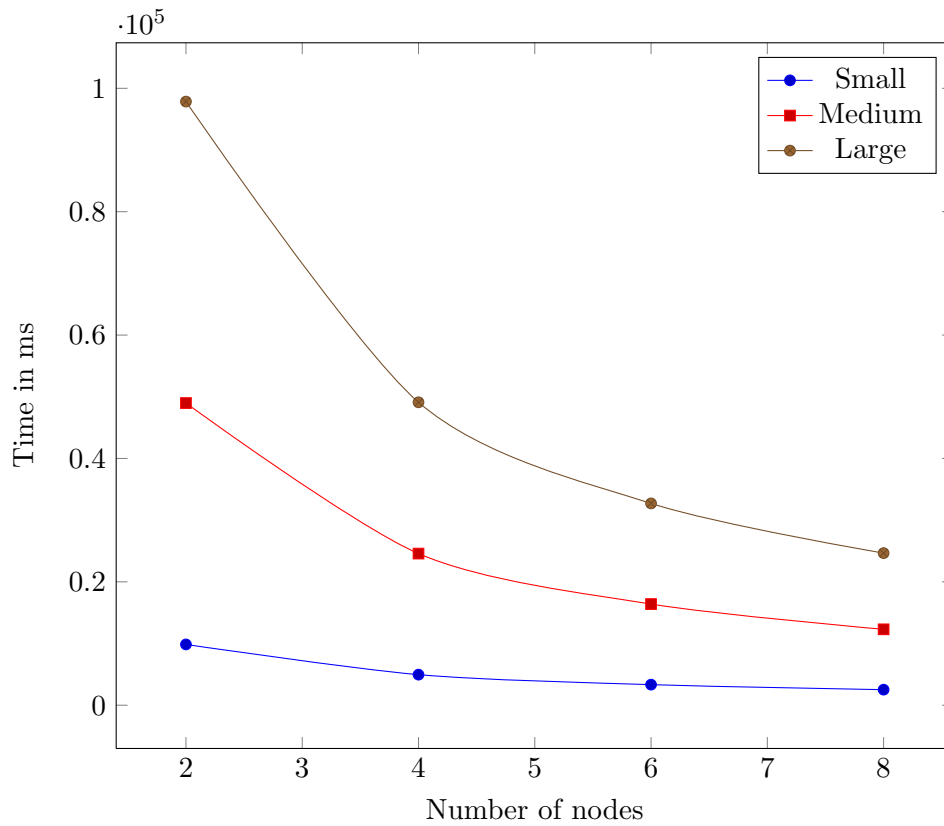


Figure 6.4: Experiment 2 – Results Comparison Chart

6.3.4 Experiment 3

In the third experiment we increase the complexity of the workflow even more. A DFA generator sends its output to two minimisers, the one using the Hopcroft algorithm to minimise the automata, while the other uses the Brzozowski algorithm. Both these minimisers then send their output to normalisers so that the automata are in their canonical form. The two streams are then compared with a comparator component.

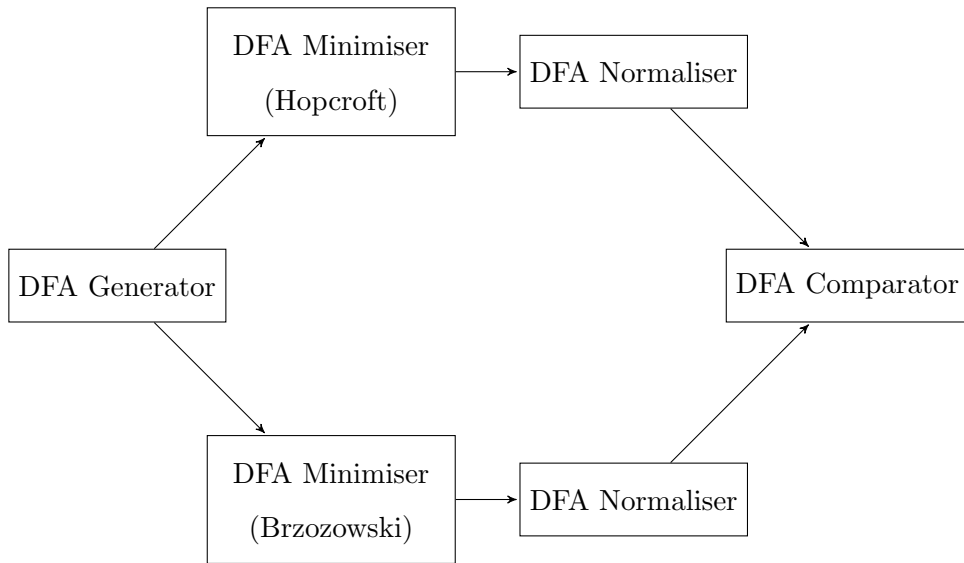


Figure 6.5: Experiment 3

Size	Time in milliseconds			
	Number of nodes			
	2	4	6	8
Small	9766	5004	3391	2562
Medium	48666	24676	16601	12414
Large	96857	49287	33116	24698

Table 6.4: Experiment 3 – Average Results

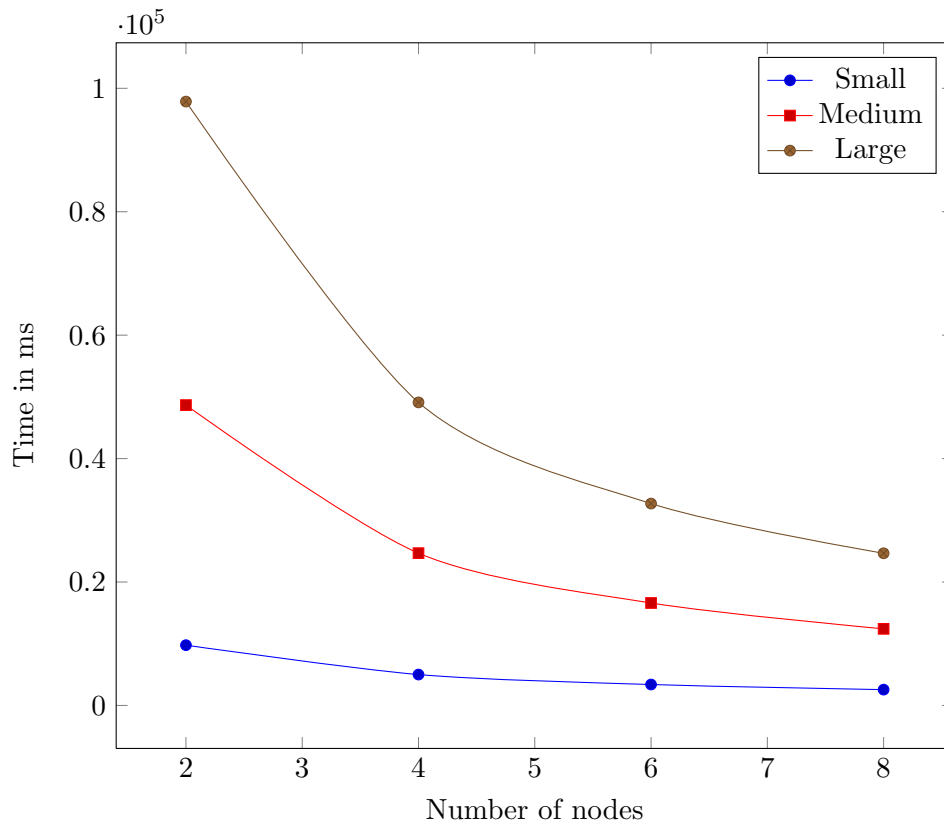


Figure 6.6: Experiment 3 – Results Comparison Chart

6.3.5 Experiment 4

In experiment 4 we can see how the Brzowski algorithm works. The generator sends its output to two streams. The first uses the singular minimiser component, using the Brzowski algorithm. The other stream has the individual components that make up the Brzowski algorithm. First, the automata are reversed, then determinised, reversed again and finally determinised again. A comparator component then compares the two streams and displays if indeed the two processes produce the same result. The workflow can be seen in Figure 6.7, while the results can be seen in Table 6.5 and Figure 6.8.

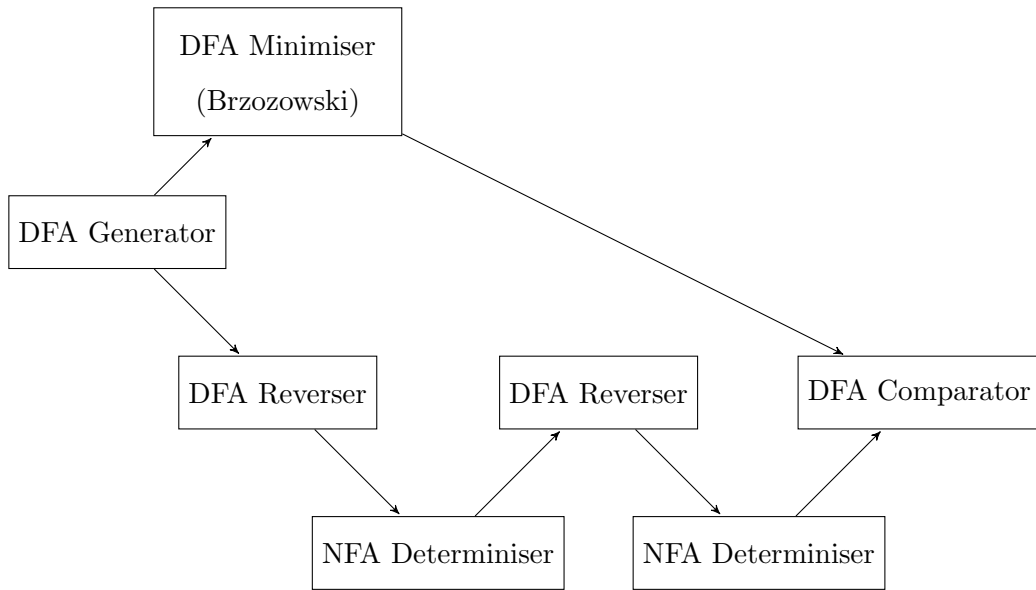


Figure 6.7: Experiment 4

Size	Time in milliseconds			
	Number of nodes			
	2	4	6	8
Small	12261	6907	5644	3647
Medium	47312	26848	21606	14249
Large	111116	63714	43568	34429

Table 6.5: Experiment 4 – Average Results

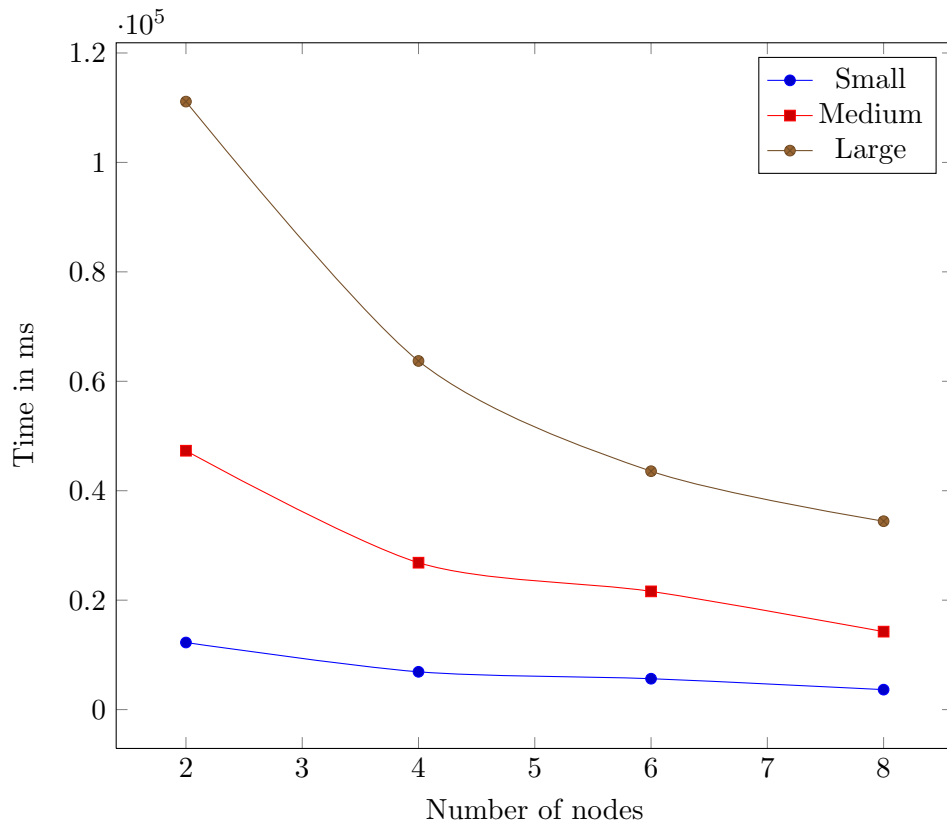


Figure 6.8: Experiment 4 – Results Comparison Chart

6.3.6 Experiment 5

In the last experiment that we present in this thesis, we wanted to highlight the splitting and merging (discussed in Section 4.4) of data streams. The workflow can be seen in Figure 6.9.

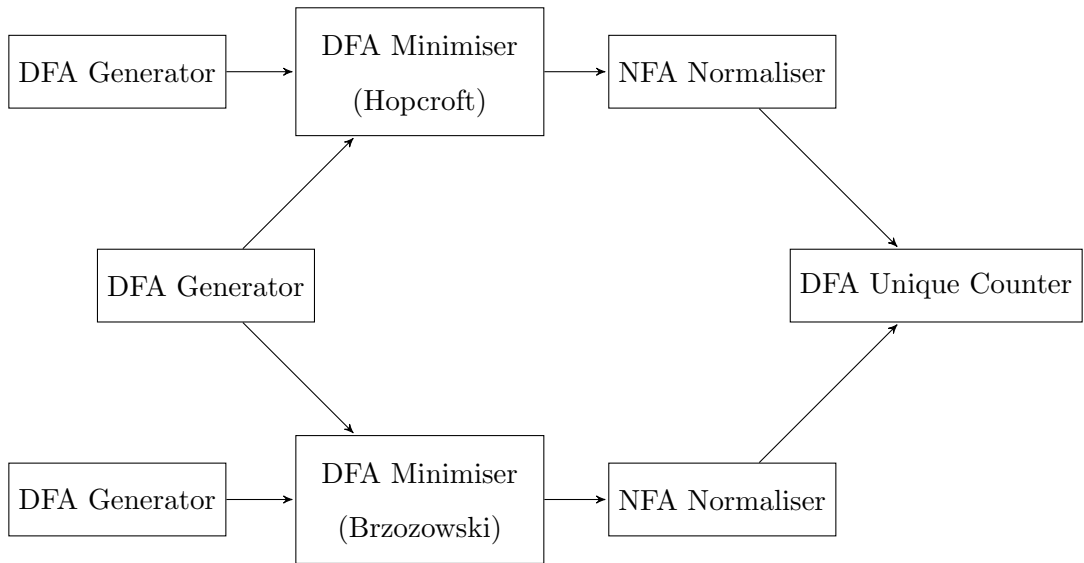


Figure 6.9: Experiment 5

Size	Time in milliseconds			
	Number of nodes			
	2	4	6	8
Small	10758	5379	3625	2780
Medium	21394	10739	7164	5422
Large	42753	21384	14249	10761

Table 6.6: Experiment 5 – Average Results

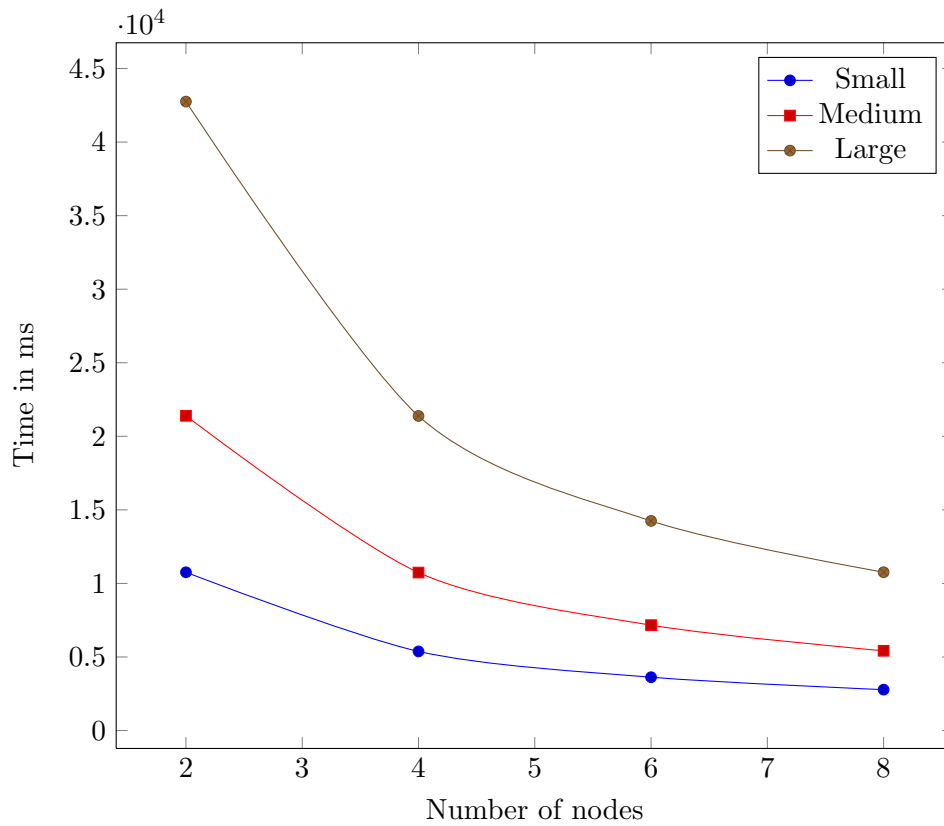


Figure 6.10: Experiment 5 – Results Comparison Chart

6.3.7 Linear Performance and Speedup

We continue the discussion of the results by looking at a property called *speedup*. In parallel computing, the term speedup refers to how much faster an algorithm executes in a parallel environment, when compared to running it in a sequential environment [31]. The formula for speedup is defined by:

$$S_p = \frac{T_1}{T_p} \quad (6.1)$$

where p is the number of processors, T_1 the execution time of the sequential algorithm, and T_p the execution time of the parallel algorithm with p processors. In the case of perfect linear speedup, or *ideal speedup*, $S_p = p$.

To measure the speedup obtained by FATKID, we ran experiment 3, with a medium size, on a single node. The average execution time was 98433 milliseconds. Using the formula for speedup, we calculate the following:

$$S_p = \frac{T_1}{T_p} = \frac{98433}{12414} = 7.9292 \quad (6.2)$$

which leads us to conclude that a speedup of 7.9292 times is obtained when executing the workflow on 8 nodes. Another important measure in parallel computing is *efficiency*. This estimates how well each extra processor is utilised, compared to the effort that is wasted in communication and synchronisation. The formula for calculating the efficiency is

$$E_p = \frac{S_p}{p} = \frac{T_1}{pT_p} \quad (6.3)$$

and by using the values we obtained from the experiments, we calculate an efficiency coefficient of:

$$E_p = \frac{S_p}{p} = \frac{7.9292}{8} = 0.9911 \quad (6.4)$$

A perfectly linear speedup has an efficiency value of 1, or in other words, each extra processor utilises 100% of the processing power it has. Acquiring a value of 0.9911 means that each additional node added to the distributed environment effectively increases the processing power by 0.9911, or, utilises 99.11% of its processing power. This indicates that the speedup we achieve with FATKID is almost perfectly linear.

If an efficiency value of greater than 1 was achieved, it would indicate that for each processor added to the distribution environment, you effectively gain more than the processing power that processor provides. This is referred to as *super linear speedup*. It might seem strange at first since some people believe that the theoretical maximum should be an efficiency value of 1, but this behaviour can be explained.

One of the possible reasons for super linear speedup, called *the cache effect*, is discussed in [31]. Each processor added to the environment not only increases the amount of processing power, it also increases the amount of cache memory. This accumulated increase in cache memory results in more data being stored in the cache, reducing memory access times dramatically. This causes extra speedup in addition to the speedup obtained from the processing power.

6.3.8 Analysis of Experimental Results

Looking at the results obtained from the experiments, it is clear that each of the three curves, in each of the five graphs, have an almost identical shape. No abnormal results are present. This behaviour can be explained by the fact that the problem space is easily parallelised. Each of the sources generate a subset of the total number of automata to be generated, and the rest of the workflow executes in complete isolation. With this taken into consideration, we expected the speedup to be very close to linear, and as we discussed in Section 6.3.7, our results confirm this. The only overhead is initially when the workflows are distributed to the worker nodes, and at the end of execution when the worker nodes send their responses back to the master. Another factor in the amount of overhead generated by each of the worker nodes, is the batch size with which each workflow iteration is executed. We discuss this issue in the next section.

6.3.9 Performace Impact of Batch Sizes

While we were running our experiments, we decided to see what the effect of different batch sizes would have on the execution of a workflow. We then proceeded to first re-run experiment 5 with large input with different batch sizes. The results can be seen in Table 6.7 and Figure 6.11.

		Time in milliseconds					
Run number	Batch size						
	1	5	10	50	1000	5000	10000
1	19203	16520	15553	14477	14524	15163	16895
2	19828	16692	15382	14633	14399	15226	16926
3	19781	16240	15507	14445	14415	15319	17682
4	19813	16442	15397	14524	14383	15211	17425
5	19750	16505	15460	14523	14523	15195	17706
Average	19675	16480	15460	14520	14449	15223	17316
% of 100	138.08	115.65	108.50	101.90	101.40	106.83	121.52

Table 6.7: Experiment 5 – Batch Size Difference on Large Set

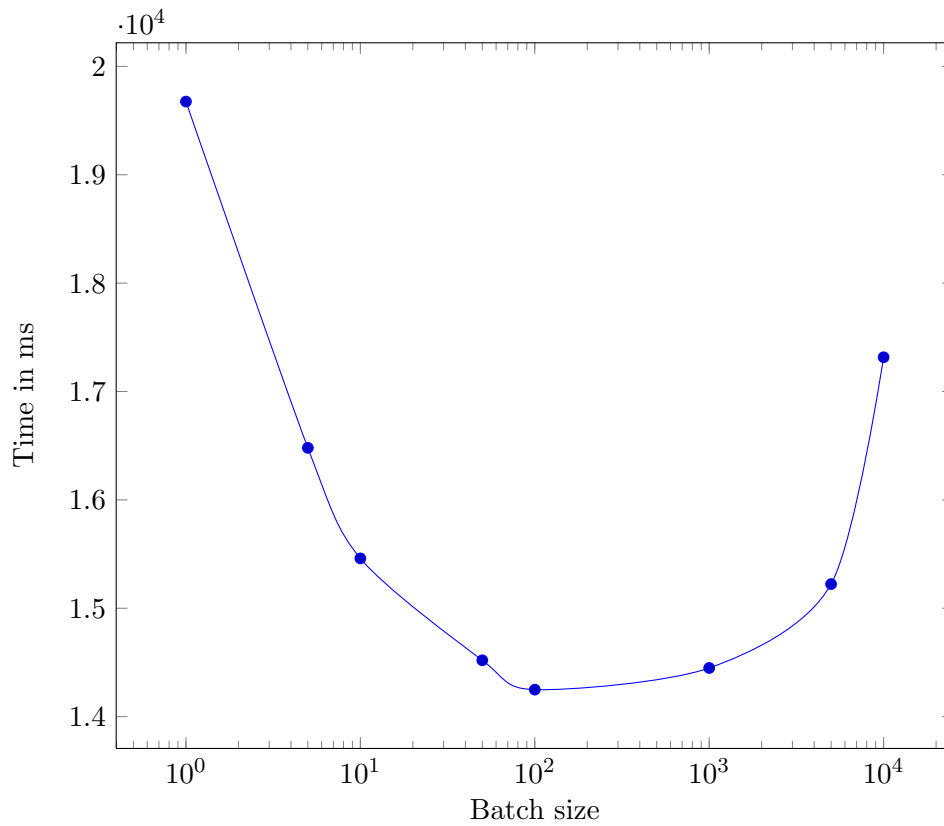


Figure 6.11: Experiment 5 – Batch Size Difference Results

Our initial impression was that the increase in execution time for very small batch sizes can be explained by the increase in the amount of output generated by the output components. Because batch sizes are so small, the components are executed more often, which in turn results in the component producing its output more often. To try and confirm this, we continued to re-run experiment 3 on its medium size set. This experiment produces very minimal output. The results of this can be seen in Table 6.8 and Figure 6.12.

Time in milliseconds							
Run number	Batch size						
	1	5	10	50	1000	5000	10000
1	29936	28813	26255	24538	24554	26193	27300
2	29858	28436	25647	24508	24586	26192	27159
3	30217	26676	26130	24539	24726	26208	27269
4	29999	27955	26395	24570	24601	26130	27207
5	29983	27692	26021	24570	24701	26286	27378
Average	29999	27914	26090	25545	24634	26202	27263
% of 100	121.57	113.12	105.73	99.47	99.83	106.18	110.48

Table 6.8: Experiment 3 – Batch Size Difference on Large Set

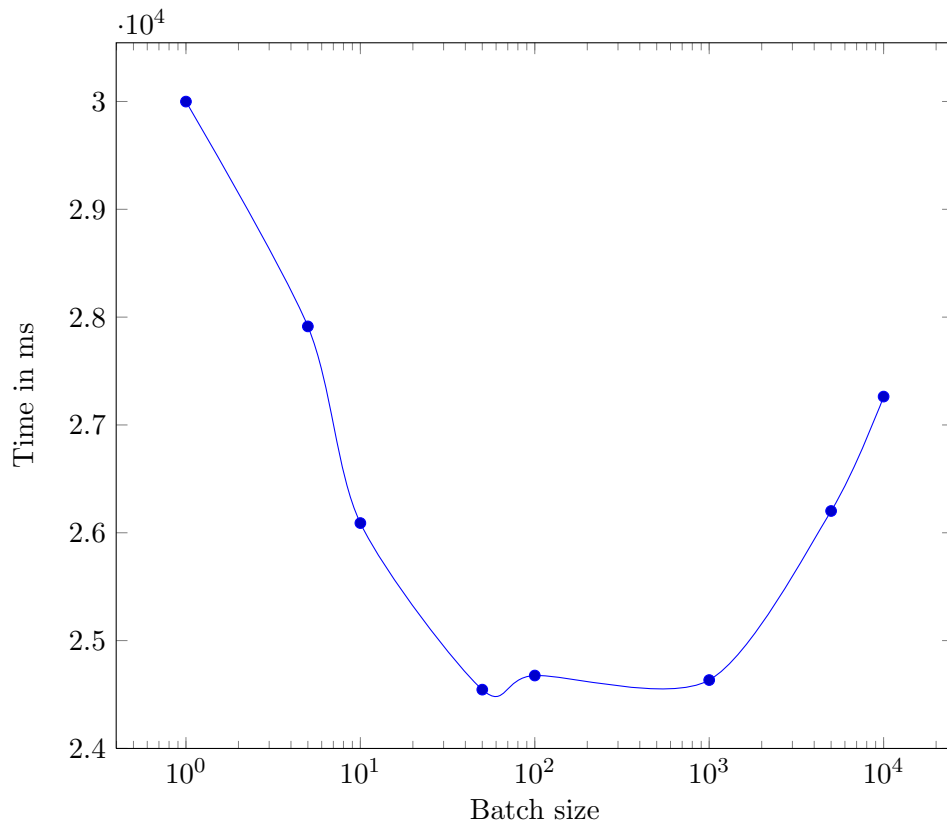


Figure 6.12: Experiment 3 – Batch Size Difference Results

Here we can see that even with essentially no output, the pattern that emerges is the same as with copious amounts of output. Upon investigation, we have come to the conclusion that the difference in execution times between the two polar batch sizes can be explained by the following. The high execution times for both large and small batch sizes is the result of overhead. In the case of small batch sizes, the entire workflow is executed at an increased rate. The first observation to be made here is that this will result in

an enormous increase in the amount of method calls and object allocation¹. Secondly, we speculate that because the increase in execution of the entire workflow results in each component being executed for a shorter period of time, it makes it more difficult for the JVM to analyse the program flow, and subsequently optimising the bytecode by compiling it to native code is not optimal.

For the large batch sizes, we are of the opinion that even though the JVM can optimise the bytecode really well (since the same executable code is run for longer periods of time and analysing the program flow is easier), the overhead of the increased memory allocation from working with larger sets, shadows the increase in performance from the optimisation.

¹While some might reason that the same number of objects should be created since in total the same number of automata are generated, this is not the case. The framework creates various objects to assist with the workflow execution, during each of the workflow iterations. An example of such an object is the container classes that are required to wrap the data that a component generates.

Chapter 7

Conclusion

The goal of this thesis was to develop a user-friendly, accessible, and extendible framework wherein users are able to create complex workflows to generate, manipulate and output automata in a distributed environment, so as to increase the performance of the execution time. We believe that this goal has been met, and the results that were obtained and discussed in Chapter 6 substantiate this.

The results confirm that running the same workflows on variable amounts of worker nodes reduces the execution time almost perfectly linearly, with the overhead in communication between worker nodes and the master increasing the execution time slightly.

7.1 Future Work

There is still a lot of work that can be done to improve the FATKID framework. The nature of its extensibility enables it to be used for a large number of use-cases. Some of the features we hope to include will be discussed next.

Unequal Distribution Engine

Currently our implementation of the distributed execution engine divides the work equally amongst the worker nodes. While this division yields good results in a homogeneous environment¹, it will definitely not be ideal in a non-homogeneous environment where one worker node can execute more than another. Our first approach to this problem would be to implement a system that uses some form of simple heuristic to determine the processing power of each worker node, and then divide the work according to the capability of each node.

Another approach would be to implement some form of task queue on the master, where worker nodes can fetch new assignments when they finish their tasks. There are two critical issues that arise from this approach, and would need to be resolved to make the system work properly. The first is that the increase in network activity from communication between the worker nodes and the master node could negatively impact performance. The second is defining the granularity of tasks that are fetched from the server. When executing very complex workflows, it might end up being too complicated to divide the workflow into tasks that are easily enough broken down and distributed independently. Some form of adaptive queue management may be needed to make the system work smoothly.

Remote Fetching of Libraries, Plugins and Components

Currently it is up to the user to make sure that all the libraries, plugins and components that are used are synchronised between all the nodes in the execution environment. This is a tedious task, especially during development of new extensions. JSPF does provide the ability to load extensions from remote locations by using the HTTP protocol. One of our next improve-

¹The homogeneous environment here refers to an environment where every worker node runs on similar hardware with the same resources available for execution.

ments for the FATKID framework is to utilise this functionality and enable the loading of remote extensions. Users will then be able to keep a single repository of extensions up to date and worker nodes can load extensions from it.

Improved Network Communication

A particularly challenging area in any distributed system like FATKID is the network communication between worker nodes. Our current implementation works, but is not robust in how it handles failures. Currently, nodes that do not respond are simply ignored and consequently their calculations are missing from the final results. This major point will be addressed in the future. We will define a proper communication protocol that will improve the system's fault-tolerance and that will gracefully deal with communication problems.

7.2 Proposed Future Architecture

Developing this first version of the FATKID system has taught us a lot. We have a much finer appreciation for the problems involved and have started thinking of new solutions to address some of the issues. In particular, we have given a lot of thought to how the next generation of architecture for FATKID would interact with our design goals which we believe would still be central to the enterprise:

E *Extensibility*: As with the current FATKID framework, extensibility is paramount. The new architecture (which we will discuss briefly), will make it even easier for developers to extend the framework. The extension points will be much simpler than currently, with fewer methods required to be implemented.

P *Performance*: Again, performance would be the most important goal.

While it is not possible to state with certainty that the new architecture we propose will perform better, we suspect that some performance gains will realise. With the new architecture we plan to eliminate the current need to convert components to their string representation which is currently required to execute certain functionality.

U *User-friendliness*: User-friendliness should stay fairly similar. But as with the current FATKID framework, the editors and engines are simply plugins, and developers are able to create more user-friendly plugins with the current iteration already.

A *Accessibility*: As with the current FATKID, the new version will also be open-sourced and well-documented.

Introduction

Without going into too much detail regarding the proposed architecture (as we ourselves are still investigating it), the most prominent change will be the way communication between components in the workflow is handled. Currently, as discussed in Section 4.4.3, developers are required to implement three methods, namely `split`, `merge`, and `execute`, to enable the communication and data passing between nodes. With sink components, two extra methods are required to enable networked communication, namely `serializeOutputToString` and `executeFromString`. In our future architecture, these five methods will be replaced with a single method, namely `execute`. Components will no longer need to be aware of how data communication takes place, as this will be handled by the framework.

Communication via Queues

All communication in the new framework will happen via queues. By using queues, and intelligent queue injection by the framework, we can imme-

diately eliminate the need for the splitting and merging functions, as the queues automatically cater for this. Merging is handled by injecting the same queue instance into the various components, and those components simply placing their output data onto the queue. Splitting is handled by using a generic deep copy² algorithm which enable the splitting of any data added to the queue.

We plan design a generic queue interface with two initial implementations. The first will handle all local³ communication between components. This implementation will be a straight-forward queue implementation as no special functionality is required. The second queue implementation is where the complexity lies. This will be a network-aware queue implementation. An abstract representation of the implementation is shown in Figure 7.1. The queue itself will know how to serialise, send, and acknowledge the communication of data between remote worker nodes. This will remove the burden from plugin developers to write serialisation code.

In short, we envision that every component's data output stream is assigned an identifier, uniquely identifying the origin of data. As data is placed onto the queue, the queue will be aware of the target locations of the data and can perform the required actions to clone and distribute the data. If the target location is remote, it is able to serialise and send the data to its remote queue counterpart. Because each component has a stream identifier that is preserved, the remote locations will be able to send the results of their executions to the correct worker nodes. This architecture will allow greater functionality over the current framework in the sense that it will

²By default the `clone()` function on a Java object performs a shallow copy. This means that any objects that an object references will not also be cloned. The same object references are passed to the new cloned object.

³Local communication refers to any data communication between components on the same local machine, in other words data that does not need to be sent to other worker nodes through the network.

enable the development of transformer components that can execute on a single node, for example a unique filter component.

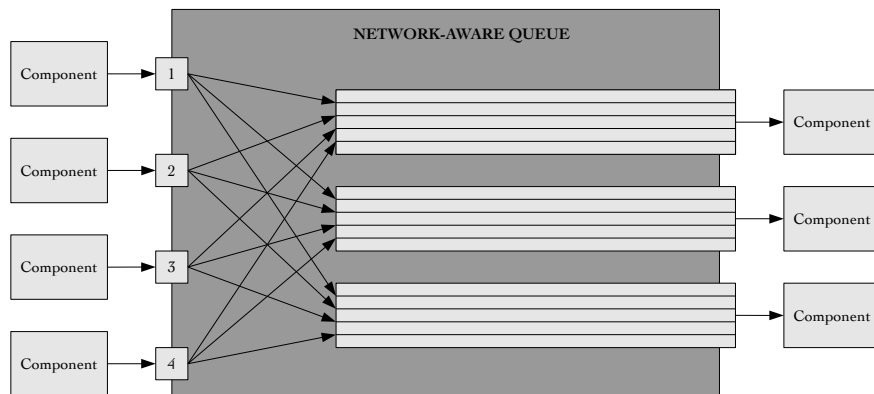


Figure 7.1: Network-Aware Queue Implementation

Appendix A

Source Code

All the code that we present here has been formatted to fit and display better in a \LaTeX document and does not reflect the format of the actual source code. Spacing has been changed, and lines and comments removed to reduce the size of the listings.

A.1 DFA Minimization Filter

```

1  @PluginImplementation
2  public class DfaMinimizer
3      extends AbstractSourceSinkComponent<List<DFA>, List<DFA>> {
4
5      private static final String COMPONENT_NAME = "DFA Minimizer";
6      private static final String PROP_ALGORITHM = "Algorithm";
7      private DFAMinimizer.ALGORITHM algorithm;
8
9      @Override
10     public String getComponentName() {
11         return COMPONENT_NAME;
12     }
13
14     public DfaMinimizer() {
15         initProperties();
16     }
17
18     @Override
19     public List<DFA>[] split(final List<DFA> dfas, int size) {
20         List<DFA>[] returnSet = new LinkedList[size];
21         for (int i = 0; i < size; i++) {
22             returnSet[i] = new LinkedList<DFA>();
23             for (final DFA dfa : dfas) {
24                 returnSet[i].add((DFA) dfa.createCopy());
25             }
26         }
27         return returnSet;
28     }
29
30     @Override
31     public List<DFA> merge(final List<DFA>[] dfaSets) {
32         List<DFA> returnSet = new LinkedList<DFA>();
33         for (final List<DFA> dfaSet : dfaSets) {
34             returnSet.addAll(dfaSet);
35         }
36         return returnSet;
37     }
38
39     @Override
40     public List<DFA> execute(final List<DFA> dfas) {
41         parseProperties();
42         List<DFA> returnSet = new LinkedList<DFA>();
43         for (DFA dfa : dfas) {
44             returnSet.add(new DFAMinimizer()
45                 .minimize(dfa, DFAMinimizer.ALGORITHM.HOPCROFT));
46         }
47         return returnSet;
48     }
49
50     @Override
51     public List<DFA>[] getNewInputContainer(int size) {
52         List[] returnSets = new LinkedList[size];

```

APPENDIX A. SOURCE CODE

91

```

53     for (int i = 0; i < size; i++) {
54         returnSets[i] = new LinkedList<DFA>();
55     }
56     return returnSets;
57 }
58
59 private void initProperties() {
60     final Set<Property> properties = new HashSet<Property>();
61     final Property property = new Property(
62         PROP_ALGORITHM,
63         "Hopcroft",
64         new String[] {
65             "Hopcroft",
66             "Brzozowski"
67         });
68     properties.add(property);
69     setProperties(properties);
70 }
71
72 @Override
73 public void parseProperties() {
74     for (Property property : getProperties()) {
75         if (property.getName().equals(PROP_ALGORITHM)) {
76             if (property.getValue().equals("Hopcroft")) {
77                 algorithm = DFAMinimizer.ALGORITHM.HOPCROFT;
78             } else if (property.getValue().equals("Brzozowski")) {
79                 algorithm = DFAMinimizer.ALGORITHM.BRZOZOWSKI;
80             }
81         }
82     }
83 }
84
85 @Override
86 public PropertyDistributor getPropertyDistributorFor(
87     final String propertyName) {
88     return null;
89 }
90
91 @Override
92 public FatkidComponent getNewInstance() {
93     return new DfaMinimizer();
94 }
95 }

```

A.2 DFA Generator

```

1  @PluginImplementation
2  public class DfaGenerator
3      extends AbstractSourceComponent<List<DFA>> {
4
5      private static final String COMPONENT_NAME = "DFA Generator";
6      private static final String PROP_NUM_DFA
7          = "Number of DFAs";
8      private static final String PROP_NUM_STATE
9          = "Number of States";
10     private static final String PROP_NUM_SYMBOL
11         = "Number of Symbols";
12     private static final String PROP_NUM_ACCEPTING
13         = "Number of Accepting States";
14     private static final String PROP_TRANS_PROB
15         = "Transition Probability";
16     private final Map<String, PropertyDistributor> propertyDistributorMap;
17     private int numDfas = 0;
18     private int numToSource = -1;
19     private int numStates = 0;
20     private int numSymbols = 0;
21     private int numAccepting = 0;
22     private double transProb = 0.0f;
23
24     public DfaGenerator() {
25         propertyDistributorMap = new HashMap<String, PropertyDistributor>();
26         initProperties();
27     }
28
29     @Override
30     public String getComponentName() {
31         return COMPONENT_NAME;
32     }
33
34     @Override
35     public List<DFA>[] split(final List<DFA> dfaSet, final int size) {
36         final List[] returnSet = new LinkedList[size];
37         for (int i = 0; i < size; i++) {
38             returnSet[i] = new LinkedList<DFA>();
39             for (final DFA dfa : dfaSet) {
40                 returnSet[i].add(dfa.createCopy());
41             }
42         }
43         return returnSet;
44     }
45
46     @Override
47     public List<DFA> execute() {
48         final List<DFA> returnSet = new LinkedList<DFA>();
49         int numToGenerate = numToSource == -1 ? numDfas : numToSource;
50         for (int i = 0; i < numToGenerate; i++) {
51             returnSet.add(generateRandomDfa());
52         }

```

```

53     return returnSet;
54 }
55
56 @Override
57 public int getTotalNumberToSource() {
58     return numDfas;
59 }
60
61 @Override
62 public void setTotalNumberToSource(final int numToSource) {
63     numDfas = numToSource;
64 }
65
66 @Override
67 public void setNumberToSourceForBatch(final int numberToSourceForBatch) {
68     numToSource = numberToSourceForBatch;
69 }
70
71 @Override
72 public void parseProperties() {
73     for (Property property : properties) {
74         if (property.getName().equals(PROP_NUM_DFA)) {
75             numDfas = Integer.parseInt(property.getValue());
76         } else if (property.getName().equals(PROP_NUM_STATE)) {
77             numStates = Integer.parseInt(property.getValue());
78         } else if (property.getName().equals(PROP_NUM_SYMBOL)) {
79             numSymbols = Integer.parseInt(property.getValue());
80         } else if (property.getName().equals(PROP_NUM_ACCEPTING)) {
81             numAccepting = Integer.parseInt(property.getValue());
82         } else if (property.getName().equals(PROP_TRANS_PROB)) {
83             transProb = Double.parseDouble(property.getValue());
84         }
85     }
86 }
87
88 private void initProperties() {
89     Set<Property> properties = new HashSet<Property>();
90     final Property numberOfDfas = new Property(PROP_NUM_DFA, "10", null);
91     numberOfDfas.setPropertyValidator(new PropertyValidator() {
92         @Override
93         public boolean validate(String s) {
94             try {
95                 Integer.parseInt(s);
96                 return true;
97             } catch (Exception e) {
98                 return false;
99             }
100         }
101     });
102
103     propertyDistributorMap.put(
104         numberOfDfas.getName(),
105         new PropertyDistributor() {
106             @Override

```

```

107         public String distribute(
108             final String value,
109             final int numNodes) {
110             return Integer.toString(
111                 Integer.parseInt(value) / numNodes
112             );
113         }
114     }
115 );
116
117 final Property numberOfStates
118     = new Property(PROP_NUM_STATE, "10", null);
119 numberOfStates.setPropertyValidator(new PropertyValidator() {
120     @Override
121     public boolean validate(String s) {
122         try {
123             Integer.parseInt(s);
124             return true;
125         } catch (Exception e) {
126             return false;
127         }
128     }
129 });
130
131 final Property numberOfSymbols
132     = new Property(PROP_NUM_SYMBOL, "5", null);
133 numberOfSymbols.setPropertyValidator(new PropertyValidator() {
134     @Override
135     public boolean validate(String s) {
136         try {
137             Integer.parseInt(s);
138             return true;
139         } catch (Exception e) {
140             return false;
141         }
142     }
143 });
144
145 final Property numberOfAccepting
146     = new Property(PROP_NUM_ACCEPTING, "2", null);
147 numberOfAccepting.setPropertyValidator(new PropertyValidator() {
148     @Override
149     public boolean validate(String s) {
150         try {
151             Integer.parseInt(s);
152         } catch (Exception e) {
153             return false;
154         }
155
156         return true;
157     }
158 });
159
160 final Property transProb

```

```

161         = new Property(PROP_TRANS_PROB, "0.75", null);
162     transProb.setPropertyValidator(new PropertyValidator() {
163         @Override
164         public boolean validate(String s) {
165             try {
166                 double prob = Double.parseDouble(s);
167                 return prob >= 0 && prob <= 1;
168             } catch (Exception e) {
169                 return false;
170             }
171         }
172     });
173
174     properties.add(numberOfDfas);
175     properties.add(numberOfStates);
176     properties.add(numberOfSymbols);
177     properties.add(numberOfAccepting);
178     properties.add(transProb);
179     setProperties(properties);
180 }
181
182 private DFA generateRandomDfa() {
183     final DFABuilder builder = new DFABuilder();
184     builder.addStates(numStates);
185     builder.setInitialState(0);
186     if (numAccepting == numStates) {
187         for (int i = 0; i < numAccepting; i++) {
188             builder.addAcceptingState(i);
189         }
190     } else {
191         final Set<Integer> accepting = new HashSet<Integer>();
192         for (int i = 0; i < numAccepting; i++) {
193             int accept = (int) (Math.random() * numStates);
194             while (accepting.contains(accept)) {
195                 accept = (int) (Math.random() * numStates);
196             }
197             builder.addAcceptingState(accept);
198         }
199     }
200
201     for (int i = 0; i < numSymbols; i++) {
202         builder.addSymbol(String.valueOf(i));
203     }
204
205     for (int i = 0; i < numStates; i++) {
206         for (int j = 0; j < numSymbols; j++) {
207             if (Math.random() <= transProb) {
208                 int toState = (int) (Math.random() * numStates);
209                 if (toState != i) {
210                     builder.addTransition(i, String.valueOf(j), toState);
211                 }
212             }
213         }
214     }

```

APPENDIX A. SOURCE CODE

96

```
215         return builder.compile();
216     }
217
218     @Override
219     public PropertyDistributor getPropertyDistributorFor(
220         final String propertyName) {
221         return propertyDistributorMap.get(propertyName);
222     }
223
224     @Override
225     public FatkidComponent getNewInstance() {
226         return new DfaGenerator();
227     }
228 }
```

A.3 DFA Unique Counter

```

1  @PluginImplementation
2  public class DfaUniqueCounter
3      extends AbstractSinkComponent<List<DFA>> {
4
5      private static final String COMPONENT_NAME
6          = "DFA Unique Counter";
7      private static final String PROPERTY_MODE
8          = "Calculation Mode";
9
10     private final Set<String> uniqueDfaSet;
11     private boolean doExact;
12     private int numCounted;
13
14     public DfaUniqueCounter() {
15         uniqueDfaSet = new HashSet<String>();
16         doExact = true;
17         initProperties();
18     }
19
20     @Override
21     public String getComponentName() {
22         return COMPONENT_NAME;
23     }
24
25     @Override
26     public void parseProperties() {
27         for (Property property : getProperties()) {
28             if (property.getName().equals(PROPERTY_MODE)) {
29                 if (property.getValue().equals("Exact")) {
30                     doExact = true;
31                 } else if (property.getValue().equals("MD5")) {
32                     doExact = false;
33                 }
34             }
35         }
36     }
37
38     @Override
39     public PropertyDistributor getPropertyDistributorFor(
40         final String property) {
41         return null;
42     }
43
44     @Override
45     public FatkidComponent getNewInstance() {
46         return new DfaUniqueCounter();
47     }
48
49     @Override
50     public List<DFA> merge(final List<DFA>[] sets) {
51         final List<DFA> merged = new LinkedList<DFA>();
52

```



```

53     for (final List<DFA> set : sets) {
54         merged.addAll(set);
55     }
56
57     return merged;
58 }
59
60 @Override
61 public void execute(final List<DFA> dfas) {
62     for (final DFA dfa : dfas) {
63         final String serializedDfa
64             = DFAUtil.serializeToString(dfa);
65         if (doExact) {
66             uniqueDfaSet.add(serializedDfa);
67         } else {
68             uniqueDfaSet.add(
69                 DigestUtils.md5Hex(serializedDfa));
70         }
71
72         numCounted++;
73     }
74
75     System.out.println("COUNTED : " + numCounted);
76     System.out.println(
77         "UNIQUE : " + uniqueDfaSet.size() + "\n");
78 }
79
80 @Override
81 @SuppressWarnings("unchecked")
82 public List<DFA>[] getNewInputContainer(final int size) {
83     final List[] returnSet = new LinkedList[size];
84
85     for (int i = 0; i < size; i++) {
86         returnSet[i] = new LinkedList<DFA>();
87     }
88
89     return returnSet;
90 }
91
92 @Override
93 public String serializeOutputToString(final List<DFA> dfas) {
94     StringBuilder sb = new StringBuilder();
95
96     for (final DFA dfa : dfas) {
97         sb.append(DFAUtil.serializeToString(dfa)).append("~");
98     }
99
100    return sb.deleteCharAt(sb.length() - 1).toString();
101 }
102
103 @Override
104 public void executeFromString(final String input) {
105     final String[] output = input.split("~");
106

```

APPENDIX A. SOURCE CODE

99

```
107     for (final String dfa : output) {
108         if (doExact) {
109             uniqueDfaSet.add(dfa);
110         } else {
111             uniqueDfaSet.add(DigestUtils.md5Hex(dfa));
112         }
113
114         numCounted++;
115     }
116
117     System.err.println("COUNTED : " + numCounted);
118     System.err.println(
119         "UNIQUE : " + uniqueDfaSet.size() + "\n");
120 }
121
122 private void initProperties() {
123     final Set<Property> properties = new HashSet<Property>();
124     final Property property = new Property(
125         PROPERTY_MODE,
126         "Exact",
127         new String[] {"Exact", "MD5"});
128
129     properties.add(property);
130     setProperties(properties);
131 }
132
133 @Override
134 public void cleanup() {
135     System.err.println("CLEARING");
136     uniqueDfaSet.clear();
137     numCounted = 0;
138 }
139 }
```

A.4 Parallel Execution Pipeline Construction

```

1 private boolean generateExecutionPipeline(
2     final Set<FatkidWorkflowNode> nodes) {
3     final Set<FatkidWorkflowNode> nextExecution
4         = new HashSet<FatkidWorkflowNode>();
5     final Map<String, Integer> processedNodes
6         = new HashMap<String, Integer>();
7
8     for (final FatkidWorkflowNode nodeFatkid : nodes) {
9         if (nodeFatkid.getComponent() instanceof SourceComponent) {
10            nextExecution.add(nodeFatkid);
11        }
12    }
13    if (nextExecution.isEmpty()) {
14        return false;
15    }
16    while (!nextExecution.isEmpty()) {
17        final Set<FatkidWorkflowNode> parallelExecutionPool
18            = new HashSet<FatkidWorkflowNode>();
19        final Set<FatkidWorkflowNode> currentExecution
20            = new HashSet<FatkidWorkflowNode>();
21
22        for (FatkidWorkflowNode n : nextExecution) {
23            currentExecution.add(n);
24        }
25        nextExecution.clear();
26        for (final FatkidWorkflowNode nodeFatkid : currentExecution) {
27            final int referenceCount
28                = processedNodes.containsKey(nodeFatkid.getNodeName())
29                    ? processedNodes.get(nodeFatkid.getNodeName())
30                    : 0;
31            if (nodeFatkid.getNumConnectingSources() == referenceCount) {
32                parallelExecutionPool.add(nodeFatkid);
33                for (final FatkidWorkflowNode sinkNodeFatkid
34                    : nodeFatkid.getSinkNodes()) {
35                    nextExecution.add(sinkNodeFatkid);
36                    if (!processedNodes
37                        .containsKey(sinkNodeFatkid.getNodeName())) {
38                        processedNodes.put(sinkNodeFatkid.getNodeName(), 1);
39                    } else {
40                        processedNodes.put(sinkNodeFatkid.getNodeName(),
41                            processedNodes.get(
42                                sinkNodeFatkid.getNodeName()+1);
43                    }
44                }
45            } else {
46                nextExecution.add(nodeFatkid);
47            }
48        }
49        executionPipeline.add(parallelExecutionPool);
50    }
51    return true;
52 }

```

A.5 FatkidPlugin

```
1 package za.ac.sun.cs.fatkid.lib.plugin;
2
3 import net.xeoh.plugins.base.Plugin;
4
5 /**
6  * The FatkidPlugin interface defines the contract for any plugin in the
7  * FATKID system. This interface should not be confused with the
8  * FatkidComponent interface, which also defines plugin behaviour (but
9  * handled differently in the system). This interface should be used for
10 * any functional component of the system (eg. editors, engines), whereas
11 * the FatkidComponent interface is only used to define components in a
12 * workflow (used in editors).
13 */
14 public interface FatkidPlugin extends Plugin {
15     /**
16      * The name of the plugin. This is mainly used for display purposes.
17      *
18      * @return - String - The name of the plugin.
19      */
20     public String getPluginName();
21
22     /**
23      * The version of the plugin. This mainly used for display purposes
24      * to identify to the user which version of a plugin he is using.
25      *
26      * @return - String - A string representation of the version of the
27      *                plugin.
28      */
29     public String getPluginVersion();
30
31     /**
32      * The initialize method is called right after the plugin has been
33      * loaded by the plugin loader. The plugin developer is free to simply
34      * return here, but it provides a mechanism to run initialization code
35      * for plugins, if the plugin developer has any. If you want your plugin
36      * to act as an event handler, this is where you should have your plugin
37      * register itself on the event bus. If you want your plugin to listen
38      * for the 'PluginsLoadedEvent' (the event fired after all plugins have
39      * been loaded, containing a set of all the plugins), it MUST register
40      * for that event during its initialization.
41      */
42     public void initialize();
43 }
```

A.5.1 FatkidEditor

```
1 package za.ac.sun.cs.fatkid.lib.plugin.editor;
2
3 import za.ac.sun.cs.fatkid.lib.plugin.FatkidPlugin;
4 import javax.swing.*;
5
6 /**
7  * The FatkidEditor interface defines the contract for an editor component
8  * in the system.
9  */
10 public interface FatkidEditor extends FatkidPlugin {
11     /**
12      * Creates a new instance of the editor. This should be implemented to
13      * return different instances each time. This will enable the user to
14      * have multiple tabs of the same editor open at the same time, without
15      * them interfering with each other.
16      *
17      * @return - JPanel - The JPanel containing the GUI for the editor.
18      */
19     public JPanel createInstance();
20 }
```

A.5.2 FatkidEngine

```
1 package za.ac.sun.cs.fatkid.lib.plugin.engine;
2
3 import za.ac.sun.cs.fatkid.lib.plugin.FatkidPlugin;
4 import javax.swing.*;
5
6 /**
7  * The FatkidEngine interface defines the contract for an engine component
8  * in the system.
9  */
10 public interface FatkidEngine extends FatkidPlugin {
11     /**
12      * Creates a new instance of the engine. This should be implemented to
13      * return different instances each time. This will enable the user to
14      * have multiple tabs of the same engine open at the same time, without
15      * them interfering with each other.
16      *
17      * @return - JPanel - The JPanel containing the GUI for the engine.
18      */
19     public JPanel createInstance();
20 }
```

A.6 FatkidComponent

```
1 package za.ac.sun.cs.fatkid.lib.plugin.editor.component;
2
3 import net.xeoh.plugins.base.Plugin;
4 import java.util.Set;
5
6 /**
7  * The FatkidComponent interface defines the contract for a component in
8  * the workflow of an editor. This interface, even though defining a plugin
9  * in the system, should not be confused with the FatkidPlugin interface.
10 * They are handled differently in the system. The FatkidPlugin interface
11 * should be used for functional components in the system (eg. editors,
12 * engines), while the FatkidComponent interface should only be used to
13 * define components in the workflow of an editor.
14 */
15 public interface FatkidComponent extends Plugin {
16     /**
17      * The name of the component.
18      *
19      * @return - String - The name of the component.
20      */
21     public String getComponentName();
22
23     /**
24      * Gets the set of properties available for the component. This is
25      * displayed to the user and enables him to manipulate the values to
26      * define the behaviour of the component.
27      *
28      * @return - Set<Property> - The set of properties.
29      */
30     public Set<Property> getProperties();
31
32     /**
33      * Sets the properties for the component, changing its behaviour.
34      *
35      * @param properties - The set of properties.
36      */
37     public void setProperties(final Set<Property> properties);
38
39     /**
40      * Forces parsing of all properties currently set on the component. This
41      * is used during execution of a pipeline running in batched mode, where
42      * some of the properties are required before execution of the component
43      * starts.
44      */
45     public void parseProperties();
46
47     /**
48      * Gets the property distributor for a given property. If the property
49      * has no distributor, it should return null.
50      *
51      * @param propertyName - The name of the property for which to retrieve
52      * a distributor.
```

```
53     * @return - PropertyDistributor - The property's distributor.
54     */
55     public PropertyDistributor getPropertyDistributorFor(
56         final String propertyName);
57
58     /**
59     * Creates a new instance of the component. This is needed due to the
60     * fact that the plugin manager loads one instance of each component,
61     * but during execution we need multiple instances for parallel
62     * computation.
63     *
64     * @return - FatkidComponent - The new FatkidComponent instance.
65     */
66     public FatkidComponent getNewInstance();
67
68     /**
69     * Called after the plugin has finished to free any resources or reset
70     * plugins.
71     */
72     public void cleanup();
73 }
```


A.6.1 SourceComponent

```

1  package za.ac.sun.cs.fatkid.lib.plugin.editor.component;
2
3  /**
4   * The SourceComponent defines the interface for a component in the
5   * FATKID workflow that acts only as a Source (component that generates
6   * output).
7   *
8   * @param <R>
9   */
10 public interface SourceComponent<R> extends FatkidComponent {
11     /**
12      * The split function is used in the workflow to split the output to
13      * various sinks. The functionality of the splitting is for the plugin
14      * developer to define, and can in fact reference the same data for
15      * each index, giving the same output to each sink.
16      *
17      * @param r - The data to split.
18      * @param n - The number of splits.
19      * @return - R[] - An array of size n, each index containing a piece of
20      *           the split data.
21      */
22     public R[] split(R r, int n);
23
24     /**
25      * The execute method defines the actual functionality of the component.
26      * Any calculations that need to happen should be in here.
27      *
28      * @return - R - The output from the component.
29      */
30     public R execute();
31
32     /**
33      * Should return the number specified to source. This should ideally be
34      * represented internally in the source component as a property.
35      */
36     public int getTotalNumberToSource();
37
38     /**
39      * Set (override) the number specified to source. This should ideally be
40      * represented internally in the source component as a property. This is
41      * used during the distribution process to divide the amount to source.
42      */
43     public void setTotalNumberToSource(final int numToSource);
44
45     /**
46      * Sets the number to source in a single batch run. This should override
47      * the global number to generate, as the framework will take
48      * responsibility of setting the batch sizes and calling it the
49      * appropriate amount of times.
50      *
51      * @param numberToSourceForBatch - The number to source in the next
52      *                                batch.

```

```
53     */  
54     public void setNumberToSourceForBatch(final int numberToSourceForBatch);  
55 }
```

A.6.2 SinkComponent

```

1  package za.ac.sun.cs.fatkid.lib.plugin.editor.component;
2
3  /**
4   * The SinkComponent defines the interface for a component in the
5   * FATKID workflow that acts only as a Sink (component that accepts input).
6   *
7   * @param <A> - The ACCEPTING type. This type of variable is accepted by
8   *             the 'execute' method.
9   */
10 public interface SinkComponent<A> extends FatkidComponent {
11     /**
12      * The merge function is used in the workflow to combine the inputs
13      * from various sources. The functionality of the merging is for the
14      * plugin developer to define, and can in fact do nothing if he only
15      * ever expects to receive input from a single source.
16      *
17      * @param a - The variable number of parameters of type A (Accepting
18      *           Type).
19      */
20     public A merge(A[] a);
21
22     /**
23      * The execute method defines the actual functionality of the
24      * component. Any calculations that need to happen should be in here.
25      *
26      * @param a - The input from a source.
27      */
28     public void execute(A a);
29
30     /**
31      * The getNewInputContainer method should return an array of length
32      * "size", all indexes containing null objects. This enables the system
33      * to wrap objects being passed as parameters in a container with the
34      * native type used by the component, and eliminates the need to use
35      * reflection.
36      *
37      * @param size - The size of the new array.
38      * @return - A[] - An array of type A.
39      */
40     public A[] getNewInputContainer(int size);
41
42     /**
43      * The serializeOutputToString method should return a String
44      * representation of the output that the component generates. This
45      * method is used to serialize the output over the network when the
46      * system is used in a distributed environment.
47      *
48      * @return - String - The String representation of the output.
49      */
50     public String serializeOutputToString(A a);
51
52     /**

```

```
53     * The executeFromString method is essentially the inverse of the
54     * serializeOutputToString method. It is used when serialized output
55     * is received over the network to reconstruct the object and call its
56     * execute method.
57     *
58     * @param input - The serialized output.
59     */
60     public void executeFromString(final String input);
61 }
```

A.6.3 SourceSinkComponent

```

1  package za.ac.sun.cs.fatkid.lib.plugin.editor.component;
2
3  /**
4   * The SourceSinkComponent defines the interface for a component in the
5   * FATKID workflow that acts as both a Source (component that generates
6   * output) and a Sink (component that accepts input).
7   *
8   * @param <R> - The RETURN type. This type of variable will be returned by
9   *             the 'execute' method.
10  * @param <A> - The ACCEPTING type. This type of variable is accepted by
11  *             the 'execute' method.
12  */
13  public interface SourceSinkComponent<R, A> extends FatkidComponent {
14      /**
15       * The merge function is used in the workflow to combine the inputs
16       * from various sources. The functionality of the merging is for the
17       * plugin developer to define, and can in fact do nothing if he only
18       * ever expects to receive input from a single source.
19       *
20       * @param a - The variable number of parameters of type A (Accepting
21       *           Type).
22       */
23      public A merge(A[] a);
24
25      /**
26       * The split function is used in the workflow to split the output to
27       * various sinks. The functionality of the splitting is for the plugin
28       * developer to define, and can in fact reference the same data for
29       * each index, giving the same output to each sink.
30       *
31       * @param r - The data to split.
32       * @param n - The number of splits.
33       * @return - R[] - An array of size n, each index containing a piece of
34       *           the split data.
35       */
36      public R[] split(R r, int n);
37
38      /**
39       * The execute method defines the actual functionality of the
40       * component. Any calculations that need to happen should be in here.
41       *
42       * @param a - The input from a source.
43       * @return - R - The output from the component.
44       */
45      public R execute(A a);
46
47      /**
48       * The getNewInputContainer method should return an array of length
49       * "size", all indexes containing null objects. This enables the system
50       * to wrap objects being passed as parameters in a container with the
51       * native type used by the component, and eliminates the need to use
52       * reflection.

```

APPENDIX A. SOURCE CODE

111

```
53     *
54     * @param size - The size of the new array.
55     * @return - A[] - An array of type A.
56     */
57     public A[] getNewInputContainer(int size);
58 }
```

Appendix B

Experimental Data

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	1528	961	872	603
2	1778	1022	822	538
3	1475	995	685	641
4	1662	952	791	627
5	1289	945	687	616
Average	1546	975	771	605
% Time of base	100.00	63.05	49.88	39.12

Table B.1: Experiment 1 – Small

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	6617	3561	2521	1853
2	5918	3145	2521	2086
3	6080	3301	2354	1899
4	6573	3358	2381	2039
5	6555	3495	2493	1975
Average	6349	3372	2454	1970
% Time of base	100.00	53.11	38.65	31.04

Table B.2: Experiment 1 – Medium

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	12258	6575	4633	3767
2	11980	6319	4576	3532
3	12641	6567	4329	3376
4	12834	6540	4466	3332
5	11669	6657	4566	3444
Average	12276	6532	4514	3490
% Time of base	100.00	53.20	36.77	28.43

Table B.3: Experiment 1 – Large

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	9846	4939	3314	2513
2	9886	4956	3407	2522
3	9838	4946	3312	2539
4	9858	4959	3301	2501
5	9821	4965	3333	2525
Average	9850	4953	3333	2520
% Time of base	100.00	50.29	33.84	25.58

Table B.4: Experiment 2 – Small

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	49034	24560	16399	12298
2	49030	24518	16427	12321
3	49081	24587	16335	12274
4	48786	24693	16428	12307
5	48970	24536	16388	12312
Average	48980	24579	16395	12302
% Time of base	100.00	50.18	33.47	25.12

Table B.5: Experiment 2 – Medium

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	97941	49146	32759	24565
2	97548	48790	32648	24463
3	97619	49122	32698	24545
4	98100	49375	32740	24662
5	98026	49087	32717	24585
Average	97847	49104	32712	24564
% Time of base	100.00	50.18	33.43	25.10

Table B.6: Experiment 2 – Large

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	9765	4992	3401	2574
2	9735	5023	3401	2559
3	9781	5007	3370	2558
4	9828	5007	3400	2574
5	9719	4992	3385	2543
Average	9766	5004	3391	2562
% Time of base	100.00	51.24	34.73	26.23

Table B.7: Experiment 3 – Small

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	48844	24680	16645	12418
2	48516	24664	16676	12433
3	48454	24695	16567	12417
4	48469	24663	16567	12417
5	49047	24679	16551	12386
Average	48666	24679	16601	12414
% Time of base	100.00	50.71	34.11	26.23

Table B.8: Experiment 3 – Medium

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	96752	49234	33072	24617
2	97204	49359	33275	24695
3	96767	49280	33087	24710
4	96812	49265	33073	24679
5	96751	49297	33072	24788
Average	96857	49287	33116	24698
% Time of base	100.00	50.89	34.19	25.50

Table B.9: Experiment 3 – Large

Run number	Time in milliseconds						
	Batch Size						
	1	5	10	50	1000	5000	10000
1	29936	28813	26255	24538	24554	26193	27300
2	29858	28436	25647	24508	24586	26192	27159
3	30217	26676	26130	24539	24726	26208	27269
4	29999	27955	26395	24570	24601	26130	27207
5	29983	27692	26021	24570	24701	26286	27378
Average	29999	27914	26090	25545	24634	26202	27263
% of batch 100	121.57	113.12	105.73	99.47	99.83	106.18	110.48

Table B.10: Experiment 3 – Batch Size Difference on Medium Set

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	11060	5491	4602	3791
2	11622	8252	5663	3135
3	11466	6396	7644	4056
4	13743	7831	5055	4462
5	13416	6567	5257	2793
Average	12261	6907	5644	3647
% Time of base	100.00	56.33	46.03	29.75

Table B.11: Experiment 4 – Small

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	49671	23010	18143	14086
2	45505	27550	19719	15023
3	51105	30077	24009	12387
4	45724	26099	21326	14820
5	44554	27503	24835	14929
Average	47312	26848	21606	14249
% Time of base	100.00	56.75	45.67	30.12

Table B.12: Experiment 4 – Medium

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	103974	61807	45677	33384
2	112024	67408	40030	32448
3	104333	59061	44787	37175
4	108872	65162	40155	35412
5	126376	65130	47190	33727
Average	111116	63714	43568	34429
% Time of base	100.00	57.34	39.21	30.98

Table B.13: Experiment 4 – Large

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	10779	5366	3634	2776
2	10795	5413	3650	2808
3	10717	5366	3604	2808
4	10811	5367	3619	2746
5	10686	5382	3619	2761
Average	10758	5379	3625	2780
% Time of base	100.00	50.00	33.70	25.84

Table B.14: Experiment 5 – Small

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	21419	10780	7161	5428
2	21356	10717	7145	5460
3	21497	10733	7239	5397
4	21403	10764	7129	5414
5	21294	10702	7144	5413
Average	21394	10739	7164	5422
% Time of base	100.00	50.20	33.48	25.35

Table B.15: Experiment 5 – Medium

Run number	Time in milliseconds			
	Number of nodes			
	2	4	6	8
1	42588	21356	14227	10826
2	42962	21309	14320	10702
3	42682	21497	14259	10733
4	42744	21341	14212	10732
5	42791	21419	14228	10811
Average	42753	21384	14249	10761
% Time of base	100.00	50.02	33.33	25.17

Table B.16: Experiment 5 – Large

Run number	Time in milliseconds						
	Batch Size						
	1	5	10	50	1000	5000	10000
1	19203	16520	15553	14477	14524	15163	16895
2	19828	16692	15382	14633	14399	15226	16926
3	19781	16240	15507	14445	14415	15319	17682
4	19813	16442	15397	14524	14383	15211	17425
5	19750	16505	15460	14523	14523	15195	17706
Average	19675	16480	15460	14520	14449	15223	17316
% of batch 100	138.08	115.65	108.50	101.90	101.40	106.83	121.52

Table B.17: Experiment 5 – Batch Size Difference on Large Set

Bibliography

- [1] Hopcroft, J.E. and Ullman, J.D. *Introduction to Automata Theory, Languages and Computation*. Addison Wesley Publishing Company, 1979.
- [2] Hopcroft, J.E. *An $n \log n$ algorithm for minimizing states in a finite automaton*. The Theory of Machines and Computations (Z. Kohavi, ed.), pp. 189–196, Academic Press, New York.
- [3] Brzozowski, J.A. *Canonical regular expressions and minimal state graphs for definite events*. Proceedings of the Symposium on *Mathematical Theory of Automata*, New York, NY, April 24–26, 1962. Polytechnic Press of the Polytechnic Institute of Brooklyn, Brooklyn, NY, pp. 529–561, 1963.
- [4] Sipser, M. *Introduction to the Theory of Computation*. Second Edition, International Edition. Course Technology, 2005.
- [5] Le Maout, V. *The Automata Standard Template Library*. Last retrieved June 27, 2012, from <http://astl.sourceforge.net/>
- [6] Le Maout, V. *Thèse de doctorat - Expérience de programmation générique sur des structures non-séquentielles : les automates*. Ph.D. Thesis. Université de Marne La Vallée, 2003.
- [7] Le Maout, V, Revuz, D, Daragon, X and Adant, A. *Automaton Standard Template Library Documentation*. Last retrieved September 20, 2012, from <http://igm.univ-mlv.fr/lemaout/ASTL/DOC/>

- [8] Hewlett-Packard Company. *Standard Template Library Programmer's Guide*. Last retrieved September 20, 2012, from <http://www.sgi.com/tech/stl/>
- [9] Sander, G. *Visualization of Compiler Graphs*. Last retrieved September 20, 2012, from <http://rw4.cs.uni-sb.de/users/sander/html/gsvcg1.html>
- [10] Graehl, J. *Carmel Finite-State Toolkit*. Last retrieved June 27, 2012, from <http://www.isi.edu/licensed-sw/carmel/>
- [11] Van der Merwe, B. *Introduction to Carmel*. University of Stellenbosch. Retrieved from <http://www.cs.sun.ac.za/rw711/lectures/lec2/l2.pdf>
- [12] Møller, A. *dk.brics.automaton*. Last retrieved June 27, 2012, from <http://www.brics.dk/automaton/>
- [13] Watson, B. W. *An introduction to the FIRE engine: A C++ toolkit for Finite automata and Regular Expressions*. Computing Science Report 94/21, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1994.
- [14] Watson, B.W. *Taxonomies and Toolkits of Regular Language Algorithms*. Ph.D. Thesis, Department of Computing Science, Eindhoven University of Technology, The Netherlands, 1995.
- [15] Raymond, D. and Wood, D. *Grail+*. Last retrieved June 27, 2012, from <http://www.csd.uwo.ca/Research/grail/>
- [16] Raymond, D and Wood, D. *User's Guide to Grail*. University of Western Ontario, London, Canada and University of Science and Technology, Kowloon, Japan. 1996. Retrieved from <http://www.csd.uwo.ca/Research/grail/.papers/user.ps>
- [17] Pedrazzini, S. *The Jacaranda Framework*. Last retrieved June 27, 2012, from <http://linux3.dti.supsi.ch/pedrazz/jacaranda/index.html>

- [18] Hetherington, L. *The MIT FST Toolkit*. Last retrieved June 27, 2012, from <http://people.csail.mit.edu/ilh/fst/>
- [19] Hetherington, L. *The MIT Finite-State Transducer for Speech and Language Processing*. In Proc. ICSLP, p2609–2612, Jeju, South Korea, 2004. Retrieved from <http://groups.csail.mit.edu/sls/publications/2004/ICSLP04.Hetherington.pdf>
- [20] May, J. *Tiburon*. Last retrieved June 27, 2012, from <http://www.isi.edu/licensed-sw/tiburon/>
- [21] May, J. and Knight, K. *Tiburon: A Weighted Tree Automata Toolkit*. In Proc. 11th Int. Conf. Implementation and Application of Automata, Volume 4094 of LNCS, p102–113. Springer Verlag, 2006.
- [22] Van der Merwe, B. *Introduction to Tiburon*. University of Stellenbosch. Retrieved from <http://www.cs.sun.ac.za/rw711/2012t2/15.pdf>
- [23] *The Java HotSpot Virtual Machine, v1.4.1*. Last retrieved on August 25, 2012, from http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_4.html
- [24] *Java SE 6 Performance White Paper*. Last retrieved on August 25, 2012, from http://java.sun.com/performance/reference/whitepapers/6_performance.html#2.1.6
- [25] *Rich Client Platform*. Last retrieved July 1, 2012, from http://wiki.eclipse.org/index.php/Rich_Client_Platform
- [26] Vogel, L. *Eclipse RCP Tutorial*. Last retrieved July 1, 2012, from <http://www.vogella.com/articles/EclipseRCP/article.html>
- [27] *The Netbeans Platform*. Last retrieved July 1, 2012, from <http://netbeans.org/features/platform/>

- [28] *jspf*. Last retrieved July 1, 2012, from <http://code.google.com/p/jspf/>
- [29] *GNU Trove*. Last retrieved August 13, 2012, from <http://trove4j.sourceforge.net/html/overview.html>
- [30] *Software Framework*. Last retrieved August 6, 2012, from http://en.wikipedia.org/wiki/Software_framework
- [31] *Speedup*. Last retrieved 7 October, 2012, from <http://en.wikipedia.org/wiki/Speedup>
- [32] *Birthday paradox*. Last retrieved 12 November, 2012, from http://en.wikipedia.org/wiki/Birthday_paradox