Automata Over Infinite Alphabets

CS310 Computer Science Project Report

Alexander Ingram

Department of Computer Science

University of Warwick

Supervisor: Dr. Andrzej Murawski

2016-2017

Abstract

Over the years, research into automata theory has shown that their capabilities and usefulness as abstract modelling machines has increased in leaps and bounds. Use of these models has been prevalent in many areas; the most notable would be their role in software verification. Standard automata definitions cannot be used in conjunction with an infinite alphabet, this report focuses on models that are capable of performing computation over an infinite alphabet. This report showcases the research, design and implementation of several automata models, ultimately producing an implementation of a fresh-register automaton. The fresh-register automaton is implemented in the Java programming language and is used in conjunction with XML files to perform property verification; showcasing the power of theoretical modelling when applied to real situations. This project highlights the differences between automata in theory and in practice, using simulation routines to provide insight into how several automata models operate over a range of alphabets.

Keywords

1. Finite-state machines
2. Fresh-register automata
3. Model checking
4. Software verification
5. XML
6. Java

Acknowledgements

I would like to dedicate this section to thanking my project supervisor Dr. Andrzej Murawski for his tireless efforts and consistent contributions to this project. Without his guidance, insight, and support the project could not have succeeded to the same extent that it has. Additionally, his dedication of countless hours to providing feedback is greatly appreciated.

Table of Contents

Chapter 1: Introduction

Standard automata theory makes use of finite alphabets, but for some applications infinite alphabets can be the more rational choice. Extending standard definitions to the infinite alphabet setting requires diligence however, as the automata must still be defined in a finitary way. For instance, automata are often used for model checking; the process of checking that a system satisfies a set of properties. They are also used "notably in hardware verification and to prove correctness of a variety of software models" [1]. But, the set of properties that these automata can be used to verify is limited by both their structure, and the language in which these properties are written: linear temporal logic (LTL) [1]. Extending automata to the infinite alphabet setting will therefore require changing the structure of a typical automaton, which will be detailed further in the chapter. This report documents a project focusing on the research of finite-state machines - namely register automata - and their application to a scenario: XML property verification. The overarching aim of the project was to implement an automaton simulation routine that showed that an XML document conformed to an arbitrary property. In this chapter the project is introduced through a summary of XML, automata theory, the consideration of the overarching incentive for the project, and concludes with an outline of the report.

Background: XML

XML stands for eXtensible Markup Language [2] and was designed to store and transport data whilst being human and machine readable; XML is just data wrapped in tags. It is structured similarly to Hypertext Markup Language (HTML) [3], which is frequently used to display information on a webpage. On the other hand, nothing is actively done with information contained within an XML document. Be that as it may, XML stores data in plain text format; a software and hardware independent way of storing, transporting and sharing data. Hence, XML has a massive domain of applications; one of these being software verification. This is because any kind of data-processing software can make use of XML.

```
<Note>
    <To>Andrzej</To>
    <From>Alex</From>
    <Subject>Dissertation Meeting</Subject>
    <Body>I'll be there!</Body>
</Note>
```

Figure 1 - Example XML Document

```
<!DOCTYPE html>
<html>
    <head>
        <title>Note</title>
    </head>
    <body>
        <h1>Dissertation Meeting</h1>
        <h2>From: Alexander Ingram</h2>
        <h3>To: Andrzej Murawski</h3>
        <p>I'll be there!</p>
    </body>
</html>
```

Figure 2 - Example HTML Document

Figures 1 & 2 show the structure of an XML document and an HTML document, respectively. Whilst comparable to HTML in structure, there are a few differences that should be understood: One key difference is that XML was designed to carry data (with emphasis on what the data is), whereas HTML was designed to present data (with attention to how the data looks). Another significant difference is that XML element tags are not predefined in the same way that HTML element tags are; XML element tags can essentially take any value, where an element tag – shortened to 'tag' in this report - is anything surrounded by angle brackets.

```
<Note id="1">
```

Figure 3 - XML tag with an attribute

XML tags can contain attributes, as shown by Fig. 3. There can be any number of attributes within a single tag, and they serve to convey additional information about the specific element. These attributes are also not predefined, which means that the attribute ID and its associated value ("id" and "1", respectively in Fig. 3) can both take any value; this forms the basis of an infinite alphabet that can be used by some atypical automata.

Background: Automata Theory

Automata are abstract mathematical models of machines that perform computations on an input by moving through a series of states or configurations. A typical input to an automaton is a concatenation of symbols contained within the automaton's *alphabet*. The automaton always starts in a specific *start state* and at each stage of computation, the automaton is said to be in a state **x** such that **x** exists within the automaton's set of states. An automaton with only a finite number of elements within its state set is called a finite-state machine, which is the type of automaton that this report will be focusing on. A *transition function* determines the next state based on the current state and input symbol. The finite-state machine computes until it reaches the end of its input, at which point it either *accepts* or *rejects* depending on whether its current state **x** exists within the set of final states or not.

A finite-state machine is formally defined as a 5-tuple $(S, \Sigma, \delta, S_0, F)$ such that:

- S = Finite set called *the states*
- Σ = Finite set of symbols called *the alphabet*
- δ = A mapping $S \times \Sigma \rightarrow S$ called *the transition function*
- $S_0 \in S$ = The *start state*
- $F \subseteq S$ = The set of *accepting states*.

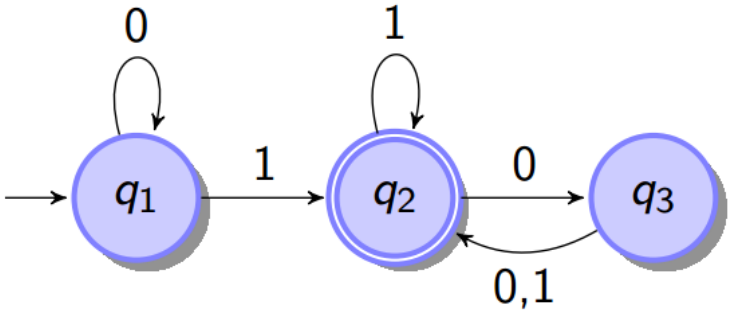

Figure 4 – A deterministic Finite Automaton [4]

Figure 4 illustrates a finite-state machine M with the following definition:

- $S = \{q_1, q_2, q_3\}$
- $\Sigma = \{0,1\}$
- $S_0 = q_1$
- $F = q_2$
- $\delta =$

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

$\delta(q_1, 0) = q_1$	$\delta(q_1, 1) = q_2$
$\delta(q_2, 0) = q_3$	$\delta(q_2, 1) = q_2$
$\delta(q_3, 0) = q_2$	$\delta(q_3, 1) = q_2$

Figure 5 - Example transition table [4]

Let $W = W_1, W_2, \ldots, W_n$ be a string over Σ. We say that a machine M *accepts* W if there exists a sequence of states $R_1, R_2, \ldots, R_n$ from S such that:

- $R_0 = S_0$
- $\delta(R_i, W_{i+1}) = R_{i+1}$, where $0 \leq i < n$
- $R_n \in F$

This means that a finite-state machine will accept the empty string if and only if the start state $S_0 \in F$. The acceptance of the input "010100" can be witnessed by the following sequence of states: $q_1, q_1, q_2, q_3, q_2, q_3, q_2$. This input is one string that is part of a greater whole; a *language.* We say that the language of a machine M is the set of strings A that M accepts. We can then write L(M) = A. Taking Figure 4 as an example, the language of this machine can be surmised as $w \in \{0, 1\}^*$ such that w contains at least one instance of 1 and the last 1 may be followed by either 01, or an even number of zeros. When specifying the language of an automata it is common practice to use the following symbols [5]:

- The Kleene star, $\sum^*$, is a unary operator on an alphabet $\sum$ that gives the infinite set of all possible strings of all possible lengths over $\sum$ including λ (the empty set).
- The Kleene plus, $\sum^+$, is the infinite set of all possible strings of all possible lengths over $\sum$ excluding λ.

A Deterministic Finite Automaton (DFA) is a special kind of Nondeterministic Finite Automaton (NFA) [6], in which there are no states involving multiple transitions over

the same letter. It is for exactly this reason that a DFA is too restrictive for the scope of this project (explained in chapter 3); hence we introduce the NFA.

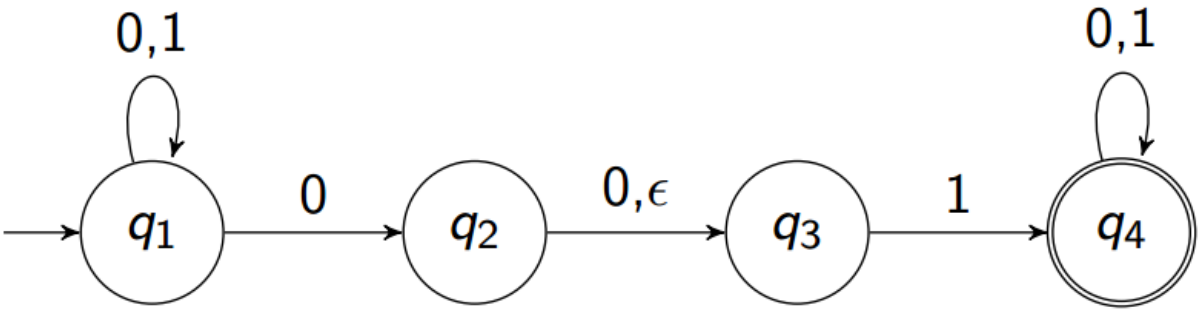

Figure 6 - A Nondeterministic Finite Automaton [6]

An NFA is defined in a very similar way to a DFA but with two important differences:

- The alphabet $\sum$ of an NFA also includes the epsilon symbol $\in$ that represents a transition over the empty string. Thus, we write $\Sigma_\in = \Sigma \cup \{\in\}$.
- The transition function δ now looks like $Q \times \Sigma_\in \rightarrow P(Q)$, where $P(Q)$ is the power set of Q; where the power set of any set S is the set of all subsets of S, including the empty set and S itself. This is because an NFA computes every possible combination of transitions.

Comparing Figures 4 and 6, we see a few key differences in the structure of these two automata. Most notably is the introduction of states that make use of multiple transitions over the same letter. Another quirk of the NFA is that transitions may be marked with $\in$, the epsilon symbol. Upon seeing an $\in$-transition, the automaton may either skip to the next state (indicated by the transition), or ignore the transition and await further input. The question still to be answered is what happens when more than one transition is available? The answer is that the machine splits into multiple copies of itself, where each branch follows one possibility and hence altogether, branches follow all possibilities. If no transition matches the input the branch dies, and the automaton accepts if some branch accepts.

Project Motivation

Register automata theory is being actively researched now with hopes of developing their range of application, or creating more powerful versions of existing models. This project aimed to contribute to this field by implementing a register automaton that can function with an infinite alphabet. XML attributes can take any value, and consequently

the values of these attributes serve as an infinite alphabet. To clarify, the use of register automata (or some form of automata-extension with that effect) and not the automata shown previously in the chapter is necessary because the alphabet of a finite-state machine is usually a finite range of symbols that are fed into the automaton one at a time. Put in context, these automata (NFA/DFA) could be used to verify that the language used to generate individual attribute values is regular – whereas the system we aim to create will compare attribute values (as whole strings) to each other. This is incomputable without some form of memory because finite state machines lack a mechanism to compare symbols to one another. Therefore, some automata with memory must be used if we hope to verify properties. Also, the alphabet that the automaton will use is infinite, as there is no way of limiting the range of values that an attribute can have.

A language currently exists that fulfils a somewhat similar role to the routines proposed in this project, XML Schema [7]. XML Schema, known as XSD, is used to define the structure of an XML document. Written in XML, its similarity to this project stems from the fact that it can be used to constrain attributes and elements. However, this is achieved by constraining the data types that attributes or elements can take, and not the value of the attributes themselves. To summarise, whilst XSD is used to verify that an XML document follows some set of structural definitions, it does not perform the same task as the software that will be developed by the end of this project.

Report Overview

The remainder of the report will be structured as follows:

- Firstly, Chapter 2 focuses on the research of automata with memory or systems to that effect, as well as the research undertaken to formulate methods for parsing XML documents.
- Design, development and final implementation of the project will be detailed in Chapter 3.
- Chapter 4 specifies the testing of the systems developed in Chapter 3.
- Project management methodologies and techniques are declared in Chapter 5.

- Project evaluation is considered in Chapter 6 with respect to technical achievements and hindrances, before the approach to project execution and organisation is inspected.
- The document closes in Chapter 7 with the overarching project conclusions, future work and plausible project extensions.

Chapter 2: Research

XML attributes and their respective values must be extracted from an XML document and processed into *data words;* chunks of information that an automaton can digest easily. The exact definition of a data word will be detailed later in the chapter. This research section is dedicated to two topics: Firstly, how to parse XML elements and their associated attribute values; processing of these values into data words will be detailed in the Chapter 2. Secondly, how to extend automata to the infinite alphabet setting. The entirety of the implementable section of this project was written in Java. As the project was already substantial by itself, the decision was made to use a programming language that been used extensively in the past. Consequently, research related to the implementation of this project was confined to methods available in Java.

XML Parsing: XPATH/DOM Traversal

A prevalent method of parsing XML is by making use of the Document Object Model (DOM) [8]. The DOM is an Application Programming Interface (API) for HTML and XML documents. It simultaneously defines the logical structure of these documents and ways in which they are accessed and manipulated; i.e., anything found in an HTML or XML document can be accessed, changed, deleted, or added by utilising the DOM.

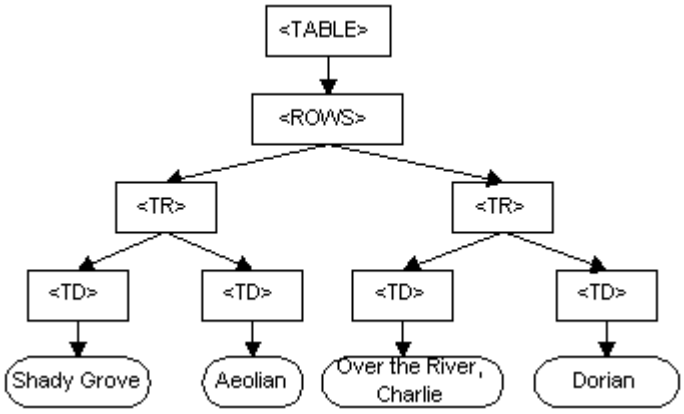

Figure 7 - Example DOM [8]

Figure 7 is a representation of a DOM that describes some table. The table has two rows, with each row subdivided into two columns. As we can see, the DOM has a similar structure to the Tree data structure [9], hence, each node in the DOM has a relationship to any node connected to it. To reach the leaf node “Shady Grove”, we must first travel from TABLE → ROWS, ROWS → TR (left child), TR → TD (left child)

and finally from TD → Shady Grove. The process of describing the route from root (top-most) node to leaf (bottom-most) node is exactly what will be used by the XML parser to first navigate to appropriate attribute nodes and then collect their respective values. Attribute nodes are not shown in Fig. 7 but would have been displayed as an additional child to any element node (an element can have many attributes) with its value being a child to it.

XPATH [10] is a language that navigates through a DOM as if it were a file system (E.G. think of a windows file path with folders and sub-folders) and can be used to gather data from an XML file accordingly. An example of this is shown here [11]. However, as we wish to parse any XML file - including ones we have not seen before - then it is infeasible to use XPATH. This is because it requires the programmer to know the path that the parser will take before runtime, which is an assumption that undermines the usefulness of XPATH in this situation. Therefore, a solution to parsing must be general, in the sense that it must work for any XML file.

Referring to Fig. 7, note that each square box in the image is an element. Each of these elements can have attributes, and it is these that we are after. As this hierarchy only exists in XML and HTML documents, these documents must be imported into Java whilst maintaining the DOM structure. Fortunately, Java has a class for this: DocumentBuilder [12]. This class contains the parse function, which returns a new DOM Document Object when given an XML file as its input. The DOM Document Object (Document) [13] interface embodies the full XML document, theoretically, it is the root of the document tree from which all the elements in the document can be accessed. This is done by inputting "*" to the getElementsByTagName function as "The special value "*" matches all tags". This function returns a NodeList object, which is conceptually the same as a list of Node objects [14]. The Node object makes use of the getAttributes function, which is the final step in retrieving attributes from an XML document. To surmise, pseudocode for the XML parser is given in Fig. 8.

Algorithm 1 XML Parsing

```
1.    Sequence, NodeList ← {}
2.    Parse XML document
3.    NodeList ← getElementsByTagName("*")
4.    for each Element in NodeList do
5.        Letter ← {}
6.        for each Attribute in Element do
7.            if Attribute is not empty then
8.                Letter ← Attribute.getName() + Attribute.getValue()
9.            end if
10.       end for
11.       Sequence ← Letter + Sequence
12.   end for
```

Figure 8 - XML Parsing Pseudocode

Infinite Alphabet Automata

There are numerous ways of extending automata to the infinite alphabet setting, namely Linear Temporal Logic + freeze [15], Register Automata [16,17,18] and Fresh-Register Automata [19]. As the extension made to the automata is tailored to the problem it solves, some types of automata are not relevant to this project. For example, LTL + freeze normally corresponds to alternating one-register automata [21] which gives rise to complicated verification procedures with high complexity. For the most part, this would be overcomplicated with respect to the types of properties we would wish to verify, and simpler model can be just as effective in this scenario.

Clearly, we must know the problem more intimately if we are to adapt our current automata successfully; starting with the definition of data words. A data word D is a sequence of *data letters L* (where U is the union operator) such that $D = L_i \cup L_{i+1} \cup \ldots \cup L_k$ where L_k is the last data letter created from an element node. A data word has the form: <(E.Name, open)>(A.ID[i], A.V[i])(A.ID[i+1], A.V[i+1])…(A.ID[k], A.V[k])<(E.Name, close)> Where E.Name is the name of element E (label from a finite alphabet), A.ID[i] is the ID of the i^{th} attribute, A.V[i] is the value of the i^{th} attribute (a data value from an infinite alphabet) and k is the last attribute of element E. A data letter is one member in the sequence of pairs of attribute IDs and values, i.e., a data letter is (A.ID[i], A.V[i]). This allows the relation of several attributes to a single element, a format specifically chosen to mimic the structure of the DOM tree.

It has been shown that equivalence relations can be imposed on separate letters (known as positions) within a data word by adding memory to a finite-state machine [16, 17, 18]. The memory can be written to by the automaton and is used to store a copy of the current symbol (or input) which can be compared with a new symbol. A limitation of register automata is that "the only allowed operation on registers (apart from assignment) is a comparison with the symbol currently being processed" [17]. Principally, the old symbol (input at some position a that has been stored in a register) is tested for equality with the new value (input at some position b where 0<a<b). For example, in the string "3343" with 1 register (storage medium) it is possible to have 3 different sets of equivalent values (known as equivalence classes): {0,1}, {0,3}, and [1,3}, where the numbers 0-3 represent the position within the input string, otherwise known as the index. However, with 2 registers there is only one equivalence class: {0,1,3}, where we ignore values that are only equivalent with themselves i.e., 4. The cardinality (size) of an equivalence class is bounded above by r+1 - where r is the number of registers – as the value used for equality checks does not have to be present in a register to be included in an equivalence class. Note that while symbolic (symbols at specific indices in an input string) comparison operations are not allowed when testing XML attribute values (as we will be comparing entire strings) the principle behind the example (using equality checks) is the same.

As detailed in Chapter 1, there exists a transition function that determines when an automaton should change state. This transition function is usually different for each 'type' of automaton (DFA, NFA, etc.), and that which follows is an intuitive description of the transition function δ of a register automaton: When processing a string, an automaton compares the value at the current position with values in its registers. It can then decide on its next course of action by considering this comparison, its current state and the current position (index) itself. The possible actions are storing the current data value in some register and transitioning to another state. An equivalent formalism that fits with data words is given on p43 of [16]. Theoretically, by restricting the transition function of the register automata, one is specifying a property that is then verified by the automaton.

Register automata can be generalised by adding a freshness quality: “an automaton can now accept (and store) an input name just in case it is fresh in the whole run” [19]. This is the equivalent of saying ‘if the automaton has seen this input before, do x, otherwise do y.’ This quality greatly extends the usefulness of the verification routine: the onus has been shifted from the value of the input symbol, to whether it has been seen before. This layer of abstraction on the input entails that the verification routine could be used with an alphabet formed of multiple input types (images, symbols, strings, etc.). Or, more generally, it could be used for more than just XML verification, leading to possible future extensions of the project (discussed in Chapter 7).

This ‘global-freshness’ quality can be used for dynamic name creation [20], something that is not within the scope of this project. We modify this quality to a concept of local-freshness: if an input symbol is not currently held in registers, it is locally fresh. It has been shown [20] “that freshness does not affect the complexity class of the problem” when comparing fresh-register automata to register automata via bisimulation; a binary relation amongst state transition systems, relating systems that behave similarly in the sense that one system simulates the other and vice versa. Moreover, “an RA is an FRA with no globally fresh transitions” [19], hence the automata that we will use for XML property verification is the equivalent of a register automata (one without freshness), but one that can be extended to make use of globally fresh transitions should the need arise. What we have now is a set of four types of transitions in a locally-fresh register automaton:

1. The input symbol is fresh and is written to a register, but the state remains unchanged.
2. The input symbol is fresh, is written to a register, and there is a change in state.
3. The input symbol is not fresh, is written to a register, but the state remains unchanged.
4. The input symbol is not fresh, is written to a register and there is a change in state.

These transitions would not be possible without nondeterminism, as there is no way of performing transitions 1&2 or 3&4 deterministically as they require multiple transitions over the same input.

The culmination of this chapter is a formal definition of the fresh-register automata to be used in this project; a fresh-register automata of k registers is a quintuple such that:

- S = A finite set of states
- $S_0 \in S$ = The start state
- A = The set of data words
- $k_0 \in Reg_K$ = The initial register assignment of the k registers
- $F \in \{True, False\}$ = The freshness quality
- $\mu : S \rightarrow \{1, 2, ..., k\} \times F$ is a partial function from S to $\{1, 2, ..., k\} \times F$ called the *reassignment*. The intuitive meaning of this function is as follows: If automaton A is in state s, $\mu(s)$ is defined, and the input symbol appears in no register, then A writes the input symbol into the $\mu(s)^{th}$ register. Else, if automaton A is in state s, $\mu(s)$ is defined, and the input symbol appears in some register, then A writes the input symbol into one of the k registers.
- $\delta \subseteq S \times \{1, 2, ..., k\} \times F \times S$ is the transition function. The intuitive meaning of which is as follows: If automaton A is in state s, the input symbol is present within the k^{th} register (defined by F = True), and $(s, k, t) \in \mu$, then A may enter state t. In addition, if the input symbol appears in no register (defined by F = False) and is placed into the k^{th} register ($k = \mu(s)$), then to enter state t the transition relation must contain (s, k, t). That is, the reassignment is made prior to a transition.
- $F \subseteq S$ = The set of accepting states.

Where we define:

- $n(xy) \in Z$ as the index of a data letter y in the data word x.
- $[n] = [1, 2, 3, ..., n(XY)]$ where n(XY) is the last data letter in the last data word.
- $Reg_K = \{k : [n] \rightarrow A \cup \{\#\} \mid \forall i \neq j, k(i) = k(j) \Rightarrow k(i) = \#\}$
- A_x as the data word $x \in A$ where $n(x) \in Z$
- A_{xy} as the data letter $xy \in A_x$ where $n(xy) \in Z$

Chapter 3: Software Design & Development

As described in Chapter 5, the project took an agile approach to software development. This allowed the implementation to be completed and alterations to the design to be made if problems were discovered without having to re-design the whole project, avoiding delay to completion. This chapter is subdivided into seven sections. The implementation of the automaton took relatively many iterations to finish because it was more convenient to build a DFA and add to it, than to build the final automaton in one take. It also meant that errors/bugs could be found and dealt with in each stage, rather than discovering a whole host of problems at the very end of implementation. The project was developed with elements of the Extreme Programming (XP) management approach, ensuring that implementation was completed in conjunction with testing, allowing faults in the program to be discovered and corrected rapidly. Eclipse was used as the IDE for its project folder structure as well as its ability to execute and compile code, which simplified the task of separating different automaton implementations into different executables. The subheadings (shown in bold when referencing a specific file) relate to this folder structure: showing the name of the folder, followed by a dash and then the name of the class.

XML Parser - Version 1

As discussed in Chapter 2, the XML parser will produce a sequence of data words from the element and attribute tags of an XML document. These data words would serve as the alphabet of the automaton. The implementation of the parser was relatively straightforward at this stage, as it consisted of the steps detailed in **Algorithm 1**. The relevant code can be found in **CS310 - xmlParser.**

Deterministic Finite Automata

The model chosen to act as the foundation of the project is shown in Figures 4 & 5. It was chosen for its simplicity, and because it would be straightforward to extend.

The design for the DFA consisted of four classes:

- State: Used to define is a state was accepting or not.
- Transition table: The transition table of the automaton.
- Pair: Defined the structure of a single transition.

- Automaton: Used all the above classes, simulated computation, and gave an output.

DFA - State

The 'State' class, used to store a single variable (used to dictate whether a state is in the set F of final states) that is initialised when the constructor method is called, and accessed with the isFinish method.

DFA - Pair

The 'Pair' class was created to represent instances of input and state combinations, (i.e. state q1 sees input symbol 0) and was created as a compact way to store these combinations as well as being able to access the two constituent parts. Generics were used here because the pair class could be used to put any two items together (in case it was needed in the future) and it did not seem consistent to limit the type that an input could take, considering that the fresh-register automaton could be used to compare inputs of two different types (i.e. an image and a string of characters). Finally, this class stored the combined hash code [23] of its constituent entities (explained below).

DFA - TransTable

The 'TransTable' class, created to mimic the operation of a transition function whilst maintaining a pointer to the start state of the automaton. This was implemented by storing the transitions in a hash table [24] which has a format like Fig. 5. The hash table stores values in key, value pairs, where each key corresponds to one value. The key is generated by the pair class, and the value is which state the transition should return. By implementing transitions this way, the deterministic nature of the automaton is captured succinctly and efficiently, as the get, put, and remove functions can yield time complexity equivalent to constant time.

DFA - Automaton

The 'Automaton' class. This class is essentially the summation of the first three classes, with a loop that handles the computation. Upon reaching the end of the input string, the automaton then decides whether the input string is part of its language, and outputs to the console appropriately.

There were multiple reasons for splitting the automaton into several classes: Firstly, it made each class reusable and extendable. Secondly, if there were any errors in the system they could be (usually) traced to a single point of failure - much better than having a cascading plethora of errors stemming from an unclear source. Finally, it modelled the underlying dependencies of the system clearly; this meant that it would be easier to make extensions to the system in the future. Whilst on the topic of extensions, it was decided at this point that the XML parser would be integrated into the final model of the automaton; as it made the most sense for testing purposes.

Difficulties encountered during this stage of implementation were instigated by a misunderstanding of the hashing function as well as inexperience with String to int data type conversions. To elaborate, instead of using the hashing function on each constituent piece of the pair class and summing the result, I tried using it on the pair itself to no avail. Furthermore, I did not realise that using the charAt function to convert a digit from a string would not return the integer value. After some debugging, I found that I needed to make use of the Character.getNumericValue function to return the proper value. Another solution of this would have been to change the input type from int to String, and use letters instead of digits.

Whilst managing to implement a DFA was a step in the right direction, there was still much to be done before the automaton was capable of verifying properties. Primarily, the automaton needed to be able to deal with nondeterminism. Moreover, it needed some form of memory system to be implemented.

Nondeterministic Finite Automata

Figure 6 is an example of an NFA, one which was to be implemented by extending the finished DFA in the following ways:

- To achieve nondeterminism, there should be a way for one transition to lead to two states.
- Nondeterminism must be dealt with deterministically, so there needs to be a way to simulate this.
- Epsilon transitions can be removed with by computing ϵ-closure [31]

To perform these extensions, no new classes are needed; ones already present were modified to encapsulate the new model.

NFA - State

Each state object now has an additional ID attribute that was used to store the name of the state and a method to retrieve this ID.

NFA - TransTable

The changes made to the TransTable class are that for every key, the related value is now a pair of states. If both states in the pair are the same, then this is the equivalent of a deterministic transition. Hence, we have a nondeterministic transition when the two states in the pair are not the same. Also, the epsilon closure of Fig. 6 has been computed, and states and their relevant transitions have been changed accordingly; states q2 and q3 are now one joined state (named q2q3), looping to itself on seeing an input of 0 and moving to state q4 on seeing an input of 1.

NFA - Automaton

The changes made to the Automaton class illustrate how nondeterministic transitions are dealt with. To begin with, the IDs of the left and right members in the pair are compared. If they are the same, then we conclude that the transition is of a deterministic nature and process it in the same way as the DFA would. If they are not the same, then the transition is of a nondeterministic nature and an actual NFA would create a copy of itself to take both routes; where one NFA executes the transition dictated by the left member in the pair, and the other NFA takes the right transition. As an alternative to this, the user is asked which transition they would like to take; the console is written to with the choice of two transitions (each of which are numbered by the order that they appear in) and a visualisation of what the transition will do in terms of whether the current state will stay the same, or if the automaton will enter a new state (hence the state class having the ID attribute). The user then inputs the corresponding number on their keyboard (handled by the Scanner class [25]), presses enter, and the automaton performs their chosen transition. This process happens for every nondeterministic transition and is the equivalent of simulating nondeterminism at the whim of the user.

Completing the NFA meant that the system could now deal with nondeterministic transitions, which are essential for register automata (as explained in the research

chapter). The next important milestone is to add an implementation of memory to the current model to make it into a register automaton.

Fresh-Register Automata

Figure 9 (below) shows an example of a fresh-register automaton (FRA); whose transition function has the following intuitive meaning: For every fresh input that is seen (denoted by "*") write it into a register specific to the current state and either stay in the same state, or move into the new state. If the state remained the same and the next input is not fresh, move to the accepting state. This example of fresh-register automata would accept an input that contained an equivalence class of cardinality up to 2. This expression serves as a part of the language that would be used for XML property verification, and hence implementing this automaton would be a step towards creating the model for the final automaton.

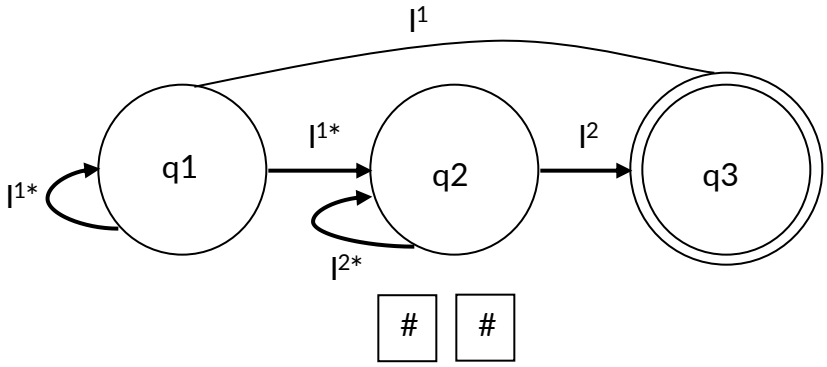

Figure 9 - Example Register Automaton

A few changes need to be made to the NFA to turn it into a FRA, this primarily involves the inclusion of a register class as well as modifications to both the TransTable and Automaton classes. The register class will instantiate some form of 'memory' of variable size, which can be written to, overwritten when needed, and whose contents can be compared.

The Array [26] data structure was chosen to implement the register for several reasons:

- Extremely efficient insertion and removal times (constant time complexity).
- Efficient comparison times (linear time complexity).

- The number of registers needed is static as per the definition of each automaton; this matches the static initialisation of the Array.

NFRA - TransTable

Each key to the hash map can now access a pair of states, which deals with nondeterministic transitions as well as deterministic transitions. The accepting state is kept track of with a variable and related 'get' function (explained in more detail later).

NFRA - Register

The size of the Array is defined when the constructor function is called, and each index of the Array is initialised to null (equivalent to # in the diagram). An Array can be written to if (and only if) the correct register is selected, and the value is fresh. The register class handles freshness by calling a function that checks whether the current value exists within any of the registers; returning false if so and true if not.

NFRA - Automaton

This section describes the changes made to accommodate registers into the Automaton class. The first change is that the user is only asked to choose a nondeterministic transition if the input value is found to be fresh. This is done by surrounding the interaction block (in which the user is asked to input their choice) in a statement which tests if the input value was added to a specific register (1 if in the first state, 2 if in the second state) successfully. If it was not successfully added, then the value already exists in registers and the automaton moves to the accepting state. This is done by setting curState = finalState, and is why the final state is defined explicitly.

However, if it was successfully added, then the user makes their choice. If they decide to make the transition that changes the state of the machine, then the machine loops through the input deterministically (i.e. no user input required) until the end of the input string. This is done by creating a second control loop, where input values are stored in the second register if they are fresh. If they are not fresh, then the loop terminates and the automaton moves to the accepting state. After the input string is processed, the user is told whether it contained a duplicate of the value they selected or not; accepting or rejecting respectively.

I met some difficulties when implementing the Automaton class, most of which concerned when to increment the loop counter. Usually with a for loop, one would

increment the loop counter at the end of every complete execution of the loop. This was not possible, as the complete execution of the first loop depended on the outcome of the second loop. I found where to increment the loops by analysing Fig. 9, and paying careful attention to the logic that it entailed:

- If a duplicate was found in the second loop, then the second loop must terminate both itself and the first loop and give an accepting result.
- If no duplicate was found, then the second loop must terminate itself and the first loop as well as, giving a rejecting result.

The first loop exists so that the user can pick and choose which input value they would like to verify. So, if they wanted to, they could iterate through the entire input string. This meant that the automaton would need to accept if two equivalent values were next to each other within the string, and reject if otherwise.

The problem that had to be faced now was that the register automaton that had just been implemented worked for one property, but needed to be able to accept any property. This meant that the automaton class needed to be redesigned to be more abstract, and would hence work for any property. This implied being able to work with a variable number of registers.

Generic Fresh-Register Automata:

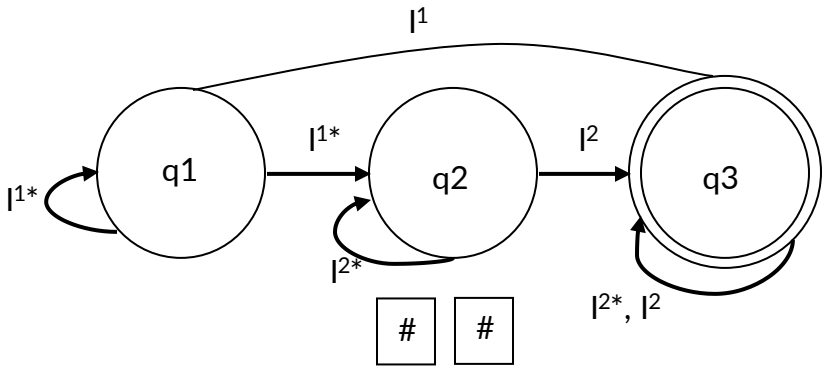

Figure 10 - Example Generic Register Automaton

The difference between Figs. 9 & 10 is that once the automaton has reached the accepting state(s), it will continue to process (and hence overwriting its registers) but will remain in the accepting state(s). Whilst this is not a substantial theoretical leap from the fresh-register automaton that has been developed, it does simplify the implementation somewhat; the method of having two (or more) loops can be replaced

by having one loop that ends once the last input symbol has been processed. The design for the Register, TransTable and Automaton classes were changed in the following ways:

- To write to a register, the input no longer needs to be fresh. Registers themselves needed to be able to store more than just integers, so they now store Objects.
- Keys (to store transitions) are now generated from pairs consisting of a state and freshness, where freshness = true/false.
- Transitions are now a quadruple: source state, sink state, Freshness, Register. This means that every detail concerning an action carried out by the automaton is contained within the transition, meaning that all the automaton needs to do is read these transitions to function.
- The structure of the automaton needs to be changed to better represent its generality. For instance, no longer inserting values into hard-coded registers.

GNFRA - Register

The registers now store Objects, do not check for freshness when inserting data into a register, and the method of checking for freshness has been altered for the new data type by using the toString [27] method:

The object stored in the register is converted into a String of equivalent value, the input value is converted into a String of equivalent value and the two Strings are then compared. As every Object has a toString method, this should be a very consistent technique to compare input values.

GNFRA - Quad

The 'Quad' class; essentially an extension of the pair class, where it contains four values instead of two and has 'get' methods for each. This class does not generate keys, however, as there is no need for it to do so.

GNFRA - TransTable

The TransTable class now has a different structure. Firstly, where in **NFRA - TransTable** the keys were generated by using a pair consisting of (State, Value) the keys are now generated using pairs consisting of (State, Freshness). This means that the emphasis moves from the value of the input to whether it has been seen before.

Secondly, the transitions were changed to be quadruples, and were grouped together in ArrayLists [28] as they are essentially dynamically sized Arrays, with the following rules to simplify processing them:

- Transitions are put into the same ArrayList if their Source state is the same.
- All fresh transitions must come before stale (not fresh) transitions.

GNFRA - Automaton

The differences between the generic automaton and the fresh-register automaton were simple in theory, but required a complete restructure of the automaton class. The changes made were to the following intuitive effect:

- The automaton now does all its computation within one loop that ends once the final input value/symbol has been processed.
- The number of transitions that are relevant at any one time depends on the current state and whether the input is fresh or 'stale' (i.e. not fresh). For example, if we are looking to count transitions involving fresh values, we look for transitions in which the freshness quality is "true" ("false" if we were looking for stale transitions), incrementing a counter when applicable.
    - If the number of transitions is 1, execute it deterministically and move to the sink state found within the transition.
    - Else, knowing how many transitions there are, and that they are all one after the other, we can loop through the transitions and display them to the user. This is the equivalent of a nondeterministic transition.
    - Once the user has selected a transition, execute it and reset all counters to zero.

As this algorithm is only dependent on the transition table, the simulation routine will execute any combination of transitions given in the correct format. However, the final change in design is focused on how the transition table is specified. Presently, it is hard coded into the automaton, which requires the implementation to be modified to change the transition table. We seek to change this so that the transition table, states and number of registers can be specified outside the confines of the automaton. Also, as this is the last major change in design of the automaton, it is time to incorporate the XML parser. This means that the output from the parser will be used as the alphabet

for the automaton. These alterations will mean that the automaton will be able to process any transition table specified by the user, essentially meaning that the user will be able to specify properties that an XML file should have, and the automaton will read the XML file, parse it, and verify the property.

XML Parser - Version 2

Currently, the XML parser generates data words that are a sequence of data letters. Theoretically, this is what the automaton will use as its alphabet. Realistically, the data words will be broken down into data letters so that individual pairs of attribute ID and value can be stored and hence compared. The input currently has the following shape: <(**E.Name,** open)>(**A.ID**[i], **A.V**[i])(**A.ID**[i+1], **A.V**[i+1])…(**A.ID**[k], **A.V**[k])<(**E.Name,** close)> which is necessary, as it relates a sequence of attributes to their corresponding element tag. However, this relationship will not be used in practice. Therefore, the question we need to ask is: how do we obtain each data letter? Thanks to the structure of the design, separating data letters is just a matter of finding when each data letter starts and finishes. This results in three main cases:

- Case 1: The first data letter will always begin after “>(”.
- Case 2: The last data letter will always end before “)<“.
- Case 3: When there are multiple data letters in the same sequence, they will be separated by “)(“.

The following is a pseudocode design for the algorithm that will be used to convert data words into sequences of data letters.

Algorithm 2 Data Letter Parsing

```
1.    Input, Index, Letters, dataWords ← {}
2.    dataWords ← Sequence from Algorithm 1
3.    Input ← dataWords.toCharArray
4.    for each character in char array do
5.        if sequence of characters matches case 1 then
6.            Index ← position of ")>"
7.        end if
8.        if sequence of characters matches case 2 then
9.            Index ← position of "<("
10.       end if
11.       if sequence of characters matches case 3 then
12.           Index ← position of ")("
13.           Index ← position of ")("
14.       end if
15.   end for
16.   for each pair of entries in Index do
17.       Letters ← dataWords.subString(index 1, index 2)
18    end for
```

Figure 11 – Pseudocode for Data Letter Parsing Algorithm

USNFRA - xmlParser

The main difference between design and implementation of this algorithm is how cases one through three were specified:

- Case 1: Two nested if statements, the first checked whether the character in question was ">", the second verified that ">" was not the last character in the input string and that the next character was "(".
- Case 2: Two nested if statements, the first verified whether the character in question was ")", the second checked whether the next character was "<".
- Case 3: Two nested if statements, the first checked if the character in question was ")", the second verified that the next character was "(".

The Property Verifier

A small change needs to be made to the Automaton class such that it makes use of the output of xmlParser, this has the effect of making the alphabet of the automaton consist of data letters instead of strings of digits. The last step was to redesign the

TransTable class so that it parsed a file that contained the specification of a transition table, including the names of all states and the number of registers required.

Fig. 12 illustrates a file consisting of one such specification. The first line is a series of comma separated integers; the first of which is the start state, and each integer after the first is a final state. The second line is where the user specifies how many registers the automaton should make use of, and hence the line must contain one integer. The lines that follow are all transitions specified as comma-separated quads.

```
1, 3
3
1, 1, true, 0
1, 2, true, 0
1, 3, false, 2
2, 2, true, 1
2, 3, false, 1
3, 3, true, 2
3, 3, false, 2
```

Figure 12 - Example Specification of a Transition Table

The following pseudocode algorithm was designed to parse the specification file:

Algorithm 3 Specification File Parsing

```
1.    CurState, Registers, FinalStates, Transitions, Commas, L, ML, MR, R ← {}
2.    Read specification file
3.    for each line in the file do
4.        if on the first line then
5.            Count number of commas on the line
6.            CurState ← first integer on line
7.            FinalStates ← Every subsequent integer
8.        end if
9.        if on the second line then
10.           Registers ← integer on the line
11.       end if
12.       Quad ← {}
12.       Commas ← index of every comma on the line
13.       L ← integer between start of line and first comma index
14.       ML ← integer between first and second comma indices
15.       MR ← string between second and third comma indices
16.       R ← integer between last comma index and end of line
17.       Quad ← L, ML, MR, R
18.   end for
```

Figure 13 - Specification-File Parsing Algorithm

USNFRA - TransTable

The final version of the TransTable class stores the initial state of the automaton, the possible final states of the automaton (integers stored in an Array), the number of Registers needed, and an ArrayList of quads i.e. the transitions parsed. All the stored data is accessed by using public methods; methods callable by any class that instantiates the TransTable class. It makes use of a BufferedReader [29] to read the specification file line by line.

Looking at **Algorithm 3** in more detail: Line 5 is implemented by converting the first line in the specification file into an Array of characters; looping through it to find instances of ",". Line 6 is realised by taking a substring between the start of the line and the first instance of "," (done by using the indexOf function). Line 7 was more challenging, and required the implementation of an algorithm that removed preceding white space from a string of characters. To parse the comma-comma separated list of final states, the indices of all the commas in the line were stored in an array. Then, a substring of everything between each pair of commas returned the digits that represented final states. However, the last digit that needed to be parsed required a substring between the index of the last comma and the end of the line (white space removed). Lines 10, and 12 through to 16 copied this method of retrieval, storing the data gathered in the appropriate variable.

USNFRA - Automaton

The only way in which this class changed was by calling the xmlParser class to parse an XML file and looping through its output instead of looping through a hard-coded string of integers. The reason this class needed to change so little was because the previous automaton (**GNFRA - Automaton**) had been designed to compute transitions formatted into the structure specified by the Quad class; this matches the way transitions are specified by the user. One final change is that the State class was removed, as its function was redundant: the list of final states is given in the specification file, and the ID of each state is the same as the ID specified in a transition.

Significant difficulty came from the implementation of **Algorithm 2** & **3**. Specifically, the way in which to keep track of the indexes of multiple commas on the same line. Algorithm 2, initially, was implemented to delete everything up to and including the

index of the next comma, storing the number of deleted characters. When the digit had been parsed, the index of the next comma would be incorrect as the line was now x digits shorter, but for some reason adding the number of removed characters to this number did not work. I worked around this by using the substring method instead, grabbing the string located between the indices of two commas whilst making sure not to include the comma itself.

With the completion of this class' implementation, the automaton can now read in any XML file, produce a sequence of data words from a list of element tags containing attributes, parse a list of transitions from a specification file and simulate a fresh-register automaton with k registers that verifies a property. The property expressed in Fig. 12 is 'accept if there exist duplicate attributes'.

Chapter 4: Testing

As mentioned previously, the software development methodology used throughout the project made use of features innate to the Extreme Programming agile method, where testing occurs in combination with implementation. Implementation was broken down into multiple phases with testing interleaved throughout each phase in the form of unit (individual) and component (interlaced units) tests, with system tests occurring at the end of each phase of implementation. This chapter details the testing carried out on the software created for the verification of XML documents. Please note that the tables included in this chapter do not include the full range of tests, as tests are often repeated with different inputs to ensure accuracy. Therefore, tests will be labelled with general inputs (unless specific cases need to be tested) to show the breadth of the test plan.

XML Parser - Version 1

The testing of this phase was carried out to ensure that the class parsed an XML document correctly, and produced the appropriate output. As this class is used to create the alphabet of the final automaton, it was paramount that it functioned properly as an error in this part of the system would cause errors in later parts of the system.

Unit Tests

These tests are carried out on specific parts of the class to ensure correct functionality throughout development.

ID	Description	Input	Expected Output	Actual Output	Result
1	Ensuring successful retrieval of XML elements	Books.xml	List of all elements present within XML file	List of all elements present within XML file	Pass
2	Ensuring successful retrieval of XML attributes	List of all elements within XML file	List of all attributes present within XML file	List of all attributes present within XML file	Pass
3	Ensuring successful creation of data letters	List of all attributes present within XML file	List of data letters spanning all XML attributes in file	List of data letters spanning all XML attributes in file	Pass
4	Ensuring successful creation of data word	List of data letters created from "XML Developer's Guide"	<Book, open>(cat,action)(id,bk101)<Book, close> OR <Book, open>(id,bk101)(cat,action)<Book, close>	<Book, open>(cat,action)(id,bk101)<Book, close>	Pass
5	Ensuring successful creation of data words	Books.xml	Sequence of data words spanning every element which had attributes	Sequence of data words spanning every element which had attributes	Pass

Table 1 - Unit Tests (XML Parser - Version 1)

Component Tests

As the XML parser would be instantiated from within another class, it was imperative that the results from the unit tests were consistent with the resulst from multi-class component tests.

ID	Description	Input	Expected Output	Actual Output	Result
1	Ensuring consistent behaviour when used in the confines of another class	xmlParser.parse(Books.xml)	Sequence of data words spanning every element which had attributes	Sequence of data words spanning every element which had attributes	Pass
2	Ensuring correct behaviour when XML file specified incorrectly	xmlParser.parse(boks.xml)	FileNotFoundException	FileNotFoundException	Pass
3a	Ensuring correct behaviour when XML file is not well formed	xmlParser.parse(nwfBooks.xml)	SAXParseException	SAXParseException	Pass
3b	Ensuring correct behaviour when XML file is empty	xmlParser.parse(emptyBooks.xml)	Premature end of file error	Premature end of file error	Pass
3c	Ensuring correct behaviour when no attributes contained in the class	xmlParser.parse(noAttBooks.xml)	Empty String	Empty String	Pass

Table 2 - Component Tests (XML Parser - Version 1)

Deterministic Finite Automata

The testing of this phase was carried out to ensure that the system simulated a deterministic finite automaton properly when given an input consisting of a string of digits, and produced an appropriate output. As this system was to be the foundation for all the varying models of automata to be developed, it was paramount that it functioned correctly as an error in this part of the system would cause errors in later versions.

Unit tests

These tests are carried out on individual classes (which combined to make the system) to ensure correct functionality throughout the development of the DFA. Any classes that are copied between versions of the system (i.e. from DFA to NFA) will not be re-

tested unless the class in question has been changed. The tests outlined in Tables 3-7 are not the full set that were used, however, the tests outlined here were those necessary to cover all functionalities of the respective class.

State

ID	Description	Input	Expected Output	Actual Output	Result
1	Successful instantiation of state	State q1 = new State(false)	No Error	No Error	Pass
2	Successful instantiation of state	State q2 = new State(true)	No Error	No Error	Pass
3	Successful retrieval of data stored in State	q1.isFinish()	False	False	Pass
4	Successful retrieval of data stored in State	q2.isFinish()	True	True	Pass

Table 3 - Unit Tests (DFA/State)

Pair

ID	Description	Input	Expected Output	Actual Output	Result
1	Ensuring successful storage of two pieces of data	State, Integer (q1, 0)	366712642 (0 + 366712642	366712642	Pass
2	Ensuring two different pairs have distinct keys	(q1, 0) & (q1,1)	366712642, 366712643	366712642, 366712643	Pass
3	Ensuring four different states have distinct hash codes	q1, q2, q3, q4 q1 = false, q2 = true q3 = false q4 = true	Four different hash codes	366712642, 1829164700, 2018699554, 1311053135	Pass
4	Ensuring correct retrieval of left element in pair	(q1, 0)	q1	q1	Pass
5	Ensuring correct retrieval of right element in pair	(q1, 1)	1	1	Pass

Table 4 - Unit Tests (DFA/Pair)

Unit & Component tests

The next two classes that were tested (TransTable & Automaton) made use of all the previous classes, hence, their unit tests also acted as component tests for the system. These tests were carried out to ensure that the program simulated a DFA correctly.

TransTable

ID	Description	Input	Expected Output	Actual Output	Result
1	Successful instantiation of state	State q1 = new State(false)	No Error	No Error	Pass
2	Successful instantiation of state	State q2 = new State(true)	No Error	No Error	Pass
3	Successful retrieval of data stored in State	q1.isFinish()	False	False	Pass
4	Successful retrieval of data stored in State	q2.isFinish()	True	True	Pass
5	Successful instantiation of pair	Pair<State, Integer> t1 = new Pair<State, Integer>(q1, 0);	No Error	No Error	Pass
6a	Successful retrieval of left element in pair	t1.getLeft()	q1	q1	Pass
6b	Successful retrieval of right element in pair	t1.getRight()	0	0	Pass
6c	Successful retrieval of key generated by pair	t1.getKey	366712642	366712642	Pass
7a	Successful storage of Key, Value pair in hash map	transTable.put(k1, q1)	No Error	No Error	Pass
7b	Successful query of Key, Value pair in hash map	transTable.containsKey(k1)	True	True	Pass
7c	Successful retrieval of Value in hash map using appropriate Key	transTable.get(k1)	q1	q1	Pass

Table 5 - Unit/Component Tests (DFA/TransTable)

Automaton

ID	Description	Input	Expected Output	Actual Output	Result
1	Ensuring successful instantiation of TransTable	TransTable transTable = new TransTable(); transTable.initTransTable();	No Error	No Error	Pass
2	Ensuring correct retrieval & assignment of CurState	State curState = transTable.getCurState();	No Error	No Error	Pass
3	Ensuring correct character retrieval from input string	String in = "01100"	0, 1, 1, 0, 0	0, 1, 1, 0, 0	Pass
4	Ensuring correct pair generation from curState & input character	curState = q1, input = 1	q1, 1	q1, 1	Pass
5	Ensuring successful key generated from pair	Pair consisting of q1, 1	366712643	366712643	Pass
6	Ensuring correct functionality of the transTable.get(key) function	transTable.containsKey(366712643)	True	True	Pass
7	Ensuring successful retrieval of state from transTable	transTable.get(366712643)	q2	q2	Pass
8	Ensuring successful retrieval of state information	curState.isFinish() when curState = q1	False	False	Pass
9	Ensuring successful retrieval of state information	curState.isFinish() when curState = q2	False	False	Pass
10	Ensuring Automaton accepts when input is valid	String in = "01100"	Valid Input	Valid Input	Pass
11	Ensuring Automaton rejects when input is invalid	String in = "0110"	Invalid Input	Invalid Input	Pass

Table 6 - Unit/Component tests (DFA/Automaton)

DFA System Tests

This final set of tests seeks to ensure that the automaton behaves correctly within this stage of implementation by testing it with borderline/extreme cases to try and produce erroneous results.

ID	Description	Input	Expected Output	Actual Output	Result
1a	Ensuring that the automaton doesn't simulate erroneous input strings	Empty string	Invalid Input	Invalid Input	Pass
1b		"abcdef"	Invalid Input	Invalid Input	Pass
1c		"abc01100def"	Invalid Input	Invalid Input	Pass
1d		"0a1b1c0d0e"	Invalid Input	Invalid Input	Pass
1e		"01100abcd"	Invalid Input	Invalid Input	Pass
2a	Ensuring that the automaton accepts input strings within the language	"011"	Valid Input	Valid Input	Pass
2b		"001100"	Valid Input	Valid Input	Pass
2c		"0001100100"	Valid Input	Valid Input	Pass
3a	Ensuring that the automaton rejects input strings not within the language	"0"	Invalid Input	Invalid Input	Pass
3b		"0110"	Invalid Input	Invalid Input	Pass
3c		"0101010"	Invalid Input	Invalid Input	Pass

Table 7 - DFA System Tests

Nondeterministic Finite Automata

Three components of the DFA were changed to produce the NFA; State, TransTable and Automaton. The new features in these three classes need to be tested to ensure that they function as intended.

Unit Tests: State

ID	Description	Input	Expected Output	Actual Output	Result
1a	Ensuring correct instantiation of State	State q1 = new State (false, "q1");	No Error	No Error	Pass
1b		State q2 = new State (true, "q2");	No Error	No Error	Pass
2a	Ensuring correct retrieval of State data	q1.getID()	"q1"	"q1"	Pass
2b		q2.getID()	"q2"	"q2"	Pass

Table 8 - Unit Tests (NFA/State)

Unit/Component Tests: Automaton

ID	Description	Input	Expected Output	Actual Output	Result
1a	Ensuring correct key generated by pair	(q1,1) Where q1 = (false, "q1")	1442407170 + 1	1442407171	Pass
1b		(q1, 1) Where q1 = (true, "q1")	1442407170 + 1	1442407171	Pass
2a	Ensuring successful comparison of states	Two States q1 & q2 are different	False	False	Pass
2b		Two states q1 & q2 where q1=q2	True	True	Pass
3a	Ensuring that the choice made by user is received correctly	1	TransTable.get(key).getRight()	TransTable.get(key).getRight()	Pass
3b		2	TransTable.get(key).getLeft()	TransTable.get(key).getLeft()	Pass
4a	Ensuring that the automaton accepts input strings contained in the language	01	Valid Input	Valid Input	Pass
4b	Ensuring Automaton rejects input string not contained within the language	00	Invalid Input	Invalid Input	Pass

Table 9 - Unit/Component Tests (NFA/Automaton)

NFA System Tests:

The whole system is now rigorously tested with extreme/borderline tests to verify that it operates as intended.

ID	Description	Input	Expected Output	Actual Output	Result
1a	Ensuring automaton rejects erroneous input	"abcde"	Invalid Input	Invalid Input	Pass
1b		"1a0b0c1d"	Invalid Input	Invalid Input	Pass
1c		"1001abcd"	Invalid Input	Invalid Input	Pass
1d		"Abcd1001"	Invalid Input	Invalid Input	Pass
1e		"10201"	Invalid Input	Invalid Input	Pass
1f		Empty String	Invalid Input	Invalid Input	Pass
2a	Ensuring all paths of an input can be reached	"1001"	Final state q1 (Invalid Input) q1, q1, q1, q1	Final State q1 (Invalid Input)	Pass
2b			Final state q4 (Valid Input) q1, q2q3, q2q3, q4	Final state q4 (Valid Input)	Pass
2c			Final state q4 (Valid Input) q1, q1, q2q3, q4	Final state q4 (Valid Input)	Pass
3	Ensuring correct comparison of states	States q1 & q2 where q1!=q2 but ID of both is "q1"	True	True	Fail

Table 10 - NFA System Tests

The system testing of the NFA implementation lead to the discovery of an oversight made by the developer: two inequivalent states would be deemed equivalent if their respective ID's were equivalent. Whilst this behaviour does require erroneous input in the first place, it is still a flaw. The most obvious solution to this problem is to ensure IDs are always distinct. In the final version of the system (USNFRA) the problem is resolved more substantially by having the ID of a specific state equivalent to the state itself, i.e., the state is referenced by its own instantiation.

Fresh-Register Automata

The design of the automata was changed with the introduction of registers; it was imperative for these registers to function consistently as most of the computation done by the simulator required them. This meant that their functionality (storage, comparison) had to be thoroughly tested. The algorithm for simulating the automaton was modified to accommodate the introduction of registers, hence, it needed to be retested to ensure that it continued to perform correctly.

Unit tests: Register (of type Integer)

ID	Description	Input	Expected Output	Actual Output	Result
1a	Verifying the correct initialisation of size	new Register(2);	Register with two indexed locations	Register with two indexed locations, 0 & 1	Pass
1b	Verifying the correct initialisation: null type	new register(1) if (register(0) == null) return true	True	True	Pass
2a	Verifying that the register determines freshness of input correctly	Integer not currently stored in any register	True	True	Pass
2b		Integer currently stored in register	False	False	Pass
3a	Verifying that the register is written to correctly	Input is fresh	Written to register	Written to register	Pass
3b		Input is not fresh	Not written to register	Not written to register	Pass
4a	Verifying that the register accepts correct data type: Integer	Input type: String	Not written to register	Not written to register	Pass
4b		Input type: Integer	Written to register	Written to register	Pass

Table 11 - Unit Tests (FRA/Register)

Unit & Component Tests: Automaton

ID	Description	Input	Expected Output	Actual Output	Result
1	Verifying correct instantiation of TransTable	New TransTable, initialised with correct contents	No Error	No Error	Pass
2	Verifying correct initialisation of Registers	Function call to create new Register object of size 2	No Error	No Error	Pass
3	Verifying correct instantiation of initial state	curState = transTable.getCurState()	curState = q1	curState = q1	Pass
4	Verifying correct assignment of final state	curState = transTable.getFinalState()	curState = q3	curState = q3	Pass
5a	Verifying that fresh input is written to specific register successfully	Register.add(aChar, 0) aChar = fresh	aChar written to register location 0	aChar written to register location 0	Pass
5b		Register.add(aChar, 1) aChar = fresh	aChar written to register location 1	aChar written to register location 1	Pass
6a	Verifying that stale input is skipped	Register.add(aChar, 0) aChar = stale	aChar not written to register location 0	aChar not written to register location 0	Pass
6b		Register.add(aChar, 1) aChar = stale	aChar not written to register location 1	aChar not written to register location 1	Pass
7a	Verifying that control loops iterate as intended	Loop through entirety of input string in first control loop	End of input reached	End of input reached	Pass
7b		Loop through entirety of input string in second control loop	End of input reached	End of input reached	Pass
7c		Skip to end of input on seeing stale character in first control loop	I = 0, 1, 6 (end)	I = 0, 1, 6 (end)	Pass
7d		Skip to end of input on seeing stale character in second control loop	I = 0... (switch to J) J = 0, 1, 2, 6 (end) I = 6 (end)	I = 0... (switch to J) J = 0, 1, 2, 6 (end) I = 6 (end)	Pass

Table 12 - Unit/Component Tests (FRA/Automaton)

FRA System Tests

These tests are carried out to ensure that the FRA functions as intended (accepting and rejecting valid and invalid inputs respectively) whilst also testing the system with borderline/extreme cases in an attempt to produce flawed behaviour.

ID	Description	Input	Expected Output	Actual Output	Result
1a	Verifying that the automaton accepts input string contained in the language	"01201"	0 is not fresh in register 2	0 is not fresh in register 2	Pass
1b		"01201"	1 is not fresh in register 2	1 is not fresh in register 2	Pass
1c		"001201"	0 is not fresh in register 1 (valid input)	0 is not fresh in register 1 (valid input)	Pass
1d		"001201"	0 is not fresh in register 2	0 is not fresh in register 2	Pass
2a	Verifying that the automaton rejects input string contained not in the language	"012"	Invalid Input	Invalid Input	Pass
2b		"01201"	Invalid input (2)	Invalid input	Pass
3a	Verifying rejection of erroneous input strings	"aabbcc"	Invalid Input	Invalid Input	Pass
3b		"a11"	Invalid Input	Invalid Input	Pass
3c		"11a"	Invalid Input	Valid Input	Fail

Table 13 - FRA System Tests

As shown by test 3c, the automaton accepts an erroneous input string where it should technically fail. This is caused by incorrect/missing transitions in the transition table. Specifically, the final state was stored in the transition table and was used to halt the simulation as soon as a duplicate was found. Meaning that input that appeared after a duplicate symbol is never considered. To correct this, the automaton needs to still do computation whilst in the final state, which can be achieved by adding a transition taken on any input, originating from the final state and looping to itself. No changes will be made to this version of the automaton to correct this behaviour, but will be applied to the subsequent version of the automaton instead.

Generic Fresh-Register Automata

A modified version of the FRA, this version of the automaton boasts the ability that it will be able to simulate any set of transitions over an arbitrary number of registers. The GFRA makes use of a new class called Quad.java, and uses modified versions of TransTable, Register and Automaton:

- A register no longer requires fresh input when being written to. Also, registers now store data of "Object" type instead of Integers.
- TransTable makes use of the new class, Quad and has seen a slight change in structure and now makes use of ArrayLists.
- Automaton has seen a complete change in structure.

Testing at this stage is to guarantee that the modifications made to the classes mentioned previously has not introduced any incorrect behaviours.

Unit Tests: Quad

ID	Description	Input	Expected Output	Actual Output	Result
1	Verifying the correct instantiation and assignment of a Quad	Quad<State, State, String, Integer> p7 = new Quad<State, State, String, Integer>(q1, q1, "true", 0);	No Error	No Error	Pass
2a	Verifying the correct retrieval of elements stored in a Quad: Left element	p7.getLeft()	q1	q1	Pass
2b	Mid Left element	p7.getMidLeft()	q1	q1	Pass
2c	Mid Right element	p7.getMidRight()	True	True	Pass
2d	Right element	p7.getRight()	0	0	Pass
3a	Verifying that Quads cannot be assigned incorrectly if instantiated properly: Quad<State, State, String, Integer> p7	p7= new Quad<State, State, String, Integer>("q1", q1, "true", 0);	Syntax Error	Syntax Error	Pass
3b		p7 = new Quad<State, State, String, Integer>(q1, "q1", "true", 0);	Syntax Error	Syntax Error	Pass
3c		p7 = new Quad<State, State, String, Integer>(q1, q1, 12, 0);	Syntax Error	Syntax Error	Pass
3d		p7 = new Quad<State, State, String, Integer>(q1, q1, "true", "false");	Syntax Error	Syntax Error	Pass

Table 14 - Unit Tests (GFRA/Quad)

Register

ID	Description	Input	Expected Output	Actual Output	Result
1	Verification that the Register data structure instantiates correctly	Register register = new Register(2)	No Error	No Error	Pass
2a	Verifying that data is stored in register correctly	Register[0].add("abb")	Register[0] = "abc"	Register[0] = "abc"	Pass
2b		Register[1].add("1223")	Register[1] = "1223"	Register[1] = "1223"	Pass
2c		Register[0].add("a1b2")	Register[0] = "a1b2"	Register[0] = "a1b2"	Pass
2d		Register[2].add("f87d")	Out of bounds error	Out of bounds error	Pass
3a	Verifying that freshness is determined correctly	Register[0] contains "0a1b" Register.isFresh("0a1b")	False	False	Pass
3b		Register.isFresh("0a1c")	True	True	Pass

Table 15 - Unit Tests - (GFRA/Register)

Unit & Component Tests: TransTable

ID	Description	Input	Expected Output	Actual Output	Result
1	Confirming that the Quad data type can be used in generics	ArrayList<Quad<State, State, String, Integer>> c1 = new ArrayList<Quad<State, State, String, Integer>>();	No Error	No Error	Pass
2a	Confirming that Quads can be added to an ArrayList of the same type	Quad<State, State, String, Integer> p7 = new Quad<State, State, String, Integer>(q1, q1, "true", 0); c1.add(p7)	No Error	No Error	Pass
2b		Quad<State, State, String, Integer> p1 = new Quad<State, State, String, Integer>(q1, q2, "true", 0); c1.add(p1);	No Error	No Error	Pass
2c		Quad<State, State, String, Integer> p4 = new Quad<State, State, String, Integer>(q1, q3, "false", 0); c1.add(p4);	No Error	No Error	Pass
3	Confirming that Quads can't be added to an ArrayList that isn't of the same type	Quad<State, State, String, String> p2 = new Quad<State, State, String, String>(q1, q1, "true", "0"); c1.add(p2)	Syntax Error	Syntax Error	Pass
4a	Confirming correct retrieval of Quads from ArrayList	c1.get(0)	p1	p1	Pass
4b		c1.get(1)	p4	p4	Pass
4c		c1.get(2)	p4	p4	Pass
5a	Confirming correct storage of ArrayList in Hash Map	transTable.put(k1, c1) where k1 is 1445976208	No Error	No Error	Pass
b	Confirming correct retrieval of ArrayList from Hash Map	transTable.get(1445976208)	c1	c1	Pass

Table 16 - Unit/Component Tests (GFRA/TransTable)

Automaton

ID	Description	Input	Expected Output	Actual Output	Result
1a	Confirming correct instantiations and assignments: TransTable	Initialise and assign TransTable with initTransTable function	No Error	No Error	Pass
1b	Register	Initialise register with k addressable locations	No Error	No Error	Pass
1c	curState	Initialise and assign curState with transTable.get(curState)	No Error	No Error	Pass
2a	Confirming correct detection of freshness	Character = "a", not currently present in registers	Fresh	Fresh	Pass
2b		Character = "a", currently present in registers	Stale	Stale	Pass
3a	Confirming relevant transitions are retrieved	Key generated matches key in Hash Map	Relevant transitions found	Relevant transitions found	Pass
3b	Confirming automaton rejects invalid input (i.e. when no relevant transitions found)	Key generated doesn't match key in Hash Map	Invalid Input	Invalid Input	Pass
4a	Confirming automaton handles situation where key exists but transitions do not	Key for fresh input	"Key for fresh input found with no transitions"	"Key for fresh input found with no transitions"	Pass
4b		Key for stale input	"Key for stale input found with no transitions"	"Key for stale input found with no transitions"	Pass
4a	Confirming that the number of relevant transitions are counted successfully	Fresh input, State = q1	2	2	Pass
4b		Stale input, State = q1	1	1	Pass
4c		Fresh input, State = q2	1	1	Pass
4d		Stale input, State = q2	1	1	Pass

5a	Confirming that user is asked to choose between multiple transitions	Number of relevant fresh transitions = k	User has choice between transitions numbered from 0 to k	User has choice between 0 & k	Pass
5b		Number of relevant stale transitions = m	User has choice between transitions numbered from k to m	User has choice between transitions numbered from k to m	Pass
6a	Verifying choice made by user is within permitted bounds	Number of transitions = 2, numbered 0, 1	Any input < 0 or >= 2 asks for the user to give a different input	Any input < 0 or >= 2 asks for the user to give a different input	Pass
6b		Number of transitions = 3, numbered 2,3,4	Any input < 2 or > 4 asks for the user to give a different input	Any input < 2 or > 4 asks for the user to give a different input	Pass
7a	Verifying that when only one relevant transition is found, it is executed deterministically	One fresh transition	Fresh transition executed	Fresh transition executed	Pass
7b		One stale transition	Fresh transition executed	Fresh transition executed	Pass

Table 17 - Unit/Component Tests (GFRA/Automaton)

GFRA System Tests

These tests are carried out to ensure that the GFRA functions as intended (accepting and rejecting valid and invalid inputs, respectively) whilst making sure that the set of tests spans as much of the source code as possible. This is because the logical structure of the Automaton will not change much in subsequent iterations, so it is paramount that any erroneous behaviours are found and corrected now.

ID	Description	Input	Expected Output	Actual Output	Result
1a	Verifying that the automaton responds to presence/lack of key in transTable	Key that exists in transTable	Simulator continues into next logical block	Simulator continues into next logical block	Pass
1b		Key does not exist in transTable	Invalid Input	Invalid Input	Pass
2a	Confirming that the simulator responds to fresh/stale input correctly	Input is not contained in any register	Automaton counts number of transitions that match the source state & "true"	Automaton counts number of transitions that match the source state & "true"	Pass
2b		Input is contained in some register	Automaton counts number of transitions that match the source state & "false"	Automaton counts number of transitions that match the source state & "false"	Pass
3a	Confirming that the simulator responds correctly to the number of transitions available	Number of transitions = 0	"No transitions found"	"No transitions found"	Pass
3b		Number of transitions = 1	Transition executed deterministically	Transition executed deterministically	Pass
3c		Number of transitions > 1	User asked to choose transition to execute	User asked to choose transition to execute	Pass
4a	Verifying that the user's choice corresponds to the correct transition	Choice = 0	Transition corresponding to '0' is executed	Transition corresponding '0' is executed	Pass
4b		Choice = 1	Transition corresponding to '1' is executed	Transition corresponding to '1' is executed	Pass
4c		Choice = 2	Transition corresponding to '2' is executed	Transition corresponding to '2' is executed	Pass
5a	Verifying that the choice made by the user is always within the bounds set by the simulator: Possible choices include 1, 2, 3	User inputs 0	User is requested to input choice between 1 & 3	User is requested to input choice between 1 & 3	Pass
5b		User inputs 4	User is requested to input choice between 1 & 3	User is requested to input choice between 1 & 3	Pass

6a	Confirming that the automaton accepts strings within the language: Multiple instances of the same character	"a01201"	Valid input (0)	Valid input (0)	Pass
6b		"a01201"	Valid input (1)	Valid input (1)	Pass
6c		"aa012"	Valid input (a)	Valid input (a)	Pass
6d		"0aa0"	Valid input (0)	Valid input (0)	Pass
6e		"0aa0"	Valid input (a)	Valid input (a)	Pass
7a	Confirming that the automaton rejects strings that are not within the language: Multiple instances of the same character	"01201"	Invalid input (2)	Invalid input (2)	Pass
7b		"x01b20d1a"	Invalid input (2,x, b, a)	Invalid input (2)	Pass

Table 18 - GFRA System Tests

XML Parser – Version 2

The second version of xmlParser simply extends the first by adding an implementation of **Algorithm 2**. The output from this version of xmlParser will be used as the input to the final version of the automaton; creating a system that can parse an XML file and verify that it conforms to a specified property. Tests for this algorithm will have complete code coverage as it is of vital importance that the input to the automaton is correct. The full set of tests will not be included, however, a subset spanning the different types of tests will be shown. Component tests were skipped for the xmlParser (version 2) as they will be included in USNFRA section.

Unit Tests

ID	Description	Input	Expected Output	Actual Output	Result
1a	Verifying input to new algorithm is correct	Books.xml	Sequence of data words spanning all attributes	Sequence of data words spanning all attributes	Pass
1b		Cd.xml	Sequence of data words spanning all attributes	Sequence of data words spanning all attributes	Pass
2a	Confirming that indexes are added to list properly: First data letter of each data word	Cd.xml	9, 52, 96, 139	9, 52, 96, 139	Pass
2b		Books.xml	11, 72, 105, 153, 188	11, 72, 105, 153, 188	Pass
2c	Last data letter of each data word	Books.xml	48, 81, 130, 163, 202	48, 81, 130, 163, 202	Pass
2d		Cd.xml	33, 77, 120, 164	33, 77, 120, 164	Pass
2e	All data letters in between first and last	Cd.xml	21, 64, 107, 152	21, 64, 107, 152	Pass
2f		Books.xml	23, 33, 115	23, 33, 115	Pass
3a	Verifying output of algorithm is correct: Sequence of data letters spanning all attributes in XML file	Books.xml	(cat,action) (id,bk101) (type,hardback) (lang,en) (id,bk101) (type,hardback) (id,bk103) (status,alive)	(cat,action) (id,bk101) (type,hardback) (lang,en) (id,bk101) (type,hardback) (id,bk103) (status,alive)	Pass
3b		Cd.xml	(Genre,Rock) (Rating,***) (Genre,Rock) (Rating,****) (Genre,Pop) (Rating,****) (Genre,Blues) (Rating,***)	(Genre,Rock) (Rating,***) (Genre,Rock) (Rating,****) (Genre,Pop) (Rating,****) (Genre,Blues) (Rating,***)	Pass
4a	Confirming that erroneous input is dealt with appropriately	XML file spelt incorrectly	File not found error	FileNotFoundExcepti on	Pass
		XML file not properly formed	Parse exception	SAXParseexception	Pass

Table 19 - Unit Tests (USNFRA/xmlParser)

The Property Verifier (USNFRA)

We know that the xmlParser, Quad, Pair and Register classes all function correctly as judged by unit/component tests in the previous phases, and hence unit tests for these classes will be omitted. Unit and component tests will be carried out on the Automaton and TransTable classes (which test the other classes), and the chapter concludes with system tests of the USNFRA. A successful set of system tests will serve to prove the accurate simulation of a Fresh-Register Automata over XML files.

Unit/Component Tests: TransTable

ID	Description	Input	Expected Output	Actual Output	Result
1a	Confirming that the specification file is read properly	Specification.txt	No Error	No Error	Pass
b	Confirming that transTable handles a missing specification file	N/A	File not found error	File not found error	Pass
c	Confirming that transTable handles a misspelled specification file	Spcification.txt	File not found error	File not found error	Pass
2	Confirming that the specification file is read properly	Specification file is 8 lines long	LineNo (variable to store current line number) ranges from 0-7	LineNo ranges from 0-7	Pass
3a	Confirming that the number of commas specified on the first line is counted correctly	# commas = 1 (minimum)	1	1	Pass
3b		# commas = 0	Error	"Initial state & final state(s) must be specified	Pass
3c		# commas = K	K	K	Pass
4a	Confirming that the index of each comma is retrieved correctly	# commas = 1	Index = 1	Index = 1	Pass
4b		# commas = 2	Index = 4, 7	Index = 4	Pass
4c		# commas = k	index = 1, 4, 7, 10 ... 3k-2	index = 1, 4, 7, 10 ... 3k-2	Pass
5	Confirming that the initial & final states specified are retrieved properly	1, 2, 3, 4	1, 2, 3, 4	1, 2, 3, 4	Pass
6a	Confirming that the value used to specify the number of registers to create is valid	1 (minimum)	1	1	Pass
6b		K such that k < 1	Error	"Value must be a positive, non-zero integer"	Pass
6c		K such that k is not an integer	Error	NumberFormatException	Pass
d		K such that k > 1	5	5	Pass

7	Confirming that indexes of commas are found correctly	One, two, “true”, 0	3, 8, 16	3, 8, 16	Pass
8a	Confirming that quads are read and generated correctly	1, 1, “true”, 0	No Error getLeft() = 1 getMidLeft() = 1 getMidRight() = “true” getRight() = 0	No Error getLeft() = 1 getMidLeft() = 1 getMidRight() = “true” getRight() = 0	Pass
8b		1, 1, “false”, 0	No Error getLeft() = 1 getMidLeft() = 1 getMidRight() = “false” getRight() = 0	No Error getLeft() = 1 getMidLeft() = 1 getMidRight() = “false” getRight() = 0	Pass
8c		Quad not specified correctly (minimum length not surpassed)	Error	"Please specify a quad with the following format: source, sink, true/false, register”	Pass
9a	Ensuring value of midRight is either “true” or “false”	midRight = “true”	“true”	“true”	Pass
9b		midRight = “false”	“false”	“false”	Pass
9c		Other	Error	“midRight must be either true or false”	Pass
	Ensuring register specified in transition exists within 0 <= # < k where k is number of registers	Number specified is within 0<=#<k	No Error	No Error	Pass
		Number specified is not within 0<=#<k	Error	"Register in transition (line number) is invalid	Pass
10	Confirming that transition table is generated correctly	Specification.txt	Initial state = 1, final State = 3 # Registers = 2 1, 1, true, 0 1, 2, true, 0 1, 3, false, 1 2, 2, true, 1 2, 3, false, 1 3, 3, true, 1 3, 3, false, 1	Initial state = 1, final State = 3 # Registers = 2 1, 1, true, 0 1, 2, true, 0 1, 3, false, 1 2, 2, true, 1 2, 3, false, 1 3, 3, true, 1 3, 3, false, 1	Pass

Table 20 - Unit/Component Tests (USNFRA/TransTable)

Automaton

For test ID 5, the number of transitions being smaller than 1 is impossible and will not be tested. This is because if a transition relating to a specific input does not exist in the transition table, then the key for that transition will not exist. Therefore, the case will be caught in test 4b.

ID	Description	Input	Expected Output	Actual Output	Result
1a	Verifying that all components to the automaton instantiate properly	TransTable transTable = new TransTable(); transTable.initTransTable("specification.txt");	No Error	No Error	Pass
1b		Register register = new Register(transTable.getRegisters());	No Error	No Error	Pass
1c		String curState = transTable.getCurState();	No Error	No Error	Pass
1d		xmlParser parser = new xmlParser(); xmlIn = parser.parse("books.xml");	No Error	No Error	Pass
2	Verifying that the main control loop has the correct limits	xmlParser generates 8 data letters	Loop ranges from 0 – 7	Loop ranges from 0 -7	Pass
3a	Verifying that freshness is determined correctly	Register.isFresh(aChar) where aChar is not contained in any register	aChar is fresh	aChar is fresh	pass
3b		Register.isFresh(aChar) where aChar is contained in some register	aChar is stale	aChar is stale	Pass
4a	Verifying that containsKey(key) works as intended	transTable.containsKey(k) where k is contained in the transition table	True	True	Pass
4b		transTable.containsKey(k) where k is not contained in the transition table	False	False	Pass
5a	Verifying that the number of fresh/stale transitions is counted correctly	F fresh transitions where F is an integer greater than 0	F	F	Pass

5b		S fresh transitions where S is an integer greater than 0	S	S	Pass
6a	Verifying that single transitions (no choice involved) are executed automatically	# fresh transitions = 1	Transition executed automatically	Transition executed automatically	Pass
6b		# stale transitions = 1	Transition executed automatically	Transition executed automatically	Pass
7a	Verifying that when multiple transitions are deemed relevant, the user is asked to choose one to execute	# fresh transitions = k Example of k = 5	User has a choice between k different transitions, enumerated from 0 to k-1	5 transitions returned, numbered from 0 to 4	Pass
b		# stale transitions = k Example of k = 3	User has a choice between k different transitions, enumerated from 0 to k-1	5 transitions returned, numbered from 0 to 2	Pass
8a	Confirming that user input (for choosing a transition) is within the range specified by the automaton	# transitions = 4, user enters a number between 0 & 3	No error	No Error	Pass
8b		# transitions = 4, user does not enter a number between 0 & 3	Error	"Please enter integer corresponding to the transitions available"	Pass
9	Ensuring that registers are written to correctly	Input value = (id, bk101) To be written to register 0	(id, bk101) written to register 0	(id, bk101) written to register 0	Pass
10	Ensuring that current state is updated correctly	1, 2, true, 0	New state = 2	New state = 2	Pass

Table 21 - Unit/Component Tests (GNFRA/Automaton)

System Tests

In this last set of tests, we aim to verify that the system accepts and rejects valid and invalid inputs respectively. Various XML files will be used as well as several specification files, however, only a subset of the repeated tests will be listed to show the breadth of testing.

ID	Description	Input	Expected Output	Actual Output	Result
1a	Verifying a Property: XML file contains two identical attributes	Books.xml Specification.txt	Valid input (id,bk101)	Valid input (id,bk101)	Pass
1b			Invalid Input (cat,action)	Invalid Input (cat,action)	Pass
1c			Valid Input (type,hardback)	Valid Input (type,hardback)	Pass
1d			Invalid Input (lang,en)	Invalid Input (lang,en)	Pass
1e			Invalid Input (id,bk103)	Invalid Input (id,bk103)	Pass
1f			Invalid Input (status,alive)	Invalid Input (status,alive)	Pass
1g		cd.xml Specification.txt	Valid Input (Genre,Rock)	Valid Input (Genre,Rock)	Pass
1h			Valid Input (Rating,***)	Valid Input (Rating,***)	Pass
1i			Valid Input (Rating,****)	Valid Input (Rating,****)	Pass
1j			Invalid Input (Genre,Blues)	Invalid Input (Genre,Blues)	Pass
2a	Verifying a property: XML File contains two identical attributes, in positions i, i+1 in sequence of data letters	Cd.xml Specification2.txt	Valid input (rating,****)	Valid input (rating,****)	Pass
2b		(rating,****) removed	Invalid input	Invalid input	Pass
2c		Books.xml Specification2.txt	Invalid Input	Invalid Input	Pass
2d		(lang,en) & (id=bk101) removed	Valid Input	Valid Input	Pass
2e		(type,hardback) & (lang,en) removed	Valid Input	Valid Input	Pass
3a	Verifying that the automaton correctly handles missing input files	No input files specified	Error	FileNotFound Error	Pass
3b		Specification file not given	Error	"Specification file not found"	Pass
3c		XML File not given	Error	SAXParseException	Pass

Table 22 - USNFRA System Tests

The final set of tests show that the automaton works as intended (respective to the specification file) over complete code coverage. Tests that were not included in Table 22 involved changing classes and specific inputs to try and cause erroneous behaviours.

Chapter 5: Project Management

The project management approach played a significant role in ensuring that the project progressed efficiently and deliverables were documented as required. This chapter details the management techniques that were used to allow the project to progress successfully whilst completing all the objectives listed in the project specification document. To conclude this section, the project timeline is presented with a description of how each objective was met.

Software Design & Development Approach

This project was split relatively evenly between research and implementation, meaning that an agile approach to design and development was preferred. After a few meetings with the project supervisor, it was clear that the best approach to implementing the solution was to start with a simple design and keep adding to its functionality until a final solution had been made. This meant that each phase of development could be split into four sections: research, design, implementation and testing. Developing the project with an agile approach meant that the design or implementation could be changed at any point instead of only during the design state. Should problems have arisen with the design during development, the problematic section could be redesigned without having to start the entire section again.

Research played an important part in this, as it was paramount to know what the goal of each phase was before design or implementation were even considered. A top-down approach was used when designing each iteration of the system; this aided in ensuring that the overarching aims of each phase were clear (greatly contributing to the success of the project), whilst simultaneously avoiding over encumbrance with details of the implementation. On the other hand, a bottom-up approach was taken when developing the system because it meant that high level elements did not need to be changed even if low level elements were altered frequently.

The agile methodology chosen utilised some aspects of Extreme Programming; specifically, it meant that implementation and testing could coincide with one another. This meant that solutions could be designed and implemented upon discovery of a problem, usually without affecting high-level elements in the system; this meant that

the project could continue progressing efficiently because only the problematic unit would need to be changed.

Many aspects of Extreme Programming could not be used whilst the project was in development. For instance, Extreme Programming necessitates regular contribution from the customer in the design and development phase so that they can shape the product being developed. This was impractical, as there was no identifiable customer as the project was research oriented. Nevertheless, regular input from the project supervisor was substituted for customer comments and recommendations. One feature of Extreme Programming that could not be facilitated was the pair programming arrangement, where developers test and give feedback on each other's code. Finally, acceptance testing plays another large part in Extreme Programming. However, with no definitive user application being created, or customer to tailor to, acceptance testing was not carried out in the usual manner. Instead, the testing that occurred at the end of each development phase could be viewed as acceptance testing from the point of view of the researcher; they knew what needed to be implemented and it was up to them to decide if the system met the requirements needed to proceed to the next development cycle.

Weekly meetings with the project supervisor were held to exploit the benefits of agile design and development. These formal meetings with the project supervisor were an excellent way to deliberate project updates and discuss the future direction of the project, as well as receiving feedback on the current state of the project. E-mails were often sent to clarify any points that remained unclear at the end of the meetings, as well as to arrange the date and time of the next meeting. Frequent communication enabled the project to progress swiftly and without disruption.

Project Timeline

The project timeline initially specified in the project specification document, and later mentioned in the progress report, held to the end of the project with only a few alterations. The two Figures below show a reiteration of the initial project timeline as well as the edited version.

	13/10/16	15/11/16	15/12/16	15/01/17	15/02/17	15/03/17	15/04/17	30/04/17
Obj. 1								
Obj. 2								
Obj. 3								
Obj. 4								
Obj. 5								
Obj. 6								
Obj. 7								
Obj. 8								
Obj. 9								
Obj. 10								
Obj. 11								
Obj. 12								
Project Spec.								
Progress Report								
Oral Pres.								
Project Report								

Figure 14 - Initial Project Timeline (Blue = Time Allocated, Red = Deadline)

	13/10/16	15/11/16	15/12/16	15/01/17	15/02/17	15/03/17	15/04/17	30/04/17
Obj. 1								
Obj. 2								
Obj. 3								
Obj. 4								
Obj. 5								
Obj. 6								
Obj. 7								
Obj. 8								
Obj. 9								
Obj. 10								
Obj. 11								
Obj. 12								
Obj. 13								
Obj. 14								
Project Spec.								
Progress Report								
Oral Pres.								
Project Report								

Figure 15 - Revised Project Timeline (Blue = Time Allocated, Red = Deadline)

There are some differences between Figures 14 & 15, the first was that Obj. 12 (a stretch objective, listed below) was infeasible as the design and implementation of an automaton cannot be automated. Nevertheless, a somewhat similar task was achieved in its place: Instead of designing the automata from a specified input, the input file contained a description of the automaton that the user had designed. The system then read the file and simulated the automaton. Some other differences involved small changes to time allocation, objectives 6 & 7 did not need as much time as initially thought, and the oral presentation work had to be rescheduled to make time for other coursework. Finally, objectives 8&9 are replaced by objectives 8-12 as defined in the next section.

Objectives

Objectives 8 & 9 from the initial project timeline have been rewritten into objectives 8 through to 12, this was to better represent the project's progression.

1. Identify how to parse XML.
    a. Read relevant literature.
    b. Create prototypes.
    c. Refine prototypes.
    d. Repeat b and c until decided on a final implementation.
2. Find out how to generate 'keys' from XML tags.
    a. Same process as in 1.
3. Identify how to generate 'values' from XML attributes.
    a. Same process as in 1.
4. Find out how to generate letters that are (key, value) pairs from objectives 2 and 3.
    a. Same process as in 1.
5. Identify how to generate words that are strings of objective 4.
    a. Same process as in 1.
6. Design and implement a tool to automate objective 5, using an XML script as input.
    a. Same process as in 1.

7. Test implementation of objective 6 for errors/bugs.
    a. Create testing rigs.
    b. Identify erroneous results.
    c. Analyse erroneous results.
    d. Update implementation of objective 7.
    e. Repeat until it is not trivial to think of new test cases.
8. Create a Deterministic Finite Automaton
    a. Research.
    b. Design.
    c. Implement.
    d. Test.
9. Create a Nondeterministic Finite Automaton
    a. Same process as in objective 8.
10. Create a Register Automaton
    a. Same process as in objective 8.
11. Create a Generic Register Automaton
    a. Same process as in objective 8.
12. Simulate an automaton that is specified within an external file
    a. Same process as in objective 8
13. Check that the simulation from objective 12 accepts words obtained from the parser tool if the property holds.
    a. Same process as in objective 7.
14. Check that the simulation from objective 12 rejects words obtained from the parser tool if the property doesn't hold.
    a. Same process as in objective 7.

Objective Completion

The following Table shows which files corresponded to the completion of which objectives.

Objective(s)	File(s)
1 – 7	CS310/xmlParser USNFRA/xmlParser.java
8	DFA/Automaton.java
9	NFA/Automaton.java
10	NFRA/Automaton.java
11	GNFRA/Automaton.java
12 - 14	USNFRA/Automaton.java Books.xml Specification.txt

Table 23 - Objectives & Their Completion

In each case that Automaton.java is specified, the class itself makes use of other classes; these other classes were not included in the table for the sake of brevity, but are found within the folder containing the class specified.

Chapter 6: Evaluation

The project has two main features to be evaluated: the approach to design and development, and the way in which the project was managed. Any issues in project development that were caused by management techniques will be outlined, as well as any improvements or changes that could have been made to the methods used for designing and developing the project.

Evaluation of Design & Development

This project utilised an agile approach to design and development, with a top-down approach to design and a bottom-up method of development. The implementable section was split into phases, with each phase having a different goal to the one preceding it. If the phases are viewed as a cumulative whole, then the completion of each phase contributed to the overarching objective of the project. This method of managing the project's development was extremely successful as the overall objective for each phase of development could be considered and decided upon, and then programs to achieve these goals were envisioned.

An agile approach turned out to be very useful because development of a phase could start before all aspects of the design were finished. For instance, the State and Pair classes were programmed before the TransTable class had been fully designed. This was incredibly useful as those units could then be tested to make sure that there were no problems with them, which helped mitigate errors that could occur further on in development. The agile approach also worked well with the research focus of the project, meaning that if a problem with the design was identified, new insight could be gained then and there, culminating in the redesign of a unit instead of the whole system. For instance, when it became apparent that automata with memory systems would be required, research could begin immediately without having to change the project timeline. Overall, the use of an agile approach with aspects of Extreme Programming lent itself extremely well to the project's successful and punctual completion.

Evaluation of Project Management

The project was managed with multiple activities, some of which were recurring. At the beginning of the project, designing a timeline was of utmost importance as it would aid to organise the project into manageable chunks and keep track of progress. The timeline was reassessed at roughly mid-way through the project because it became clear that objectives 8 and 9 (from the original project timeline) needed to be split into smaller sub-objectives and objective 12 was infeasible. This made the project more manageable and showed the project's progression in a much clearer way, whilst not actually affecting the project's timeline at all as the new objectives were to be completed in the same timeframe. While this was not a setback as such, it could have been avoided if the objectives had been better designed, this was fundamentally caused by a lack of prior research when the project was still in its infancy. However, it was eventually resolved when the appropriate research had been completed.

The project's progress was discussed in weekly meetings with the project supervisor. These regular meetings were quintessential to the project's development as the aims and objectives of the project were discussed and clarified. Once the implementable section had been broken down into phases, the meetings were a way to receive feedback on the development of a phase, as well as discussing the general direction that the next phase should take, with smaller details being left to be researched individually or to be discussed further in e-mail conversations. Without these regular meetings and e-mail discussions project progression would have been heavily hindered.

Overall, the use of an agile approach and the regular meetings were the main contributors to the project's completion. Had a Waterfall approach been taken; the project may not have been completed on time. This is because the entire system may have had to be redesigned multiple times to deal with changes that were made to individual units during development that were unavoidable at the time. However, this is assuming that the details of the design of the system were not added to in great detail, as this would have avoided having to redesign units in the first place. A Waterfall method would therefore have worked, as long as sufficient detail was added to the original design of each phase.

Chapter 7: Conclusion

Within this final chapter, the project is summarised and its key achievements and contributions are identified. Future work is outlined, where improvements are expressed and extensions applicable to the project are elaborated upon.

For the project to be a success, a routine that simulated an automaton verifying XML attribute properties was to be created. This entailed using equivalence operations on XML attribute values to verify whether the XML document conformed to a specific property; where an example property could be that two attributes found in the document have the same value. The routine required that a user inputted an XML file and a specification file (.txt or equivalent) that described the automaton to be simulated in terms of states, registers and transitions.

The routine initially began by simulating a DFA, going through many phases consisting of research, design, development and testing in which the automaton was tweaked and refined until the final version of the automaton was settled upon. A fresh-register automaton was determined to be an effective model, as it meant that equivalence relations could be imposed on the indexes of its input by using memory systems to store input values.

The challenge of this project was found in the design and development of the automaton, in that there were significant differences in theory than in implementation. For instance, a transition table in theory is just a combination of symbols and operators, but those symbols and operators had to make sense in Java. This meant that the implementation of the automaton was far more in depth than at first imagined, as data structures and classes had to be created to give meaning to these theoretical concepts. Not only this, but there were many ways that this simulation routine could have been implemented, so deciding on an implementation strategy was of utmost importance.

The contribution of this project is a routine that accurately simulates a fresh-register automaton processing data words created from the element tags and attribute values found in an XML document with the overarching objective of verifying whether the XML document conforms to an arbitrary property.

Future Work & Extensions

This project could be expanded significantly in three ways. To begin with, the implementation could be refined and optimized. For instance, instead of counting the number of commas found in the first line of the specification file and statically sizing an array based on these calculations, an ArrayList could have been used instead.

The project also has the capacity to be extended from an academic point of view, where the model of automata used could be replaced by a more complex version – like one register alternating automata [21] – this is significant because it remains to be seen whether using a more complicated model of automata results in a more complex set of properties that could be verified. Furthermore, a lot of insight could be gained from seeing how these models function when implemented as an algorithm, i.e. how changing the implementation affects its performance.

Lastly, the project could be expanded in terms of usability. Where the simulator currently lives in the Eclipse IDE, it could be transformed into a fully-fledged application with its own Graphical User Interface (GUI). This would be extremely beneficial for many reasons, namely, it means that the inputs to the simulator can be defined outside of the source code, and that the application could run on any machine with the Java Virtual Machine (JVM). These small improvements entail a significantly better user experience overall, as the application could be used practically anywhere, and would have a much better 'look-and-feel'.

References:

[1] Chalmers University of Technology, Gerardo Schneider, Model Checking <http://www.cse.chalmers.se/~gersch/model-checking/What-is-Model-Checking.html> Accessed 30/03/2017

[2] W3Schools, Introduction to XML <https://www.w3schools.com/xml/xml_whatis.asp> Accessed 29/03/2017

[3] W3Schools, Introduction to HTML <https://www.w3schools.com/html/html_intro.asp> Accessed 29/03/2017

[4] University of Warwick, CS259 DFA Slides, A. Murawski, p14-19 <http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs259/reg1.pdf> Accessed 30/03/2017

[5] TutorialsPoint, Introduction to Automata Theory <https://www.tutorialspoint.com/automata_theory/automata_theory_introduction.htm> Accessed 01/04/2017

[6] University of Warwick, CS259 NFA Slides, A. Murawski, p18-28, <http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs259/reg2.pdf> Accessed 01/04/2017

[7] W3Schools, Introduction to XML Schemas <https://www.w3schools.com/xml/schema_intro.asp> Accessed 03/04/2017

[8] W3, Introduction to DOM, Johnathan Robie <https://www.w3.org/TR/WD-DOM/introduction.html> Accessed 03/03/2017

[9] TutorialsPoint, Introduction to the Tree data structure <https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm> Accessed 04/04/2017

[10] W3Schools, Introduction to XPATH <https://www.w3schools.com/xml/xml_xpath.asp> Accessed 04/04/2017

[11] TutorialsPoint, XML parsing with XPATH <https://www.tutorialspoint.com/java_xml/java_xpath_parse_document.htm> Accessed 05/04/2017

[12] Oracle Help Centre, DocumentBuilder class

<https://docs.oracle.com/javase/7/docs/api/javax/xml/parsers/DocumentBuilder.html> Accessed 05/04/2017

[13] Oracle Help Centre, Document interface <https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Document.html> Accessed 05/04/2017

[14] Oracle Help Centre, Node Interface <https://docs.oracle.com/javase/7/docs/api/org/w3c/dom/Node.html> Accessed 05/04/2017>

[15] S. Demri & R. Lazić, LTL with the Freeze Quantifier and Register Automata, 2008, p1-11, Accessed 05/04/2017

[16] L. Segoufin, Automata and Logics for Words and Trees over an Infinite Alphabet, 2006, p41-57, Accessed 05/04/2017

[17] F. Neven, T. Schwentick, V. Vianu, Finite State Machines for Strings Over Infinite Alphabets, Jan 2003, p404-409, Accessed 05/04/2017

[18] M. Kaminski, N. Francez, Finite-memory automata*, Oct 1993, p329-339, Accessed 05/04/2017

[19] N. Tzevelekos, Fresh-Register Automata, Nov 2010, p1-7, Accessed 05/04/2017

[20] A. S. Murawski, S. J. Ramsay, N. Tzevelekos, Bisimilarity in Fresh-Register Automata, 2015, p1-12, Accessed 06/04/2017

[21] D. Figueira, Alternating register automata on finite data words and trees∗, Mar 2012, p1-43, Accessed 10/0-4/2017

[22] K. Beck, Extreme Programming Explained: embrace change <http://dl.acm.org/citation.cfm?id=318762> Accessed 10/04/2017

[23] Oracle Help Centre, hashCode function <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()> Accessed 12/03/2017

[24] University of Warwick, CS126 Hash Table Slides, M. Joy, p2-31 <http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs126/09_hash_tables.pdf> Accessed 13/03/2017

[25] Oracle Help Centre, Scanner class

<https://docs.oracle.com/javase/7/docs/api/java/util/Scanner.html> Accessed 13/03/2017

[26] University of Warwick, CS126 Array & Linked List Slides, M. Joy, p2-7 <http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs126/05_arrays_and_linked_lists.pdf> Accessed 14/04/2017

[27] Oracle Help Centre, Object class, toString method <https://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#toString> Accessed 14/04/2017

[28] Oracle Help Centre, ArrayList class <https://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html> Accessed 15/04/2017

[29] Oracle Help Centre, BufferedReader <https://docs.oracle.com/javase/7/docs/api/java/io/BufferedReader.html> Accessed 15/04/2017

[30] Oracle Help Centre, Java Collections Framework <http://docs.oracle.com/javase/7/docs/technotes/guides/collections/overview.html > Accessed 16/04/2017

[31] University of Warwick, CS259 Subset Construction Slides, A. Murawski, p18-27 <http://www2.warwick.ac.uk/fac/sci/dcs/teaching/material/cs259/reg4.pdf> Accessed 18/04/2017