A framework for benchmarking FA-based String Recognizers*

Ernest Ketcha Ngassam
SAP Meraka UTD
and
University of South Africa
Pretoria, 0001
ernest.ngassam@sap.com

Derrick G. Kourie
Department of Computer Science, University of Pretoria
Pretoria, 0002
dkourie@cs.up.ac.za

Bruce W. Watson
Department of Computer Science, University of Pretoria
Pretoria, 0002
bwatson@cs.up.ac.za

ABSTRACT

Previous work on implementations of FA-based string recognizers suggested a range of implementation strategies (and therefore, algorithms) aiming at improving their performance for fast string recognition. However, an efficient exploitation of suggested algorithms by domain-specific FA-implementers requires prior knowledge of the behaviour (performance-wise) of each algorithm in order to make an informed choice. We propose a unified framework for frequently evaluating existing FA-based string recognizers such that FA-implementers could capture appropriate problem domains that guarantee an optimal performance of available recognizers. The suggested framework takes into consideration factors such as the kind of automaton being processed, the string and alphabet size as well as the overall behaviour of the automaton at run-time. It also forms the basis for further work on FA-based string recognition applications in specific computational domains such as natural language processing, computational biology, natural and computer virus scanning, network intrusion detection, etc. It is well-known that performance remains a significant bottleneck to the high-performance solutions required in such industrial applications.

Keywords

Automata, String Recognizer, Automata Implementation, Performance of String Recognizers

1. INTRODUCTION

The resolution of computational problems involving pattern matching using Finite Automata (FAs) is not new. Many authors in the field have reported on a variety of solutions, both at conceptual (symbolic) as well as practical (processing) levels. Research reported in sources such as [2, 16, 17] report on a variety of transformations on automata that produce more compact ones. Although such techniques are useful for implementing real life solutions in various domains, the research is mainly reported at a conceptual level. Because much of the implemented work is company-confidential (e.g: Xerox, AT&T, Cisco), little implementation detail is available in the research literature. Furthermore, FA-based string processing appears to be a straightforward exercise in the sense that the theoretical complexity of the required algorithm is linear to the length of the string to be tested. However, investigations have revealed that the computational medium used for processing the recognizer as well as the characteristics of the automaton (the sparsity of its transition table and caching behaviour at run-time) can be fundamental performance bottlenecks [11, 5].

In [10], various ways were considered in which cache usage could be optimised at run-time. This work featured the following:

- Up to 168 algorithms were suggested. However, many of these were based on theoretical considerations that aimed to maximise the advantages of cache memory utilisation. The impact of other aspects of the computing infrastructure that could give rise to additional algorithms (such as the operating system and the machine architecture) were not considered.
- Only a limited number of algorithms were actually implemented and benchmarked. While this was based on artificially generated data the algorithms frequently outperformed their conventional counterparts. Nevertheless, there is a need more precisely to characterise the various domains where they can be optimally exploited.

The foregoing points to the need for a framework to efficiently test, cross-compare, classify and rank FA-based recognizers, taking into account parameters such as hardware capabilities, the problem domain, and input characteristics.

The remaining part of this paper is structured as follows. Section 2 below briefly introduces conventional string recognizers and cache optimized strategies suggested for performance enhancement. In Section 3, a generic library is proposed that encapsulates all implementation strategies suggested in previous work. It is shown in this section that the efficient exploitation of the various algorithms is subject to the availability of a consolidated platform to study their usage. The building blocks of such a platform is suggested in Section 4. The conclusion and further directions to this work are presented in Section 5.

2. FA-BASED STRING RECOGNIZERS

An FA-based string recognizer/processor is an algorithm that that takes as input a string and checks whether it is part of the language modeled by the automaton or not. Throughout this paper, we assume the automaton (FA) to be deterministic, meaning that, given a transition function δ, a state q_i of the automaton, and any symbol s_j of the alphabet, the relation $\delta(q_i, s_j)$ exists, that is, each symbol s_j of the alphabet always triggers a transition to a destination state.

Traditionally, the table-driven (refer to [4] for details) approach is used for implementating a recognizer. The algorithm maintains a two dimensional array in memory (the transition function) and each symbol is scanned and evaluated to ascertain whether it triggers a transition to a next state. The process terminates either when there are no more symbols to scan, or when the FA has reached a rejecting state. The recognition process is therefore very simple and the recognizer's role is to return a boolean that establishes whether the string is part of the language modeled by the FA or not. It is easy to establish that theoretical complexity of the recognizer is linear to the length of the string. However in practice, the memory load could constitute a performance bottleneck if large numbers of cache misses occur. The sparsity of the transition table may also affect the recognizer's performance.

As early as in the late 60's Kent Thompson in [14] suggested hardcoding the FA as an alternative to the table-driven approach. Here, the transition table is hardcoded into instructions that make up the recognizer as a whole. Also, the string to be tested for acceptance is embedded in the algorithm using directly executable instructions. The performance of a hardcoded (HC) recognizer is hampered by the number of instructions required to implement an automaton made of many instructions. Thus, as the automaton size grows, the number of cache misses may also grow significantly, especially if contiguous instructions are accessed relatively infrequently. Investigations in [4] revealed that the hardcoded algorithm outperforms the table-driven (TD) version for automata in the range of hundreds of states. This observation further suggested that there is a need to investigate appropriate algorithms for processing large automata.

A hybrid version for implementing string recognizers referred to as mixed-mode (MM) was suggested in [9]. In this case, the transition set is split into two disjoints subsets such that a portion of the transition table is hardcoded into the recognizer, while the remaining portion is implemented as a table to be loaded into memory. During the recognition process a hardcoded state is directly executed based on appropriate instructions, and memory access is performed when a table-driven state is encountered. In terms of performance, the MM algorithm may still suffer from memory and instruction load and there is a need to further investigate and establish the appropriate method for splitting the transition set. Furthermore, there is a lack of tangible domain specific information establishing the appropriate computation domain where such a hybrid algorithm performs at an optimum.

In [10], a range of cache optimized strategies was suggested. The subsections below briefly review the respective strategies. These strategies serve as the foundation for the suggested class-library discussed in section 3.

2.1 The Dynamic State Allocation Strategy

Implementation of FA-based string processors that rely on the dynamic state allocation principle requires that a dynamically allocated space be created in memory which is used during acceptance testing. At runtime, as each state is encountered that falls for the first time within the *string path*[1], it is allocated a memory block into which the state's transition information (i.e. a row in the original transition table) is copied. Subsequent references to such a state's transitions are then made via this new piece of memory, rather than via the original transition table. Furthermore, the memory blocks allocated to states on the string path are contiguous, and arranged in the order in which the states are encountered [5, 6].

The DSA strategy is a form of Just-In-Time (JIT) processing, applied in the context of FA-based recognizers. The states being accessed are dynamically allocated in memory according to the string being processed. If the string path involves repeated visits to a limited number of states, and if the order in which states are visited remains more or less the same, then it is expected that such an approach will have certain advantages. Specifically, it is hoped that because states to be visited are regrouped in a compact fashion and organized contiguously, the number of cache misses in memory will be relatively low. In practice, the DSA strategy can be employed when implementing core recognizers (i.e. TD, HC and MM). The portion of the memory reserved for the allocation of newly encountered states can be infinite (***unbounded***) or predefined (***bounded***) based on some threshold. The definition of such threshold should therefore be based on capabilities of the hardware used for processing as well as the behaviour of the automaton at run-time. Another implementation strategy is described next.

2.2 The Allocated Virtual Caching Strategy

[1]String path is construed to mean the set of visited states that are encountered during the processing of the input string.

The implementation of FA-based string processing algorithms using the allocated virtual caching strategy (henceforth called the AVC strategy) involves the dedication of a portion of the memory that contains state information to holding state transition information that is needed for acceptance testing. Such a dedicated portion of the memory is referred to as the *allocated virtual cache*. During acceptance testing, states are reordered in the cache as they are visited in order to enhance the spatial and temporal locality of reference of the cache's contents in subsequent phases of testing the input string. Due to its limited size, the virtual cache is unlikely to always contain every single state required. As a result, when reference is made to a state that is not present in the cache, a replacement policy is followed to remove a state from the cache [10]. The transition information of the state swapped out of the cache has to be copied to the memory block previously occupied by transition information relating to the state to be placed into the cache. There are various state replacement policies that could be followed, for example: *direct mapping* ; a *LRU* policy; or an *associative mapping*[12] policy. On the other hand, because of the overheads involved, it might be better not to carry out any replacement at all. In this latter case, once the cache is full, acceptance testing continues in the table without any replacement. Such an approach will reduce overheads while hoping that the states in the cache remain organized in a fashion that has a high cache hit rate. The term *virtual* cache is used to reference the dedicated memory block in order to differentiate it from the well-known *hardware* cache memory.

The AVC strategy thus aims to exploit the benefits of cache memory, in the hope of deriving algorithms that are more efficient under certain conditions than the traditional algorithms. The algorithms derived from using the AVC strategy are to be considered when recognizers are based on large automata and the string path tends to visit states that are frequently present in the virtual cache. If, in addition, the string path visits states *contiguously stored* in the virtual cache, then that would lead to even better performance.

In practice, although states initially present in the cache may be part of the string path, there is no guarantee that this is always the case. Therefore, the AVC strategy requires that all the states that fall on the string path are moved into the cache even if the cache is full. This move operation is performed as the string is being processed.

The cache may be viewed as a stack. Initially, it is empty, and its top (which will also be referred to as the cache line) is a pointer to the memory occupied by state 0 transition information. If, at any stage while the cache is not full, the next state to be accessed is not at or below the position where the cache line currently points, then the required state information is located in memory and swopped out with the data in the cache line. Thereafter, the cache line pointer is increased. Eventually, the cache line pointer reaches a position which indicates that the cache is full. When the cache is full and the state being processed is out of the cache, a replacement policy is used to swap the state in cache. As for the DSA strategy, the TD, HC and MM algorithms can be implemented based on the AVC strategy or as a combination of both AVC and DSA. Such combinations would certainly lead to more algorithms to be studied in order to capture their domain of optimality. We further discuss yet another strategy below.

2.3 The State pre-Ordering Strategy

During FA-based string processing, it may happen that the string being tested frequently visits only a small part of the whole transition graph. Also, the overall number of states visited during acceptance testing may also be well below the automaton's number of states. In order to handle such as case, it is necessary to have a mechanism for reorganizing the transition graph so that frequently accessed states are grouped together, thus optimizing the performance of the recognizer. This enables frequently visited states to be put next to each other so as to reduce the total number of cache misses at run-time. The State pre-Ordering (SpO) strategy addresses this issue by making use of a pre-processing function to reorder the position of the automaton's states before any recognition takes place. It assumes that the implementer has some foreknowledge of an appropriate ordering in a given context.

In practice, one may have to deal with FAs of considerable size in which only a limited number of states are frequently accessed most of the time. Furthermore, these frequently visited states could be spread throughout the transition table such that page swaps occur when accessing state's information. The SpO strategy would be recommended if it is envisaged that the same pattern of state visitation is likely to occur over and over again. In this case, the order in which states are visited should somehow be assessed. To this end, as a first step, a function could be incorporated into whichever core algorithm is being used. The job of this function would be to keep track of the order in which states are visited. After running one or more acceptance tests in the conventional fashion, this function could be used to pre-order the state information in memory, to be thus used for future acceptance testing. The SpO strategy will incur overhead costs, depending on whether the ordering of states takes place before acceptance testing (preprocessing) or during acceptance testing (run-time). Such overhead costs need to be offset against the gains to be made by increasing the cache hit probability. The strategy would be advantageous under circumstances where, for example, pre-ordering occurs on a once-off basis (or periodically, but relatively infrequently and according to changing circumstances) while many acceptance testing runs take place after each pre-ordering. Again as for the previous strategies, we can rely either on the TD, HC or MM algorithm to implement the SpO strategy. SpO, DSA, and AVC can indeed be combined together in order to produce more algorithms that will form part of the class-library to be discussed in the next section. Therefore, one may choose to implement for example a TD algorithm based on any of the above strategies and combinations thereof. Theoretically, a taxonomy tree can be constructed based on the combination of the above mentioned implementation strategies [7].

The end-result is presented in the form of a taxonomy graph made of the following components:

- *The root node* represents the starting point of the

taxonomy graph. In the present case, it represents a simple specification of the problem. That is, a specification of the transition sets (TD and/or HC). The root node is therefore, not an algorithm, but rather a specification of the problem.

- *An abstract node* is a child node in the taxonomy that cannot be instantiated. In order words its algorithm cannot be derived. However an abstract node is always the parent node of some concrete node discussed below.

- *A Concrete node* is a concrete implementable algorithm. Concrete nodes are not necessarily leaf nodes in the taxonomy graph; they may be parents to various concrete/abstract nodes in the graph.

- *The relationship* between a parent node and a child node specifies the derivation rule applied on the parent node in order to obtain the child node. In the taxonomy, the refinement rules are the strategies applied on parents in order to obtain children.

The taxonomy graph suggested here is not final in the sense that many other implementation strategies may be suggested in order to produce several new algorithms not discussed here. Figure 1 depicts the taxonomy graph. The root node labeled FA represents the problem specification, more precisely that of the transition sets (HC and TD). The root node is further refined into three different children according to the nature of the transition sets provided. When the FA is specified with the HC transition set empty, the derived algorithm is that of the table-driven (t). If provided with the TD transition set empty, the derived algorithm is that of the the HC algorithm (h). However, if both of the transition sets are non-empty, the derived algorithm is that of the MM algorithm (m).

The three children of the root node represent the traditional (core) algorithms discussed in Section 2. Further refinement may be used in either of the nodes at that level to produce various algorithms. Dashed edges on various nodes in the graph reflect the possibility to add more algorithms based on new implementation strategies that could be discovered. In the graph, nodes in dark represent concrete algorithms, whereas the others are nondeterministic algorithms of which concrete implementations are provided by either of the children. The overall approach used to derived algorithms in the graph can be summarized as follows: *At a given node, investigate possible refinement strategies to be used for possible derivations. If one exists, then apply it to the node and draw the derived children. Repeat the the same process on all the nodes in the graph.* For example, the derivation of the TD algorithms is given below (the same applies for HC and MM algorithms):

1. *Algorithm* t is derived from the root node. It is obtained if and only if the associated table-driven transition set is non-empty and the hardcoded transition set is empty.

2. *Algorithms* t_1, t_2, t_3 are obtained from algorithm t by applying the DSA, SpO, and AVC strategies respectively. Two of the algorithms (t_2 and t_3) are concrete whereas algorithm t_1 is abstract. Therefore, further refinement is necessary in order to obtain concrete algorithms derived from t_1. In the taxonomy graph, t_2 is a terminal node since no further refinement strategy has been found in order to produce new algorithms from t_2.

3. *Algorithms* t_{b1}, *and* t_{u1} are derived from t_1. They are concrete in the sense that they represent the bounded and unbounded implementations of the table-driven algorithm based on the DSA approach. The two algorithms may further be refined to produce new algorithms.

4. *Algorithm* t_{23} is obtained from t_3 by applying the SpO strategy. In the same way, *algorithm* t_{123} is derived from t_3 by applying simultaneously DSA and SpO strategies. The algorithm is not concrete since the DSA strategy is nondeterministic. Further refinement is needed to obtain concrete algorithms from t_{123}

5. *Algorithm* t_{b123} *and* t_{u123} are derived from the abstract node t_{123}. They are concrete algorithms that exploit the bounding nature of the DSA strategy. t_{b123} uses simultaneously the bounded DSA, SpO and AVC on table-driven FA-based string recognizers, and t_{u123} uses simultaneously the unbounded DSA, SpO and AVC on TD.

6. *Algorithms* t_{b12}, *and* t_{b13} are derived from the node t_{b1}; they are respectively the combination of the concrete bounded DSA and SpO strategies, as well as the the bounded DSA and the AVC strategies.

7. *Algorithms* t_{u12}, *and* t_{u13} are described in the same fashion as the previous bounded algorithms, with the difference that they are based on the unbounded DSA strategy.

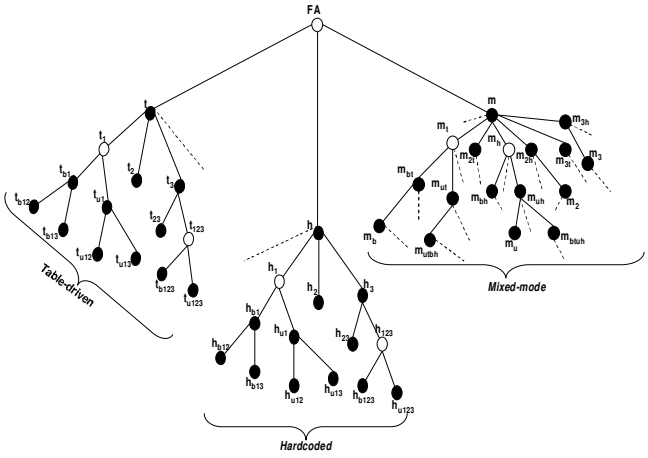

Figure 1: A taxonomy of FA-based String Processing Algorithms.

Having provided for a taxonomy graph depicting all algorithms derived from implementation strategies, the main challenge remains the utility of each of the algorithms obtained. Clearly, this exercise can only be done if there is a consolidated platform for facilitating cross-comparisons amongst algorithms. Before presenting the platform, we briefly discuss in the next section a high level architecture of a toolkit (class library) obtained from the taxonomy.

3. A CLASS-LIBRARY FOR STRING RECOGNIZERS

The library is a self-contained package of implementable algorithms that can be used by any external application that requires string recognition to satisfy some or other computational need. For example, a system for network intrusion detection may be regarded as a potential client of the library, since such an application typically needs to test whether a given string pattern is part of the language modeled by a well specified automaton. Moreover, the library can be used for educational and research purposes by supporting experimentation and benchmarking of the various algorithms.

The following are well-known FA-based toolkits (class-libraries):

- The **Grail** system [13]. Its primary aim is to facilitate teaching and research of language theory. It is used to perform various operations on finite automata and regular expressions such as: automata minimization, conversion from regular expressions to finite automata (and vice-versa), etc.
- The **Amore** system [3]. It is an implementation of the semigroup approach to formal language. It provides various routines for manipulating regular expressions, finite automata and semigroups. Its aim is to explore efficient implementation of algorithms for solving theoretical problems in formal language research.
- The **Automate** system [1]. The toolkit is used for symbolic computation of automata such as automata construction, minimization and transformations. Its primary intention was to be used for teaching and research.
- The **FIRE Engine**[15]. This is an implementation of all the algorithms that appear in the taxonomy of regular expression algorithms [16], and was primarily intended for teaching. A somewhat smaller version referred to as **FIRE lite** is proposed in [16]. The aim of **FIRE lite** was to provide a variety of algorithms to the user who in turn can use them according to their efficiency. Users interested in algorithms' inner structure may refer to **FIRE lite** not only for the understanding of the system's design, but also for various research that may lead to new algorithms.
- The **SPARE parts** system [17] is a string pattern recognition toolkit designed in C++. It is a library of various implementations of pattern matching algorithms obtained from the taxonomy of pattern matchers.

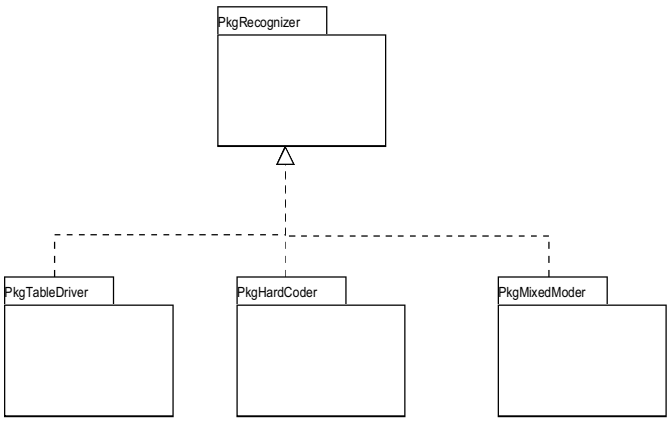

Figure 2: A high-level toolkit's view based on interacting packages.

The class-library's architectural design described below is not a complete ready-to-use package. Its efficient exploitation would rely on the availability of a framework for the purpose of identifying appropriate algorithms to be used in a given domain based on the automaton's structure.

3.1 The Architectural Design

Here, we depict the library's architecture in a top-down fashion, first providing a high level view of the architecture and then discussing each of the lower level components. The library may be regarded as a set of interacting packages which in turn are made of interacting classes that encapsulate *string processing algorithms*. Based on the previously described traditional algorithms (HC,TD and MM), we can view at a higher level our library as a system consisting of the following packages:

- The `PkgRecognizer` encapsulates the problem's transition set and the input string. The package interacts with:
- the `PkgTableDriver` which encapsulates the various table-driven algorithms that were obtained by using the various implementation strategies to modify the core table-driven algorithm;
- the `PkgHardCoder` which consists of the various hardcoded algorithms that are derivatives of the original hardcoded algorithm; and
- the `PkgMixedModer`, which encapsulates the derivatives of the MM core algorithm characterised by various combinations of the strategies.

Figure 2 depicts such a high-level view of the toolkit. It shows a *dependency* relationship between the *root package* and its children. This means that each class in each of the sub-packages inherits from a base class in the root package. However, there will be only one class (that we shall refer to as `Recognizer`) in the root package, and this will be considered as the root class in the whole toolkit's class-diagram. We explicitly make reference to `PkgRecognizer` to emphasize that the various other classes necessary for the complete specification of an FA-based

string recognizer are dependent on the root class. The overall class-diagram may thus be regarded as a system made up of a root class representing the specification of the problem domain, from which all other classes down the hierarchy inherit. The subsections below elaborate on each package of the system, discussing the structure of each of the classes within the package, their relationships with other classes, as well as the description of their attributes and operations.

3.2 The Package PkgRecognizer

Figure 3 depicts the class-diagram that make up `PkgRecognizer`. The package consists of the following classes: `Recognizer`, `State`, `Transition` and `AlphabetObject`.

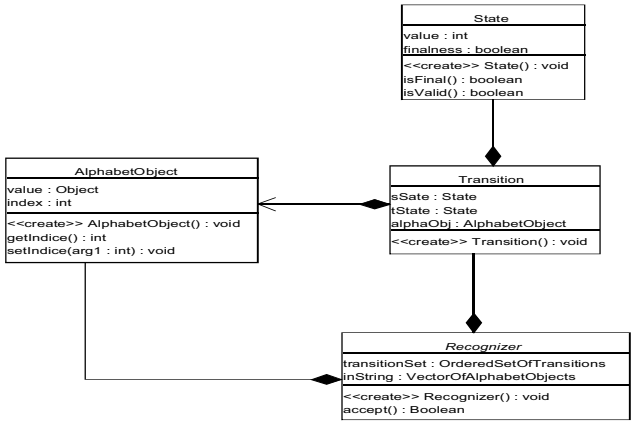

Figure 3: The class diagrams of PkgRecognizer.

In practice, a *recognizer* may be regarded as a system that receives as input a *string* and a *transition set*, and then performs *acceptance* testing on the input data, returning a boolean. This definition identifies not only the recognizer object, but also its attributes and operations. It follows that, the first class in the package must be the class `Recognizer`, that contains two attributes, *inString* and *transitionSet*, as well as an operation *accept*(). The attribute *transitionSet* of the class `Recognizer` is an ordered set of transitions. Using a set to represent the transitions guarantees that there are no duplicate elements. Although not strictly necessary, we specify the set as ordered for easy information retrieval based on sequential or direct access. A transition is a triplet of the form $(sState, alphaObj, tState)$. The source state ($sState$) and the target state ($tState$) are both objects of type `State`. Unlike a target state which may be a rejecting state, a source state is never a rejecting state. The class `State` is made of two attributes $valState$ of type integer, and $finalness$ of type boolean. A negative $valState$ is construed to mean that the state is a rejecting state. For a positive $valState$ (i.e. a valid state), $finalness$ attribute indicates whether the state is a final state or not. Beside the constructor, the copy constructor, and the destructor operations defined on the class, various other operations such as: $getVal()$ that returns the value of a state, $isFinal()$ that checks whether a state is final or not, and $isValid()$ that checks the validity of a state may be defined on the class.

The class `Transition` requires a constructor, a copy constructor, as well as a destructor. Each instance of `Transition` is used to build the transition set of a `Recognizer`.

The following relationships hold between the classes `Recognizer`, `Transition`, and `State`: A `State` *is part of* a `Transition` which in turn *is part of* a `Recognizer`. This kind of relationship is referred to as a *composition* relationship.

In order to trigger a transition from a source state to a target state, an alphabet object (conventionally referred to as a symbol in practice) is required. The choice for using an alphabet object rather than a simple character is to simply accommodate those problems whose alphabets are not simple symbols.

The attribute *alphaObj* in `Transition` is an instance of a class `AlphabetObject` containing *value* and *indice* as attributes. The attribute *indice* references the order of the alphabet element, and the attribute *value* represents the actual alphabet element which is an object. An Alphabet object *is part of* a transition; the scenario reflects the composition relationship between the two classes.

A datatype `AlphabetSet` (not present in the diagram) may be used to hold instances of `AlphabetObject`; the set inherits all operations related to a `Set` class, and it represents the alphabet of the finite automaton.

The class `Recognizer` requires an input string in order to perform acceptance testing. In this context, the attribute *inString* of the class may be regarded as sequence of alphabet objects, or put differently, a *vector* of alphabet objects. In practice, a vector datatype is less rigid than a set in the sense that is accomodates duplication. Since a string is part of a recognizer, and a string is made of alphabet objects, we may simply say that an alphabet object *is part* of a recognizer. The relationship between the class `Recognizer` and the class `Alphabet` is thus a composition relationship.

As shown in the diagram, an instance of a recognizer contains several instances of a transition, and several instances of an alphabet. In turn, an instance of a transition is made of two instances of a state and one instance of an alphabet. All classes in the diagram contain their constructor, and additional operations may be added as needed. The *accept*() operation in this class is abstract (virtual in C++) so that the operation can be explicitly defined in inheriting classes. Therefore, the class `Recognizer` is just an abstract class and cannot be instantiated. Various operations such as that of counting the total number of states of the automaton, the automaton's alphabet size and the like may be explicitly defined within the class `Recognizer`. Such operations may be considered useful in ensuring that the construction of objects down the hierarchy are well defined.

The description in the next subsection is limited to the table-driven package; the other two packages are structurally similar and are detailed in [10].

3.3 The Package PkgTableDriver

Figure 4 depicts the class diagram contained in the table-

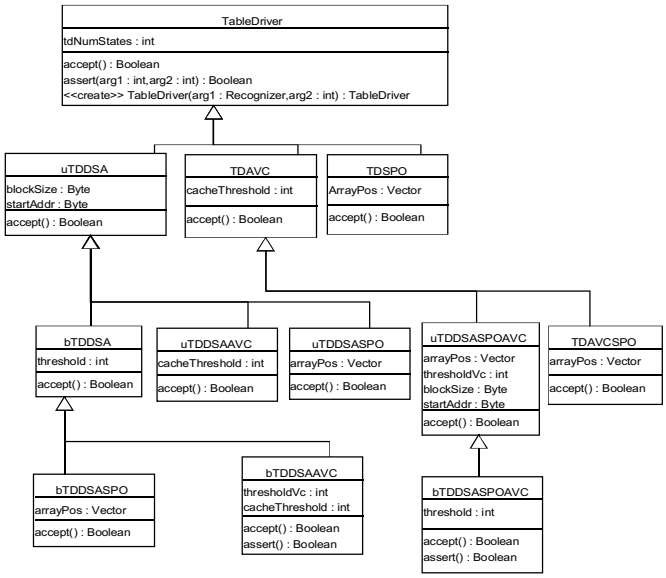

Figure 4: The table-driven class diagram.

driven package. The `TableDriver` class is the base class in the package, and all other classes directly or indirectly inherit from it. `TableDriver`, implements the core table-driven algorithm in the operation *accept*(). The class inherits all attributes of `Recognizer` necessary for its instantiation. After constructing an instance of `TableDriver` say `TD` the statement `TD.accept()` returns a boolean to indicate whether the input string is part of the language modeled by the FA or not. As shown in Figure 4, classes are given in three levels of the hierarchy. Each level of the hierarchy will now be discussed below.

3.3.1 The first level of the TD class-diagram

Three classes, namely `uTDDSA`, `TDAVC`, and `TDSPO` directly inherit from the `TableDriver` class. The last two classes were obtained by direct mapping from the taxonomy tree's nodes t_3 and t_2 respectively. The `TDAVC` class specialises the `TableDriver` class, but supports input string processing that is based on the allocated virtual caching strategy. It inherits all attributes of table-driven (as well as those of `Recognizer` indirectly) but requires an additional attribute *cacheThreshold* to specify an integer-valued threshold indicating the last state that falls within the virtual cache, starting from state 0. Thus, acceptance testing takes place between state 0 and state *cacheThreshold*; and reference to any state out of that portion requires state replacement. The AVC strategy requires that the threshold be strictly less than the total number of states of the automaton. An operation $isValid()$ is thus required in the class for checking the validity of the fundamental condition of AVC ().

The class `TDSPO` implements the state pre-ordering strategy. It directly inherits from its base class and also inherits indirectly all attributes of the class Recognizer. The class is specialized by the attribute *arrayPos* which is a vector of the new positions of the states of the automaton. While constructing an instance of the class, a preprocessing operation is used to allocate the states according to the specified positions in *arrayPos*. As for the `TDAVC` class, an operation such as $isValid()$ may be required to ensure that the new positions of the states have indeed been provided in $arrayPos$.

The class `bTDDSA` implements the bounded DSA strategy; it requires the following specialized attributes: *blockSize* that holds the size (in bytes) of the memory block to be used for dynamic state allocation; *startAddr* that holds the starting address (in bytes) in memory for dynamic states allocation; and finally, *threshold* that holds the maximum number of states to be dynamically allocated in memory. This last attribute reflects the bounded nature of the class indicating that state replacement may be required when the threshold has been reached.

For the class `uTDDSA`, the fact that it is unbounded means that there is no limit to the number of states to be dynamically allocated in memory. Therefore, only the first two attributes of the `bTDDSA` class would be required in addition to those of the `TableDriver` and the `Recognizer` classes.

The class `bTDDSA` may thus be regarded as a specialized class of the class `uTDDSA`, which in turn may be regarded are a derived class of `TableDriver` in the absence of the abstract class `TDDSA`.

For the construction of an instance of `uTDDSA`, an operation $isValid()$ is required in order to ensure that the attribute *blockSize* matches with the total number of states of the FA. A simple way to evaluate the match would be by multiplying the size of a state (in bytes) by the total number of the automaton's states and comparing the result with *blockSize*. Furthermore, its consistency must be checked on whether the address held by *startAddr* is a valid memory address or not.

3.3.2 The second level of the TD class-diagram

Four classes are derived from classes in the first level of the table-driven hierarchy; namely the `bTDDSA`, the `uTDDSAAVC`, the `uTDDSASPO`, the `uTDDSASPOAVC`, and the `TDAVCSPO`.

Our design choice has made it possible to consider the class `bTDDSA` (which relies on the table-driven based on the bounded DSA strategy) as a specialized class of `uTDDSA`. The class corresponds to the node t_{b1} of the taxonomy graph. It inherits all attributes and operations of `uTDDSA` and requires its own implementation of the operation *accept*(), as well as an attribute *threshold* that enforces its specialization towards its base class. The attribute holds the maximum number of states to be dynamically allocated. It follows that an operation $isValid()$ is required such that, when instantiating an object of the class, a validity check is made to ensure that the value that has been assigned to *threshold* is strictly less than the total number of the FA states, in line with the basic condition underlying the bounded DSA strategy [10].

The class `uTDDSAAVC` corresponds to the node t_{u13} in the taxonomy tree. It corresponds in practice to the implementation of the table-driven based on both the unbounded DSA and the AVC strategy simultaneously. The class may be considered as a specialization of both `uTDDSA` and `TDAVC`, which suggests that it is a candidate for multiple inheritance. Alternatively, it can be considered

as a specialization either of `uTDDSA` or of `TDAVC`. In the diagram, we have chosen to make it inherit directly from `uTDDSA`. The attribute *cachThreshold* indicates its specialization in respect of its base class. As for the other classes, an operation such as $assert()$ is required to check whether the value assigned for construction of an object of that type is valid according to the basic condition that makes up the implementation strategy on which the class relies.

The class `uTDDSASPO` corresponds to the node t_{u12} in the taxonomy tree. Again, as for the `uTDDSAAVC` class, the `uTDDSASPO` class may be derived from either `uTDDSA` or `TDSPO`. We choose to make it a specialized class of `uTDDSA`. The class requires an attribute *arrayPos* whose validity would be checked at construction time using the operation $isValid()$.

The class `uTDDSASPOAVC` corresponds to the node t_{u123} of the taxonomy graph. It holds the implementation of the combination of the unbounded DSA strategy and the other two strategies. We may allow this class to multiply inherit from `uTDDSA`, `TDSPO`, and `TDAVC`. However, we have chosen to make it a subclass of `TDAVC` only, so as to stick to our single inheritance convention. To achieve this, the following attributes are required: *arrayPos* that holds the new positions of the states for state reordering purpose; *thresholdVc* that holds the total number of states to be processed in the virtual cache; *cacheThreshold* that holds the size of the virtual cache; *blockSize* that holds the size of the memory to be dynamically allocated; and *startAddr* that holds the address where the first state will be dynamically allocated in memory. An operation such as $isValid()$ is required while constructing an object of the class since it is used to ensure that the values assigned to the attributes respect the conditions under which the algorithm may be used. That is: $0 < cacheThreshold < ThresholdVc < tdNumStates$, and $arrayPos \neq \emptyset$. Of course, the remaining number of states to be processed based on the DSA strategy should match with the values assigned to the attributes *blockSize* and *startAddr*.

The class `TDAVCSPO` that corresponds to the node t_{23} in the taxonomy tree may inherit from both `TDSPO` and `TDAVC`. We chose to have it as a specialized class of `TDAVC`. In order to do so, the attribute *arrayPos* is required in the specialized class to hold the new positions of the state for preprocessing purpose. The $isValid()$ operation is used at construction time to ensure that the array is indeed provided.

3.3.3 The last level of the TD class-diagram

At this level, only three classes are available. They are respectively `bTDDSASPO`, `bTDDSAAVC`, and `bTDDSASPOAVC`.

The class `bTDDSASPO` corresponds to the node t_{b12} of the taxonomy tree. In our diagram, it is considered as a subclass of `bTDDSA`. Alternatively, we could have chosen to make it a subclass of `TDSPO`, or as deriving from both classes. The class is made of the attribute *arrayPos* that holds the new position of the states required during preprocessing for reordering the states. The directly executable table-driven algorithm based on both bounded DSA and SPO strategy may be generated at construction time.

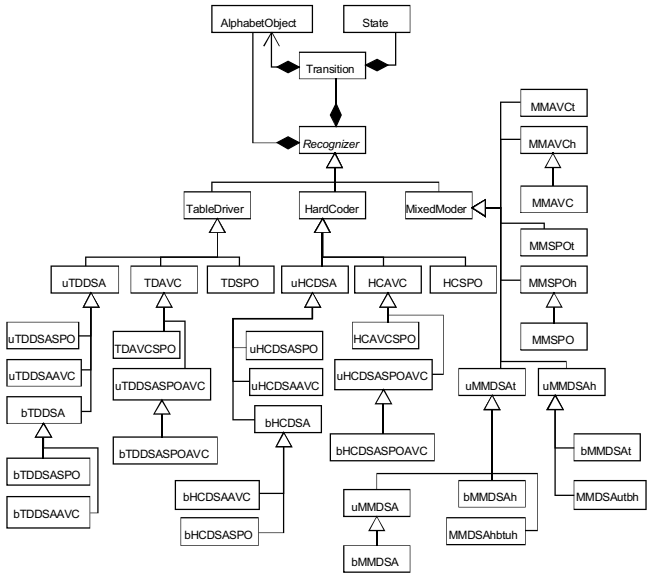

Figure 5: An extract FA-based String Recognizers class-diagram.

The class `bTDDSAAVC` corresponds to the node t_{b13} of the taxonomy graph. It is a subclass of `bTDDSA`, and requires the following attributes: *thresholdVc* is which an integer that holds the total number of cacheable states; and *cacheThreshold* that holds the size of the virtual cache. The bounded nature of the class requires that replacement could also be performed during dynamic allocation of states.

The last class in the diagram is `bTDDSASPOAVC` which corresponds to the node t_{123} of the taxonomy tree. It is a subclass of `uTDDSASPOAVC`. The class has a method that implements the bounded version of its base class. Its specialization in relation to its base class is materialized by the attribute *threshold* that holds the maximum number of states to be dynamically allocated for states that have been chosen to be processed using the bounded DSA strategy. This enables to perform state replacement in the dynamically allocated memory when the threshold has been reached. The construction of an instance of the class is therefore subject to the assignment of a valid value to the attribute *threshold*. That is, a value less than the total number of the automaton state, and also the total number of states to be processed through dynamic state allocation.

Figure 5 depicts the overall architectural view of the toolkit derived from the taxonomy tree suggested at the end of the previous section.

The class-library as is, could not enjoy an efficient exploitation by unexperienced users. It is in fact practically impossible for a user to have sound knowledge of the behaviour of each and every single algorithm in the library. Clearly, there is a need to develop a consolidated platform whereby each algorithm could be tested and evaluated in order to capture the appropriate domain where they can be exploited at optimum. We introduce in the next section the framework for the efficient evaluation of algorithms available in the library.

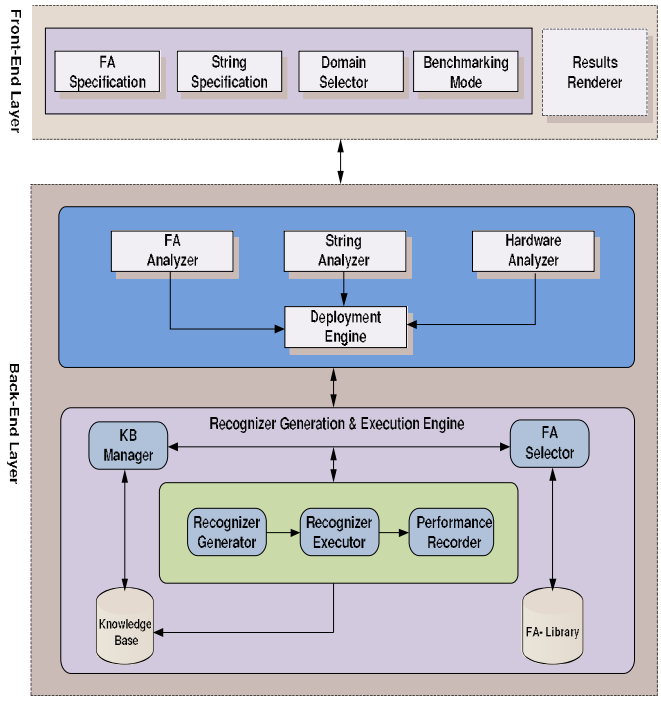

Figure 6: The framework for benchmarking recognizers

4. THE SUGGESTED FRAMEWORK

In this section, we provide at a high level our suggested framework which consists of a range of components that facilitate the selection and classification of all algorithms in the above library. As depicted by Figure 6, the framework consists of two fundamental layers: the front-end layer and the back-end layer. Each of the foregoing layers and their building blocks are described along the following lines.

4.1 The Front-end Layer

This layer is where implementers provide information on the kind of automaton required. It also serves as a placeholder for rendering the results (performance-wise) of analyzed algorithms in order to enable the user to make an informed choice on the algorithm of interest. We have omitted in this layer to explicitly mention the UI merely because during the implementation of the overall framework, the programmer can choose to make it command line in which case several steps should be followed before invoking the back-end for analysis. Of course the front-end functionality could be encapsulated in a GUI as it is advantageous to unexperienced users. Each of the building blocks of the front end layer are described below.

4.1.1 FA Specification

This component facilitates the specification of the automaton to be used. Users may optionally provide the complete FA (i.e. its transition table, starting state, alphabet size symbols, the set of states and final states, and the transition set in the form (q_i, s_k, q_j) where q_i is the current state, s_k a symbol of the alphabet and q_j the destination state). Once all characteristics of the FA have been provided, an exception handler is invoked to ensure that the specified FA is deterministic.

Alternatively, the user may request that an FA be generated. In this case, the system should be able to randomly generate the FA based on input information such as the number of states, the size of the alphabet, the number of final states and the density (sparsity) [8] of the automaton's transition table. Of course one could also give the ability to the system to randomly generate all/any of the foregoing parameters.

4.1.2 String Specification

This building block enables the user to provide the set of strings to be used for benchmarking. The string set can be generated randomly by the system or the user can also provide a set of string to be used for testing by the automaton. An exception handler checks whether the specified string symbols are restricted to the automaton's alphabet. For random generation, the user should specify the length of the strings to be generated, as well as the number of strings required.

4.1.3 Domain Selector

This building block enables the user to make a selection on the computational domain to be studied. Most popular domains include network intrusion detection, tandem repeat finders, computational biology, natural language processing, natural and computer virus scanning. However, the system should be able to accommodate any additional domain as per user requirements. This suggests that a table maintaining the list of domains should be available in the system during implementation.

4.1.4 Benchmarking mode

As previously discussed, the standard algorithm for FA implementation is known to be the TD algorithm. If the mode is set to standard, the system should then evaluate the performance of algorithms in the library against the TD. In some cases one may need to use any other algorithm as the standard. The benchmarking mode building block accommodates such choice where the user selects its own algorithm as standard for cross-comparison. Another mode option is the default option where no standard is provided therefore, the system only perform cross-comparison amongst all algorithms in the library.

4.1.5 Result Renderer

The role of the Result Renderer is to output results of the analyzed FA in various formats (plotting graphs, pie-charts, histograms, etc.). It provides classifying and ranking algorithms according to their performance. It also maintains a direct connection with the system's knowledge base (discussed below), ensuring historical data

is maintained. The Result Renderer also keeps track of the domain of concern since this is available in the knowledge base. This allows domain-related benchmarking to be collected and analysed.

Note that the front end layer is not just a sort of UI layer. It contains (not shown in the figure) a range of engines that serve to ensure that consistency is maintained when providing input. It is only when specified data is consistent that it can be deployed in the back-end for analysis.

4.2 The Back-end Layer

The back-end layer consists of two major components: The Input Analyzer and Deployment Engine (IADE), and the Recognizer Generation and Execution Engine (RGEE). Both engines are described below.

4.2.1 The Input Analyzer and Deployment Engine

Data sent to the back-end layer is guaranteed to be of good quality, as per our previous discussion. However, more analysis needs to be performed for the purpose of the classification of the results produced after executing the Recognizer based on a given automaton. The IADE plays this role in three major aspects.

These are briefly described below:

- ***FA and String Analyzers*** These enable the study of both the FA and the string provided as input. They inform on whether the given input is large or not, whether the FA provided is sparse or not, or whether the algorithm to be analyzed relies on small strings/FA or not. These verdicts are based on thresholds set in the system, either at implementation time, or by the user during input specifications at the front end.
- ***Hardware Analyzer*** It collects information about the hardware on which the FA will be processed. Since the current library was constructed having in mind the memory size of the computational medium, the hardware analyzer collects memory related information such as the overall memory size of the hardware, the cache levels (L1, L2, L3,...) in the hardware as well as their respective sizes. This will inform the implementer of the possibilities for experimentation with algorithms such as AVC, where the amount of memory to be used is pre-specified.
- ***Deployment Engine*** All information gathered from the front end as well as from the above described analyzer are now sent to the RGEE in a structured way for the actual performance analysis. The deployment engine's role is then to deploy the specified information to the RGEE in an appropriate format.

  This information will then be used for processing/generating relevant algorithms in the library and to record their performance. (Of course data resulting from the hardware analyzer is mostly used for informing the user on the capabilities of the computational medium under consideration.)

4.2.2 The Recognizer Generation and Execution Engine

Once the specified FA and all its basic elements are sent to this engine, the main job is now to iteratively launch each algorithm and record its performance. The recorded results are then updated in the KB before being despatched to the front-end (Results Renderer). We briefly describe the main components of this engine below:

- ***Knowledge-base and KB Manager*** The idea of a Knowledge Base (KB) in the present context was suggested in [9]. The KB can be regarded as a repository of performance-related information about algorithms in the toolkit. Furthermore, it may also have information about the computational domain, the characteristics of the FA and string size as well as the hardware used to record such a performance. The KB should then maintain a complex data structure subdivided in terms of computational domain, platform, and input characteristics. In addition, performance data of recognizers based on the foregoing characteristics should be maintained in the knowledge based.

  This approach enables the system to avoid duplicating performance analysis in the sense, that, a user providing information that already exist in the KB will received results without going through all the components in the system. In this case, the IADE will not perform any work and information will only be extracted from the KB for rendering purpose. Work is currently being done on the characterization of the KB and will be reported in due course.

  The KB manager serves as intermediary between the front end and the KB. It also interacts with the FA selector component in order to keep track of recognizers already processed at run-time. Upon receiving a specification from the front-end, the KB manager checks whether the specified FA has not yet been analyzed under the specified condition. If that is indeed the case, then the corresponding results are extracted from the KB and sent back to the Result Renderer. If not, the FA selector component is invoked and appropriate recognizers in the library are selected for performance recording and analysis.
- ***FA Selector and FA-library*** The role of the FA-selector is straightforward—to extract from the FA-library an appropriate recognizer upon a request from the KB manager. As previously mentioned, algorithm selection is only needed if the performance data of such an algorithm is not available in the KB under the specified conditions. The selector extracts from the FA-library a given algorithm and sends it to the execution engine for further processing. We have previously discussed the overall architecture of our class library.
- ***Generator, Executor and Performance Recorder*** Algorithms whose performance is not yet available in the KB, need to be generated, executed and processed for performance recording. This part of

the back end is devoted to such activity. The generation of an algorithm entails the creation of directly executable instructions that process the recognizer. Examples of such a process was discussed in [6] where the DSA algorithm was extensively studied. When the system is implemented and deployed for the first time, many of the algorithms in the library will have to undergo the process described in this part of the back-end in order to have their information/data available in the KB. Again, in order to avoid keeping the engine very busy, only those algorithms whose performance have not yet been collected in a given domain and based on a given characteristic will require the invocation of this part of the system

The overall framework as described above encapsulates all the necessary tools required for its actual implementation. The implementation of such a system is still under construction and will be published in due course.

5. CONCLUSION AND FUTURE WORK

In this paper, we have introduced a framework for benchmarking FA-based string recognizers for performance enhancement. The suggested framework relied on the availability of a range of algorithms in the literature that need to be studied in order to establish their strengths and weaknesses. The suggested framework is foreseen to be considered as a platform for performance evaluation of FA-based string processing algorithms. To date, algorithm implementers have generally relied on the so-called conventional approach for implementing FA. The availability of a working framework will certainly contribute to exploring other approaches to implementing FAs and domain specific optimal algorithms could be discovered and used in the implementation of optimal solutions. As future work, we need to fully characterize the KB and produce a working system for benchmarking FA based string processor for performance enhancement.

6. REFERENCES

[1] J. M. Champarnaud and G. Hansel. Automate: A computing package for automata and finite semigroups. In G. Rozenberg and A. Salomaa, editors, *Journal of Symbolic Computation*, volume 12, pages 197–220, 1991.

[2] M. Crochemore and C. Hancart. Automata for matching patterns. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages*, volume 2, pages 399–462. Springer-Verlag, 1997.

[3] V. Jansen, A. Pothoff, W. Thomas, and U. Wertmuth. *A Short Guide to the Amore System*, volume 90. Aachener Informatik-Berichte, 1990.

[4] E. N. Ketcha. Hardcoding finite automata. Master's thesis, Department of computer Science, Pretoria 0002, South Africa, November 2003.

[5] E. N. Ketcha, D. G. Kourie, and B. W. Watson. Reordering finite automatata states for fast string recognition. In *Proceeding of the Prague Stringology Conference*, Prague, Czech Republic, August 2005. Czech Technical University.

[6] E. N. Ketcha, D. G. Kourie, and B. W. Watson. Dynamic allocation of finite automatata states for fast string recognition. *International Journal of Foundation of Computer science*, 2006.

[7] E. N. Ketcha, D. G. Kourie, and B. W. Watson. A taxonomy of dfa-based string processors. In *Proceeding of the SAICSIT Conference*, pages 111–121, Gordon's Bay, South Africa, October 2006. ACM.

[8] E. N. Ketcha, B. W. Watson, and D. G. Kourie. Preliminary experiments in hardcoding finite automata. In *Proceeding of the 10th Conference on Implementation and Application of Finite Automata*, pages 299–300, Santa Barbara, CA, USA, July 2003. Springler.

[9] E. N. Ketcha, B. W. Watson, and D. G. Kourie. A framework for the dynamic implementation of finite automata for performance enhancement. In *Proceeding of the Prague Stringology Conference*, Prague, Czech Republic, August 2004. Czech Technical University.

[10] E. K. Ngassam. *Towards Cache Optimization in Finite Automata Implementations*. PhD thesis, Department of Computer Science, Pretoria, South Africa, November 2006.

[11] E. K. Ngassam, D. G. Kourie, and B. Watson. Performance of hardcoded finite automata. *Software Practice and Experience*, 36(5):525–538, 2006.

[12] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design*. Morgan Kaufmann, third edition, 2005.

[13] D. R. Raymond and D. P. Wood. The grail papers: Version 2.0. *Technical Report University of Waterloo, Canada*, January 1993.

[14] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6):323–350, 1968.

[15] B. W. Watson. The design and implementation of fire engine: A c++ toolkit for finite automata and regular expressions. *Technical Report University, Technical University of Eindhoven*, 1994.

[16] B. W. Watson. *Taxonomies and toolkits of regular Languages Algorithms*. PhD thesis, Faculty of Mathematics and Computer Science, Eindhoven, the Nederlands, September 1995.

[17] B. W. Watson and L. Cleophas. Spare parts: A c++ toolkit for string pattern recognition. *Technical Report University, Technical University of Eindhoven*, 34(7):697–710, 2004.