Part II

FA-based String Processing Algorithms

CHAPTER 3

CORE ALGORITHMS

In this chapter, the conventional algorithm for implementing FA-based string recognition, referred to as the table-driven (TD) algorithm, is briefly revisited. It is then followed by a brief overview of the hardcoded (HC) algorithm, a relatively new FA-based string recognition algorithm in which the FA's transition function is implemented as simple instructions which replace the transition table. Both algorithms are combined to introduce a new algorithm, referred to as the mixed-mode (MM) algorithm that partially relies on hardcoded instructions and partially on a table for acceptance testing. The chapter then provides the denotational semantics for these string recognizers. This is a formal characterization which will be the foundation for describing various implementation strategies in later chapters. Here the TD, HC and MM algorithms are formally described. Their formal description is fundamental to the description process, and in this sense, these three algorithms are collectively referred to as the "core" algorithms.

3.1 The table-driven algorithm

Traditionally, FA-based string recognizers are implemented using the table-driven algorithm. In this case, the transition function is represented in the form of a table (two-dimensional array) whose columns represent the symbols of the alphabet and rows the states of the automaton. The table is therefore an implementation of the function $\delta(q, c_j)$ that is associated with the FA at issue.

For convenience, and without loss of generality, it will be assumed that states are integers in the ranges $[-1, |\mathcal{Q}|)$. State -1 corresponds to the sink state that indicates rejection, and need not be represented as a row in the table. Each remaining state, q, corresponds to the q^{th} table row. By convention, 0 corresponds to the start state.

We also assume that the FA's alphabet, $\mathcal{V}$, is an ordered set such that:

$$\forall j : [0, |\mathcal{V}|) \centerdot (\exists c_j \in \mathcal{V}) \text{ and } c_j \text{ corresponds to the } j^{th} \text{ column of the table.}$$

Thus the state returned by the transition function $\delta(q, c_j)$ is the entry in the table at the intersection of row q and column j. Furthermore, we will assume that the notation $\delta[q]$ or simply δ_q refers to the q^{th} row of the table; that is, all transitions triggered by every symbol of the alphabet at state q, which is in fact the set

$$\{\forall c_j \in \mathcal{V}, j \in [0, |\mathcal{V}|) \centerdot \delta(q, c_j) \in \mathcal{Q}\}.$$

A simple driver function is then used to traverse the table, such that, at the end of the scanning operation, it is known whether the string is part of the language modelled by the FA or not. Algorithm 3.1.1 gives the pseudo-code for the table-driven algorithm.

The algorithm is generic in the sense that it does not rely on one particular transition table: the same algorithm can be used for acceptance testing of *any* input string s in respect of *any* transition function δ. It will be seen later, that the hard-coded counterpart algorithm to this table-driven algorithm does not enjoy this generic property.

Algorithm 3.1.1 (Table-driven string recognizer)

---

func $td(\delta, s)$: **boolean**
  $q, j := 0, 0$;
  do $(j < s.len()) \wedge (q \geq 0) \rightarrow$
    $q, j := \delta(q, s_j), j + 1$
  od;
  return $(q \geq 0)$
cnuf

Remark 3.1 (on algorithms in this thesis). Throughout the thesis, the algorithms are given in Dijkstra's GCL (Guard Command Language). They are simple enough to be understood without the need of explanatory annotations in terms of *invariants*, *preconditions*, or *postconditions*. The reader may refer to sources such as [Dij76] and [DF88] for a more advanced coverage of these topics.

The performance of the TD algorithm is memory load dependant. As more memory space is needed to hold the table, inefficiencies arise [Nga03]. Due to random accesses that are made into the table, as the table size increases there is a higher probability of cache misses [NKW05a].

An alternative to TD is the direct representation of the table in hardcoded form, using simple conditional statements. The next section discusses such an implementation.

3.2 The hardcoded algorithm

Hardcoding an algorithm means to embed the data on which the algorithm relies as part of its instructions. The algorithm was first suggested in the late 60's by Ken Thompson in [Tho68] for fast implementation of regular expression search algorithms. In Thompson's algorithm the entire transition table that make up the automaton's transition function, (and therefore the regular expression represented by the automaton), was implemented using simple conditional statements. Thompson's approach was later applied on FA-based pattern matching of strings by Knuth et al. in [KMP77]. In both cases, hardcoding appeared to be more efficient than the traditional table-driven approach. However, the algorithm lacked proper performance analysis in order to quantify the extent to which it may be more efficient than TD, and the conditions under which improvements occur.

Nonetheless, for the past tree decades, hardcoding has been experimentally used in compiler construction, and more precisely in parsing [Pen86, FH91, AU73, GJ91, PD04] for performance enhancement, and in some cases memory load improvement. A more elaborate investigation of hardcoding through cross-comparison with its TD counterpart in terms of alphabet size and automaton's number of states was undertaken by Ketcha in [Nga03]. These results showed that HC can sometimes be more efficient than TD for automata of relatively small size (in the order of a few hundred states). An explanation of this, is that the HC algorithm takes advantage of computer caching capabilities, resulting in relatively low probability of cache misses, and hence better processing speed.

Unlike the TD algorithm, the HC algorithm is not generic, since prior knowledge of the transition function is necessary in order to generate the hardcoded algorithm. Therefore, in practice, given a transition function, a preprocessing function has to be invoked that generates the directly executable hardcoded algorithm from the transition function. One may choose to first generate the hardcoded algorithm in a high level language before following all the compilation and linkage operations necessary to produce the directly executable hardcode.

Another alternative is to embed the preprocessing operation (the generator) in the same program, provided that one has the ability to produce self-modifying code [Cra03, Hyd03, VM03]. In this case, having the transition matrix and the string to be tested, a hardcoded program is generated and further executed within the same program. Since all blocks of instructions that make up a hardcoded recognizer are similar in size and number of basic instructions, it is easy to estimate and then reserve memory space needed to contain the generated instructions.

The amount of memory to be reserved depends on the automaton size as well as the size of instructions that will be used. For illustration, let assume that the variable, *top*, points to the position in memory where the first instruction of the hardcoded algorithm is to be written. Then, after generating all the hardcoded instructions and writing them to memory, we can redirect the program counter to *top*. This operation then results in the execution of the hardcoded algorithm with appropriate output (i.e. $true$ or $false$).

Algorithm 3.2.1 called *hcg* gives pseudo-code for generating such a hardcoded recognizer[1], and then executing it in respect of an input string, s. As input, the generator program takes the transition function δ, the starting address of the generated instructions *top*, the number of non-sink states of the FA $|Q| = n$, and the number of alphabet symbols $|\mathcal{V}| = a$. It also takes in as input, the input string for acceptance testing, s. However, this string is not used at all in setting up the hardcode, which could also be subsequently used for acceptance testing some other input string.

[1]For illustrative purposes, the algorithm here is shown to generate GCL code. Of course, the actual implementation discussed in [Nga03], involves the generation of assembler code that is generated by a program written in C++.

Algorithm 3.2.1 (Generation and direct execution of a hardcoded string recognizer)

func $hcg(\delta, top, n, a, s)$: **boolean**
  $B := top;$
  $gen(\text{"}q, j := 0, 0;\text{"}, B);$
  $gen(\text{"}\mathbf{do}\ (j < s.len()) \wedge (q \geq 0) \rightarrow \text{"}, B);$
  $i := 0;$
  do $i < n \rightarrow$
    if $i = 0 \rightarrow gen(\text{" }\mathbf{if}\ q = 0 \rightarrow \text{"}, B)$
    $[\!]\ \ i \neq 0 \rightarrow gen(\text{"}[\!]\ q = \text{"}, B);\ gen(i, B);\ gen(\text{" } \rightarrow \text{"}, B)$
    fi;
    $k := 0;$
    do $k < a \rightarrow$
      if $k = 0 \rightarrow gen(\text{" }\mathbf{if}\ s_j\ = c_0 \rightarrow \text{"}, B)$
      $[\!]\ \ k \neq 0 \rightarrow gen(\text{" }[\!]\ s_j\ = \text{ "}, B);\ gen(c_k, B);\ gen(\text{" } \rightarrow \text{"}, B)$
      fi;
      $gen(\text{"}q, j := \text{ "}, B);\ gen(\delta(i, c_k), B);\ gen(\text{", } j + 1\text{"}, B);$
      $k := k + 1$
    od;
    $gen(\text{" }\mathbf{fi}\text{ "}, B);$
    $i := i + 1$
  od;
  $gen(\text{" }\mathbf{fi}\text{ "}, B);$
  $gen(\text{"}\mathbf{od;}\text{ "}, B);$
  $gen(\text{"}\mathbf{return}\ (q \geq 0)\text{"}, B);$
  $exec@(top, s)$
cnuf

The various functions and operations referred to in the algorithm operate as follows:

- Variable B is used to indicate the start address of the next instruction to be written.
- $gen(i, B)$ writes the instruction or part of the instruction designated by i into memory, starting at address B. Thus, in some cases below, i will be a string that contains keywords and variable names which are fixed relative to the *hcg* algorithm —for example $gen(\text{"}\mathbf{if}\ q = 0 \rightarrow \text{"}, B)$. In other cases, i will be a variable of the *hcg* algorithm itself, such as $gen(s_i, B)$. In both cases, it is assumed that after writing the instruction portion at the specified memory address, then B is appropriately updated so that it points to the memory location where the next part of code should be written.

- As before, c_j designates the j^{th} symbol in the ordered alphabet $\mathcal{V}$.
- $exec@(top, s)$ sets the program counter to the address top and the execution of the written code starts there. We include s as a parameter here, merely to emphasise that it is only at this stage that the input string comes into play —i.e. once the hardcode has been set up.

It would perhaps be helpful to briefly look ahead to Algorithm 3.2.2 as an example of the kind of code which the *hcg* algorithm is intended to generate. At the start of the *hcg* algorithm, the variable B is initialized to the initial address *top*. The generator then writes the code related to variable initialization as well as setting up of a loop structure. Note that this loop is set up to have a counter j.

Then follows in *hcg* a double loop which set up nested **if** statements of the form:

$$
\begin{array}{l}
\textbf{if } q = 0 \rightarrow \\
\quad \textbf{if } s_j = c_0 \rightarrow q, j := \delta(0, c_0), j + 1 \\
\quad [\!] \;\; s_j = c_1 \rightarrow q, j := \delta(0, c_1), j + 1 \\
\quad [\!] \;\; \cdots \\
\quad \textbf{fi} \\
[\!] \;\; q = 1 \rightarrow \\
\quad \textbf{if } s_j = c_0 \rightarrow q, j := \delta(1, c_0), j + 1 \\
\quad [\!] \;\; s_j = c_1 \rightarrow q, j := \delta(1, c_1), j + 1 \\
\quad [\!] \;\; \cdots \\
\qquad \cdots \\
\quad \textbf{fi}
\end{array}
$$

The outer loop of *hcg* sets up the statements of the outer **if** statement, there being one guarded command for each (non-sink) state of the FA, i.e. for $q = 0 \cdots n - 1$. The inner loop of *hcg* sets up the inner **if** statements, each guarded command now corresponding to a test whether the j^{th} input symbol, s_j, matches the next character of the FA's alphabet.

At the end of *hcg*'s outer loop, code related to the test on whether the string is part of the language modelled by the FA or not is written in memory. The generator ends by transferring into the program counter the address *top* where the first hardcoded instruction was written, using the previously described $exec@(top, s)$ function. Once the program counter is assigned the address *top*, hardcoded acceptance testing starts. In the next subsection, we use an illustrative example to depict the generated hardcoded recognizer based of this approach.

3.2.0.1 An Example

Consider the automaton modelled by the state diagram in Figure 3.1. Its transition function δ is shown in Table 3.1. This transition function can also be represented as triples that constitute the following set:

$$\Delta = \left\{ \begin{array}{llll} (0,d,5), & (0,i,5), & (0,o,5), & (0,\mathrm{v},1), \\ (1,d,5), & (1,i,5), & (1,o,2), & (1,\mathrm{v},5), \\ (2,d,5), & (2,i,3), & (2,o,5), & (2,\mathrm{v},5), \\ (3,d,4), & (3,i,5), & (3,o,5), & (3,\mathrm{v},5), \\ (4,d,5), & (4,i,5), & (4,o,5), & (4,\mathrm{v},5), \\ (5,d,\text{-}1), & (5,i,-1), & (5,o,-1), & (5,\mathrm{v},-1) \end{array} \right\}$$

	d	i	o	v
0	5	5	5	1
1	5	5	2	5
2	5	3	5	5
3	4	5	5	5
4	5	5	5	5
5	−1	−1	−1	−1

Table 3.1. The transition table Δ of the TD recognizer of example 3.2.0.1

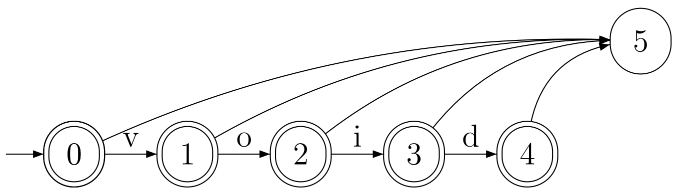

Figure 3.1. A State diagram that accepts the string *void*

If Algorithm 3.2.1 is invoked as $hdg(\delta, top, 6, 4, \text{“}void\text{”})$ where *top* is some starting address, then the instructions in Algorithm 3.2.2 will be generated.

Algorithm 3.2.2 (Pseudocode HC recognizer for a given transition function)

```
func hc(s) : boolean
   q, j := 0, 0;
   do (j < s.len) ∧ (q ≥ 0) →
       if q = 0 →
          if s_j = 'd' →
             q, j := 5, j + 1
          ▯ s_j = 'i' →
             q, j := 5, j + 1
          ▯ s_j = 'o' →
             q, j := 5, j + 1
          ▯ s_j = 'v' →
             q, j := 1, j + 1
          fi
       ▯ q = 1 →
          .......
       ▯ q = 2 →
          .......
       ▯ q = 3 →
          .......
       ▯ q = 4 →
          .......
          q, j := 5, j + 1
       ▯ q = 5 →
          .......
          q, j := −1, j + 1
       fi
   od;
   return (q ≥ 0)
cnuf
```

It is worth mentioning that in order to save space, some lines of the generated code may be combined together using the boolean OR operator when the next state to be transited to is identical for some of the symbols of the alphabet. A slightly more elaborate code generating program would be needed for this. In such a case, the size of the block of instructions used to check the next state to be transited to, will be reduced to fewer conditional branches. In the present example at state 0, the first three branches could be combined in one, since the symbols 'd', 'i', and 'o' send the FA to the same *next state*, namely state 5.

A potential drawback of the hardcoded algorithm (Algorithm 3.2.2) is the number of instructions in the algorithm. As the automaton size (i.e. the transition table) grows, the number of blocks required to hardcode a state grows as well. This results in inefficiencies by increasing the likelihood of instruction cache swaps, as discussed in [Nga03].

The TD and HC strategies can be combined together to produce an algorithm that partially relies on each of the method. The resulting algorithm will be referred to as the mixed-mode algorithm. It is discussed in the next section.

3.3 The mixed-mode algorithm

Implementing an FA-based string recognizer in a mixed-mode fashion is construed to mean that the states in the transition table are partitioned into two disjoints sets. One of the sets is represented as a table and the other is hardcoded. If the next state to be examined during acceptance testing is a hardcoded state, then the next transition is determined by the hardcoded part of the recognizer. Otherwise, the next state is determined by the table-driven algorithm.

At this point, the criterion for partitioning the set of states is not at issue. Various partitioning policies could be devised for different circumstances.

For presentation purposes, a number m is assumed as a threshold for splitting the states into HC and TD states. States that are numbered less than m are considered as hardcoded states and those above m are table-driven states.

Algorithm 3.3.1 depicts such a mixed-mode version of a string recognizer that takes δ and m as input, as well as the input string s. In the algorithm, the following functions are used for the hardcoded part:

- $genhc(\delta[0..m))$ generates the hardcode corresponding to the first m states (rows) of the transition function. However, we assume that the code is not executed after being generated.

- $exec\!hc(q, s_j)$ executes this hardcode of the mixed-mode recognizer. It takes as parameters the current state q and the current symbol s_j and returns a new value for q.

Algorithm 3.3.1 (Mixed-mode string recognizer)

---

func $mm(t, m, s)$: **boolean**
    $genhc(\delta[0..m))$;
    $q, j := 0, 0$;
    do $(j < s.len()) \wedge (q \geq 0) \rightarrow$
        if $q < m \rightarrow q, j := execPhc(q, s_j), j + 1 \;[\!]\; q \geq m \rightarrow q, j := \delta(q, s_j), j + 1$ **fi**
    od;
    return $(q \geq 0)$
cnuf

For illustration purpose, an applied version of the mixed-mode algorithm can be provided. In order to do so, we consider the implementation of the running example using the mixed-mode strategy, taking m as 3. Therefore, the transition set

$$\Delta = \left\{ \begin{array}{llll} (0,d,5), & (0,i,5), & (0,o,5), & (0,\mathrm{v},1), \\ (1,d,5), & (1,i,5), & (1,o,2), & (1,\mathrm{v},5), \\ (2,d,5), & (2,i,3), & (2,o,5), & (2,\mathrm{v},5), \\ (3,d,4), & (3,i,5), & (3,o,5), & (3,\mathrm{v},5), \\ (4,d,5), & (4,i,5), & (4,o,5), & (4,\mathrm{v},5), \\ (5,d,-1), & (5,i,-1), & (5,o,-1), & (5,\mathrm{v},-1) \end{array} \right\}$$

is split into two subsets

$$\Delta_h = \left\{ \begin{array}{llll} (0,d,5), & (0,i,5), & (0,o,5), & (0,\mathrm{v},1), \\ (1,d,5), & (1,i,5), & (1,o,2), & (1,\mathrm{v},5), \\ (2,d,5), & (2,i,3), & (2,o,5), & (2,\mathrm{v},5), \end{array} \right\}$$

and

$$\Delta_t = \left\{ \begin{array}{llll} (3,d,4), & (3,i,5), & (3,o,5), & (3,\mathrm{v},5), \\ (4,d,5), & (4,i,5), & (4,o,5), & (4,\mathrm{v},5), \\ (5,d,-1), & (5,i,-1), & (5,o,-1), & (5,\mathrm{v},-1) \end{array} \right\}$$

Δ_h represents the subset of Δ that should be hardcoded, and Δ_t that of table-driven. Algorithm 3.3.2 gives the notional idea of how the mixed-mode implementation of the running example would evolve into code. During acceptance testing, part of the transition matrix represented by a table is accessed by a driver function whereas the hardcoded part is directly executed.

Algorithm 3.3.2 (An applied mixed-mode string recognizer)

```
func mm(δ, 3, s) : boolean
   genhc(δ[0..3]);
   q, j := 0, 0;
   do (j < s.len()) ∧ (q ≥ 0) →
       if q > 3 → q, j := δ(q, s_j), j + 1
       ▯ q ≤ 3 →
          if q = 0 →
             if s_j = 'd' ∨ s_j = 'i' ∨ s_j = 'o' → q, j := 5, j + 1
             ▯ s_j = 'v' → q, j := 1, j + 1
             fi
          ▯ q = 1 →
             if s_j = 'd' ∨ s_j = 'i' ∨ s_j = 'v' → q, j := 5, j + 1
             ▯ s_j = 'o' → q, j := 2, j + 1
             fi
          ▯ q = 2 →
             if s_j = 'd' ∨ s_j = 'o' ∨ s_j = 'v' → q, j := 5, j + 1
             ▯ s_j = 'i' → q, j := 3, j + 1
```

```
            fi
         fi
      fi
   od;
   return (q ≥ 0)
cnuf
```

Having described the core string recognition algorithms in terms of pseudocode, the simple formal functional model, previously proposed as an abstract characterization of string recognition algorithms, will be revisited and applied to the core algorithms, TD, HC and MM.

3.4 Functional description of core recognizers

In this section, we give a revised version of an FA-based string recognizer's denotational semantics, this time focussing on the characteristics of the core FA-based string recognizers. To do this, we shall rely on the material in Chapter 2, where a string recognizer ρ was defined as a function whose arguments are a transition function (Δ) and an input string (s) that is to be tested for acceptance. The function notation lends itself naturally to describing other FA string recognition strategies. Each of those strategies are discussed in detail in chapters 4, 5, and 6.

Recall from Chapter 2 (Equation 2.22) that a string recognizer's denotational semantics, ρ, was defined in terms of the following possibly partial function:

$$\rho : \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q}) \times \mathcal{V}^* \nrightarrow \mathbb{B}$$

$$\rho(\Delta, s) = \begin{cases} true & \text{if } s \in \mathcal{L}(\mathrm{M}) \\ false & \text{if } s \notin \mathcal{L}(\mathrm{M}) \end{cases}$$

This definition is somewhat general —no particular detail is provided on the practical implementation of any associated string recognition algorithm. The definition can be refined to reflect attributes of the MM algorithm, where TD and HC emerge as special cases. We proceed as follows:

Let Δ_t denote the transition set that is used in the TD part of an MM implementation. Similarly, let Δ_h be the transition set that is used in the HC part of an MM implementation. Clearly, Δ_t and Δ_h must be a partition of the original transition set, Δ, i.e. $\Delta_t \cup \Delta_h = \Delta$ and $\Delta_t \cap \Delta_h = \varnothing$. An example of such a partition has already been seen in Section 3.3.

Now let ρ_C be the function to represent the recognizer based on one of the core algorithms, TD, HC or MM. Letting $\mathcal{T} = \mathcal{P}(\mathcal{Q} \times \mathcal{V} \times \mathcal{Q})$, this function can be defined as follows:

$$\rho_{_C} : \mathcal{T} \times \mathcal{T} \times \mathcal{V}^* \rightarrow \mathbb{B} \tag{3.2}$$

such that

$$\text{if } \ (\Delta_t \cup \Delta_h = \Delta) \wedge (\Delta_t \cap \Delta_h = \varnothing) \ \text{ then } \rho_{_C}(\Delta_t, \Delta_h, s) = \rho(\Delta, s)$$

The function highlights the fact that TD and HC can be viewed as two limiting cases of MM. Thus, $\rho_{_C}(\Delta, \varnothing, s)$ represents the TD algorithm applied to string s, while $\rho_{_C}(\varnothing, \Delta, s)$ represents the HC algorithm applied to s.

Remark 3.3 (on the splitting of the transition set). It is also worth mentioning that such a generalization of a recognizer in term of mixed-mode implicitly entails the following:

- The TD and HC set of states ($\mathcal{Q}_t$ and $\mathcal{Q}_h$ respectively) are disjoints and their union is the set of states $\mathcal{Q}$ of the automaton. Therefore the following holds:

$$\begin{cases} \mathcal{Q}_t \cap \mathcal{Q}_h = \varnothing \\ \mathcal{Q}_t \cup \mathcal{Q}_h = \mathcal{Q} \end{cases}$$

- The TD and HC set of final states ($\mathcal{F}_t$ and $\mathcal{F}_h$ respectively) are disjoints and their union is the set of final states $\mathcal{F}$ of the automaton. Therefore, the following holds:

$$\begin{cases} \mathcal{F}_t \cap \mathcal{F}_h = \varnothing \\ \mathcal{F}_t \cup \mathcal{F}_h = \mathcal{F} \end{cases}$$

There is no particular attention to be given to the FA's starting state since it may fall under either TD or HC set of states. Of course no splitting is permitted on the set of alphabet symbols as we are still concern with the same FA.

Having given the algorithms and provided for a formal characterization ($\rho_{_C}$) of the three core string recognizers, we now explore various *strategies*. Each of these strategies can be embedded into the core algorithms and each can also be described in a further refinement of our functional description. Furthermore, from each of the new characterization explored, we can still derive the original ones i.e. HC, TD and MM. Chapters 4, 5, and 6 are each devoted to one of these strategies.

3.5 Summary of the Chapter

In this chapter, we have revisited the conventional TD FA-based string processing algorithm as well as the relatively new implementation technique referred to as hard-coding. From both strategies, a new algorithm (mixed-mode) was suggested, whereby both TD and HC are combined to produce a single FA-based recognizer. Based on the MM, TD and HC algorithms, we further provided a unified formal definition of

FA-based string recognizer that represents the mixed-mode algorithm. It was also shown that from the formal characterization of a mixed-mode algorithm, the TD and HC algorithms can be derived without loss of generality.

Having provided such a formal characterization of FA-based recognizers, it becomes possible to express various strategies for acceptance testing in terms of the general formalism. Each of the strategies are discussed in the next three chapters. In each case, it is shown that the core implementation techniques can be obtained as a special case, from the strategies discussed. Thus, using each strategy, new algorithms are designed that have TD, HC and MM as their root strategies. The characterization therefore serves as basis for a taxonomy of FA-based string recognizers discussed in Chapter 7.

The first strategy is referred to as "dynamic state allocation". It is discussed in the next chapter.

CHAPTER 4

DYNAMIC STATE ALLOCATION

This chapter discusses a set of new algorithms for FA-based string processing. They will be referred to as the Dynamic State Allocation (DSA) algorithms. A DSA algorithm seeks to exploits the notion of spatial and temporal locality of reference on which cache memory relies in order to improve on the performance of the core FA-based algorithms discussed in the previous chapter.

This present chapter starts off by providing a formal characterization of a DSA algorithm by introducing additional arguments into the previously defined functions. It is shown that, through simple instantiations of the proposed DSA strategy arguments, the functional description specialises to a description of the core algorithms.

By the DSA strategy, we mean the implementation of either the TD, HC or MM algorithm based on a dynamic state allocation concept that will be discussed explicitly in Section 4.1 below. Discussions of TD, HC and MM algorithms based on the suggested DSA strategy then follow. Also provided in this chapter is an illustrative example that relies on the new table-driven dynamic state allocation algorithm, as well as a theoretical assessment of the new algorithms compared to their core counterparts.

4.1 DSA Characterization

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.

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

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

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. Of course, the realisation of such benefits is highly dependent on the input string's state path.

In order to describe the new strategy in a refined form of the functional description given in Chapter 3, one needs to introduce an argument to represent the DSA strategy. The argument then informs the reader on the extent to which the strategy has been adopted. This can range from not having been adopted at all, in which case the argument should be 0; to having been adopted for every possible state visited along the state path, in which case the argument should be set to $n = |\mathcal{Q}|$, where $|\mathcal{Q}|$ is the FA's number of states.

However, since the general formalism of a string recognizer contains both the HC and TD algorithms separately, we should in fact introduce *two* arguments: D_t and D_h. The first is a natural number that refers to the extent to which the TD part of the algorithm is based on the DSA strategy. Likewise, the second refers to the extent to which the HC part of the algorithm is based on the DSA strategy. Thus, when the strategy variables D_t and D_h are both non-zero, then the recognizer corresponds to an MM algorithm following the DSA strategy. As per usual, in such a case, the TD and HC transition sets have to constitute a partition of the automaton's transition set.

Thus, two scenarios are envisaged with respect to each of the TD/HC component of the algorithm:

- In the *unbounded dynamic state allocation* scenario, the relevant strategy variable is equal to the maximum number of states (i.e. $D_t = |\mathcal{Q}_t|$ or $D_h = |\mathcal{Q}_h|$). In this case, the dynamic allocation occurs as new states that have not yet been dynamically allocated in the new memory space are encountered. Therefore, in a worst case situation it is possible to have the size of the newly allocated memory equal to that of the originally used memory. All that has changed is that the state ordering in the newly allocated memory is guaranteed to be organized on a contiguous fashion according to the string path.

- In a *bounded dynamic state allocation* scenario, a relevant strategy variable is strictly less than the maximum number of states, but also greater than zero ($0 < D_t < |\mathcal{Q}_t|$ or $0 < D_h < |\mathcal{Q}_h|$). In this case, the algorithm only has a limited number of states to be allocated dynamically in memory. The restriction means that not all states need necessarily be represented in the new memory location when processing a string whose string path requires more states than those allocated.

Note that a bounded DSA strategy requires a *replacement policy* —i.e. a policy about whether and how to replace states in the dynamic space. In this thesis, we shall assume the *direct mapping* replacement policy. For this policy, when the dynamic space is full and reference is made to a state that has not yet been visited, the new state is assigned an address in the dynamic space based on the modulus operation used to identify the state to be removed from the dynamic space. Of course, there could be various other replacement policies such as: the Least Recently Used (LRU) policy whereby, state in allocated memory is removed, replacing it with the least recently invoked state; the associative mapping and the set associative mapping [Hay98, PH05], but these will not be further explored here.

Thus, given a predefined $0 < D_t < |\mathcal{Q}_t|$, the DSA algorithm allocates up to D_t states in memory according to the string path. If all the D_t have been allocated, and the string processing is not yet completed, upon accessing a state that has not yet been visited, we use the *direct mapping policy* to remove a state from the memory and put the state currently being processed. In contrast, if the strategy was provided such that $D_t = |\mathcal{Q}_t|$, no replacement strategy will be needed: the algorithm would be straightforward in the sense that states would be allocated in the new dynamic space as they are encountered, provided that they have not yet been visited.

We can now define a function which accounts for a possible DSA strategy implementation to implement FA-based string recognition. Call this function ρ_{CD}, the $_C$ subscript indicating that it can express any of the three core algorithms, and the $_D$ subscript indicating that it also accommodates the DSA strategy. The function is thus defined as follows:

$$\rho_{_{CD}} : \mathcal{T} \times \mathcal{T} \times \mathbb{N} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B} \tag{4.1}$$

such that

$$\text{if } \begin{cases} (\Delta_t \cup \Delta_h = \Delta) \wedge (\Delta_t \cap \Delta_h = \varnothing) \\ (0 \leq D_t \leq |\mathcal{Q}_t|) \wedge (0 \leq D_h \leq |\mathcal{Q}_h|) \end{cases} \text{ then } \rho_{_{CD}}(\Delta_t, \Delta_h, D_t, D_h, s) = \rho(\Delta, s)$$

With the above formalism, various properties may be used in order to provide restrictions on the usage of the strategies variables. The subsection below discusses the properties of the DSA strategies.

4.1.1 Properties of the DSA strategy

1. Since the DSA strategies provided in the characterization are natural numbers, one of the obvious properties is its relation to the total number of states of the automaton. The strategy variables should never exceed that number of states, that is: $D_t \leq |\mathcal{Q}_t|$ and $D_h \leq |\mathcal{Q}_h|$.

2. As a consequence of the foregoing, when the transition set associated with a strategy variable is empty, then the strategy variable has to be zero; that is,

$$\begin{cases} |\mathcal{Q}_t| = 0 \Rightarrow D_t = 0 \\ \text{and} \\ |\mathcal{Q}_h| = 0 \Rightarrow D_h = 0 \end{cases}$$

3. The denotational semantics of the core TD, HC and MM algorithms can be expressed in terms of this new characterization. Assuming, as before, that Δ_t

and Δ_h partition the transition set, Δ, of the FA under consideration, then the following relationship holds:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_C}(\Delta_t, \Delta_h, s) \equiv \rho_{_{CD}}(\Delta_t, \Delta_h, 0, 0, s).$$

Of course, the previous specialisations continue to apply as before. For example, the denotational semantics of TD, previously given as $\rho_{_C}(\Delta, \varnothing, s)$ can alternatively be stated as $\rho_{_{CD}}(\Delta, \varnothing, 0, 0, s)$, etc.

4. When $\Delta_t = \Delta$ (thus $\Delta_h = \varnothing$) and $D_t > 0$, then the resulting algorithm will be referred to as the TD-DSA algorithm. Its semantics is therefore:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_{CD}}(\Delta_t, \varnothing, D_t, 0, s) = \rho(\Delta, s), \text{ with } D_t \neq 0.$$

 The TD-DSA algorithm is said to be *bounded* or *unbounded* depending on whether $0 < D_t < |\mathcal{Q}_t|$ or $D_t = |\mathcal{Q}_t|$, respectively.

5. Similarly, when $\Delta_h = \Delta$ (thus $\Delta_t = \varnothing$) and $D_h > 0$, then the resulting algorithm will be referred to as the HC-DSA algorithm. Its semantics is therefore:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_{CD}}(\varnothing, \Delta_h, 0, D_h, s) = \rho(\Delta, s), \text{ with } D_h \neq 0.$$

 In this case, the HC-DSA algorithm is said to be *bounded* or *unbounded* depending on whether $0 < D_h < |\mathcal{Q}_h|$ or $D_h = |\mathcal{Q}_h|$, respectively.

6. The MM-DSA algorithm refers to the general case, i.e. when Δ is partitioned by Δ_t and Δ_h, and $(D_t > 0) \vee (D_h > 0)$. The following specifies the denotational semantics of various instances of the MM-DSA algorithm:

$$\forall s : \mathcal{V}^* \centerdot$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CD}}(\Delta_t, \Delta_h, D_t, 0, s) \\ D_t \neq 0 \end{cases} \tag{4.2}$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CD}}(\Delta_t, \Delta_h, 0, D_h, s) \\ D_h \neq 0 \end{cases} \tag{4.3}$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CD}}(\Delta_t, \Delta_h, D_t, D_h, s) \\ D_t \neq 0 \wedge D_h \neq 0 \end{cases} \tag{4.4}$$

An MM-DSA algorithm that is described by Equation 4.2 is called *HC-free*. In the case of Equation 4.3 it is called *TD-free*.

An MM-DSA algorithm is *TD-bounded* (resp. *HC-bounded*) if $0 < \Delta_t < D_t$ (resp. $0 < \Delta_h < D_h$). Similarly, the MM-DSA algorithm is *TD-unbounded* (*HC-unbounded*) if $\Delta_t = D_t$ ($\Delta_h = D_h$ respectively). All these variations are special cases of Equation 4.4.

Thus, an MM-DSA algorithm may be implemented in a variety of ways. For example, it may be TD-bounded (or TD-unbounded) and HC-free (or HC-bounded, or HC-unbounded). In the case of being both TD-free and HC-free, it “degenerates”, of course, into the core MM algorithm.

There are thus a variety of DSA algorithms that could be studied, implemented, and cross-compared in term of performance with their core counterparts. Part III is devoted to a performance analysis of a representative set of these algorithms. In the following sections pseudo-code is provided for the TD-DSA, HC-DSA and MM-DSA algorithms respectively.

4.2 The TD-DSA algorithm

As discussed in Section 4.1.1, two scenarios are accounted for the design and implementation of the TD-DSA algorithm: that is, the bounded and the unbounded case. For each scenario under consideration, an algorithm could be derived. As has been seen, the TD-DSA algorithm is described by the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho(\Delta, s) = \rho_{_{CD}}(\Delta_t, \varnothing, D_t, 0, s)$$

Where the algorithm is bounded or not according to whether ($0 < D_t < |\mathcal{Q}_t|$ or $D_t = |\mathcal{Q}_t|$).

An earlier account of the TD-DSA algorithm was given in [NKW05b], and an improved version was provided in [NKW06b]. However, in both these publications, the algorithms presented only catered for the unbounded scenario. In this section, we provide a generalized version of the algorithm that accounts for both bounded and unbounded scenarios. We use a simple conditional statement to differentiate between the two situations.

The TD-DSA algorithm is based on the premise that, during acceptance testing, upon entering a new state (a state that has not yet been visited), a block of memory is created to which the state's transition information is copied. Acceptance testing takes place within the newly allocated memory location. Unlike the TD algorithm, the TD-DSA algorithm requires three basic parameters: the input string, the transition table and a natural number that represents the bound for dynamic allocation. It is assumed that the bound is non-zero. A high-level specification of the TD-DSA algorithm is depicted in Algorithm 4.2.1 below. The variables used in the algorithm are as follows:

$n = |\mathcal{Q}_t|$: the number of states;

$a = |\mathcal{V}|$: the number of alphabet symbols;

D_t: the threshold for dynamic states allocation (bound);

s: the input string; whose symbol at position j is s_j;

j: the index of s indicating the next symbol s_j to be scanned;

δ: the transition function; the state returned by the transition function $\delta(i, s_j)$ is the entry in the transition table at the intersection of row i and column j; it is denoted δ_{i,s_j};

A: the start address in memory where information about dynamically allocated states is stored;

d: a specially reserved place in memory, indicated by the dynamic two-dimensional array, each row of which corresponds to a dynamically allocated state[2]; the i^{th} entry of d will be denoted d_i;

$m_{[0:n-1]}$: an auxiliary array, whose i^{th} entry (denoted m_i) is $k \in [0, p)$ if memory for the state corresponding to the i^{th} row in the transition table has been dynamically allocated to the k^{th} row of table d; and is -1 otherwise;

q: a reference to the row in the transition table representing the next state to be investigated, or offset by m_q if it refers to a row representing a state in the dynamically allocated table, d;

p: the index of the next row of d to be dynamically allocated;

B: points to the next memory address where space for the entry d_p is to be allocated;

Z: the amount of space to be reserved for each dynamically allocated state;

$search(array, var)$: a simple function that returns the position of the element var in $array$. The returned value is used to replace the state currently being processed by the state currently at the position referenced to by the returned value.

o: a state to be replaced when the algorithm is based on the bounded DSA strategy. For the present algorithm, we use the modulus operation for replacement. Therefore, once the row r to be replaced in d is calculated (that is, $r = MOD(q, D_t)$), a search operation is made on m with r in order to find the state (o) to be replaced. It is guaranteed that such state will be found. After the state has been found the entry m_o is switched to -1 before replacement.

Algorithm 4.2.1 (The TD-DSA algorithm)

---

func $tddsa(\delta, D_t, A, Z, s)$: **boolean**
  if $D_t < n \rightarrow$ {bounded dynamic allocation of states}
    $m_{[0:n-1]} := -1$;
    $B, q, j, p := A, 0, 0, 0$;
    { $inv(p)$ }
    do $(j < s.len() \wedge q \geq 0) \rightarrow$
      if $m_q = -1 \rightarrow$ {state not dynamically allocated}
        if $p < D_t \rightarrow$
          $m_q := p; d_p := malloc(B, Z)$;
          $d_{p,[0..a-1]} := \delta_{q,[0...a-1]}$;

[2] In an earlier description of some of the work described in this thesis [NKW05b], dynamically allocated states were referred to as "reordered states".

$q, p, B := d_{p,s_j}, p+1, B+Z$
$\| \ p \geq D_t \rightarrow$
$r := MOD(q, D_t);$
$o := search(m, r); m_o := -1;$
$m_q := r;$
$d_{r,[0..a-1]} := \delta_{q,[0..a-1]};$
$q := d_{r,s_j}$
fi
$\| \ m_q \neq -1 \rightarrow$ {state dynamically allocated}
$q := d_{m_q,s_j}$
fi;
$j := j+1$
od
$\| \ D_t = n \rightarrow$ {unbounded dynamic allocation of states}
$m_{[0:n-1]} := -1;$
$B, s, j, p := A, 0, 0, 0;$
do $(j < in.len() \wedge q \geq 0) \rightarrow$
if $m_q = -1 \rightarrow$ { state not dynamically allocated}
$m_q := p; d_p := malloc(B, Z);$
$d_{p,[0..a-1]} := \delta_{q,[0..a-1]};$
$q, p, B := d_{p,s_j}, p+1, B+Z$
$\| \ m_q \neq -1 \rightarrow$ **skip** {state dynamically allocated}
$q := d_{m_q,s_j}$
fi;
$j := j+1$
od
$\| \ D_t > n \rightarrow$ **skip**
fi;
return $(q \geq 0)$
cnuf

The loop invariant, $inv(p)$, that characterises the main loop of the algorithm, is the predicate defined below:

$$inv(p) \triangleq \forall i : [0, n) \bullet (m_i = -1) \vee (m_i \in [0, p) \wedge \forall j : [0, a) \bullet (\delta_{i,j} = d_{m_i j})$$

This loop invariant articulates the nature of m, namely that the i^{th} entry of m is either -1, or it lies in the range $[0, p)$. In the latter case, the state corresponding to row i of the original transition table has been dynamically allocated row m_i of table d.

A *select* (i.e. **if-**) command in the body of the loop determines whether the current state q refers to a dynamically allocated state in d or a state in the original table.

This is done by examining the value of m_q. If $m_q = -1$ then q has to be dynamically allocated and the next value of q is determined directly from table d. Otherwise, the next value of q is determined directly from row m_q of table d.

The algorithm handles both bounded and unbounded versions of TD-DSA algorithm. In order to do this, a *select* command is used to deal with the maximum number of states that may be dynamically allocated in memory. In the unbounded case, ($D_t = |\mathcal{Q}_t|$) all states that fall within the string path are dynamically copied to a new memory location for acceptance testing, provided that they have not yet been visited. For $D_t < |\mathcal{Q}_t|$, the algorithm is bounded; therefore a replacement policy is used to remove a state that has already been processed from the dynamic allocated space, and copy the state currently being processed. This situation occurs when the threshold of space to be allocated has been reached. In order to do space replacement, a simple modulus operation is used. However, improved techniques such as that of the Least Recently Used (LRU) strategy could be envisaged.

Following the same principle as that discussed in this section, the HC-DSA algorithm is discussed in the next section.

4.3 The HC-DSA Algorithm

In this section, we provide pseudo-code for the new HC-DSA algorithm. Hardcoding an FA-based string processing algorithm using the DSA strategy refers to dynamically allocating blocks of instructions instead of data, as was the case in the TD version. In hardcoding, the blocks of instructions that make up a state are all of the same size. Reference to a given state in the dynamically allocated space is made through its address.

The hardcoded algorithm discussed in this section is referenced by the formalism $\forall s : \mathcal{V}^* \centerdot \rho(\Delta, s) = \rho_{_{CD}}(\varnothing, \Delta_h, 0, D_h, s)$. The strategy argument D_h, informs the implementer whether the algorithm is bounded or not (that is, $0 < D_h < |\mathcal{Q}_h|$ or $D_h = |\mathcal{Q}_h|$). As in the TD-DSA case, the algorithm provided in this section accounts for both bounded and unbounded strategies. Algorithm 4.3.1 provides a high-level specification of the HC-DSA algorithm. The variables and macro-instructions used in the algorithm are as follows[3]:

$n = |\mathcal{Q}_h|$: the number of states;

a: the number of alphabet symbols;

D_h: the threshold for dynamic state allocation (bound);

δ: the transition function;

s: the input string;

j: the index of s indicating the next symbol to be scanned;

[3]Note that the notation used for referencing array entries in the TD-DSA algorithm is retained here, as well as in the algorithm in the next section.

B: the next memory address where the next block of transition instructions relating to a newly visited state is to be copied;

A: the start address in memory where instructions for dynamically allocated states is stored;

$m_{[0:n)}$: an auxiliary array, for indexing into the dynamically allocated instructions as explained below in the discussion of the algorithm's loop invariant;

q: a reference to the row in table δ (or in m) representing the next state to be investigated;

p: a counter of the number of states whose transition instructions currently reside in the dynamically allocated memory;

Z: the amount of space to be reserved for transition instructions relating to each dynamically allocated state;

$search(array, var)$: returns the index, say i, of $array$ such that $array_i = var$.

o: the previously dynamically allocated state that to be replaced when the algorithm uses the bounded DSA;

$wrt(B, \text{“}\delta_{i,[0,a-1)}\text{”})$: writes the hardcoded instructions relating to transitions from state i, (stored in $\delta_{i,[0,a-1)}$) to memory, starting at address B.

$exec@(B)$: executes the instructions starting at address B and returns with a new value for q.

The loop invariant which characterises the loop's body in both bounded and unbounded cases, inv, is the predicate defined below:

$$\begin{aligned} inv \ \triangleq\ & \forall i : [0, |\mathcal{Q}_h|) \centerdot (m_i \neq A-1) \Rightarrow \\ & (m_i \in [A, B) \wedge isDynalloc(m_i)) \end{aligned}$$

Algorithm 4.3.1 (The HC-DSA algorithm)

---

func $hcdsa(\delta, A, Z, D_h, s)$: **boolean**
 if $D_h < n \rightarrow$ { bounded dynamic allocation of states}
  $m_{[0:n-1]} := A - 1;$
  $B, q, j, p := A, 0, 0, 0;$
  $\{ \ inv \ \}$
  do $(j < s.len() \wedge q \geq 0) \rightarrow$
   if $m_q = A - 1 \rightarrow$ { state not dynamically allocated}
    if $p < D_h \rightarrow$
     $malloc(B, Z);$
     $m_q := B;$

```
            wrt(m_q, "δ_{q,[0..a−1]}");
            p, B := p + 1, B + Z
          ▯ p ≥ D_h →
            r := MOD(s, D_h);
            o := search(m, A + Z ∗ r); m_o := A − 1;
            m_q := A + Z ∗ r;
            wrt(m_q, "δ_{q,[0..a−1]}")
          fi
        ▯ m_q ≠ A − 1 → skip { state dynamically allocated}
        fi;
        q, j := exec@(m_q), j + 1
      od
    ▯ D_h = n →  { unbounded dynamic allocation of states}
      m_[0:n−1] := A − 1;
      B, q, j := A, 0, 0, 0;
      do (j < s.len() ∧ q ≥ 0) →
        if m_q = A − 1 →  { state not dynamically allocated}
          malloc(B, Z);
          m_q := B;
          wrt(m_q, "δ_{q,[0..a−1]}");
          B := B + Z
        ▯ m_q ≠ A − 1 → skip { state dynamically allocated}
        fi;
        q, j := exec@(m_q), j + 1
      od
    ▯ D_h > n → skip
    fi;
    return (q ≥ 0)
cnuf
```

It articulates the nature of m, namely that the i^{th} entry of m is either $A-1$, or it lies in the range $[A, B)$. If $m_i = A-1$, then the transition instructions for state i are not in the dynamically allocated memory. Otherwise, m_i is an address in the memory range $[A, B)$ to which the state transition instructions corresponding to row i of δ have been dynamically allocated. The predicate $isDynAlloc(m_i)$ should be regarded as an assertion to this effect.

An alternation statement in the body of the loop in order to decide whether q refers to a dynamically allocated state within $[A, B)$ or a state in δ. This is done by examining the value of m_q. If $m_q = A-1$, then q has to be dynamically allocated and the next value of q is determined after allocation by directly invoking the macro-instruction $exec@(m_q)$. Otherwise, the next value of q is determined directly by pointing at address m_q and executing subsequent instructions from that address.

Again as for the TD-DSA version, the algorithm provided is generic in the sense that it handles both bounded and unbounded DSA. In order to do this, an alternation (i.e. **if-**) statement is used to deal with the maximum number of states that may be dynamically allocated in memory. When $D_h = |\mathcal{Q}_h|$, the algorithm is said to be unbounded, that is instructions that make up all states that fall within the string path are dynamically copied to a new memory location for acceptance testing, provided that they have not yet been visited. For $D_h < |\mathcal{Q}_h|$, the algorithm is bounded; therefore a replacement policy is used to remove a state that has already been processed from the dynamic allocated space, and replace it with the state currently being processed. This situation occurs when the threshold of space to be allocated has been reached. In order to do space replacement, a simple modulus operation is used. However, a more improved technique such as that of the Least Recently Used (LRU) strategy could be envisaged.

Having depicted the hardcoded algorithm based on the dynamic state allocation algorithm, we may also combine both TD and HC DSA-based algorithms to provide an aggregated mixed-mode algorithm. The section below depicts the MM DSA-based algorithm.

4.4 The MM-DSA Algorithm

As already mentioned in this chapter and in the previous chapter, the mixed-mode algorithm refers to a combination of both TD and HC algorithms according to the way the entire transition set of the underlying FA has been split into HC and TD transition sets. In Subsection 4.1.1, various scenarios in the characterization of a mixed-mode DSA recognizer were identified. We focus in this section in the case where both TD and HC are unbounded, corresponding to the following denotational semantics:

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CD}}(\Delta_t, \Delta_h, D_t, D_h, s) \\ D_t = |\mathcal{Q}_t| \wedge D_h = |\mathcal{Q}_h|; |\mathcal{Q}_t| + |\mathcal{Q}_h| = |\mathcal{Q}| \end{cases}$$

Algorithm 4.4.1 depicts in pseudo-code the unbounded version of the mixed-mode algorithm based on the DSA strategy. It is assumed that k is some predetermined value in the interval $[0, |\mathcal{Q}|)$. A transition for a state in the interval $[k, |\mathcal{Q}|)$ of the transition set is determined from hardcode, while the transition for a state in $[0, k-1)$ in the transition table is determined from the transition table. For every iteration of the main loop, a test is made to check whether transition information for the state currently being processed is hardcoded or not. If hardcoded, the same instructions as that of the hardcoded algorithm discussed in the previous section are used to determine the next transition (if it exists). The table-driven instructions are invoked when the state currently being processed falls within the table-driven range. The variables and functions used in the algorithm are the same as those used in the TD and HC DSA algorithms where variables subscripted by t refer to the TD case while those subscripted by h refer to the HC case. Thus, in total, $|\mathcal{Q}| - k$ states are hardcoded and k states are table-driven.

Algorithm 4.4.1 (The unbounded MM-DSA algorithm)

---

func $mmdsa(\delta, A_t, Z_t, A_h, Z_h, k, s)$: **boolean**
    $m_{[0:k-1]}, m_{[k:n-1]} := -1, A_h - 1;$
    $B_t, B_h, q, j, p_t, p_h := A_t, A_h, 0, 0, 0, 0;$
    $\{ \ inv \ \}$
    do $(j < s.len() \wedge q \geq 0) \rightarrow$
        if $q < k \rightarrow$ { states are table-driven }
            if $m_q \leq -1 \rightarrow$
                $m_q := p_t;$
                $d_p := malloc(B_t, Z_t);$
                $d_{p_t,[0..a-1]} := \delta_{q,[0..a-1]};$
                $q, p_t, B_t := d_{p_t,s_j}, p_t + 1, B_t + Z_t$
            $[\!]$ $m_q > -1 \rightarrow$
                $q := d_{m_q,s_j}$
            fi;
            $j := j + 1$
        $[\!]$ $q \geq k \rightarrow$ { states are hardcoded }
            if $m_q = A_h - 1 \rightarrow$ { state not dynamically allocated}
                $m_q := B_h;$
                $malloc(B_h, Z_h);$
                $wrt(m_q, \text{``}\delta_{q,[0..a-1]}\text{''});$
                $B_h := B_h + Z_h$
            $[\!]$ $m_q \neq A_h - 1 \rightarrow$ **skip** { state dynamically allocated}
            fi;
            $q, j := exec@(m_q), j + 1$
        fi
    od;
return $(q \geq 0)$
cnuf

A characterising loop invariant in the main loop of the algorithm, *inv*, is a predicate defined as follows:

$$inv \quad \triangleq \quad \forall i : [0, |\mathcal{Q}|) \begin{cases} i \in [0, k) \wedge (m_i = -1 \vee m_i \in [0, p) \wedge \forall j \centerdot [0, a) \centerdot (\delta_{i,j} = d_{m_{i,j}})) \\ \vee \\ i \in [k, |\mathcal{Q}|) \wedge (m_i \neq A_h - 1 \Rightarrow m_i \in [A_h, B_h) \wedge isDynAlloc(m_i)) \end{cases}$$

The invariant articulates the nature of m, according to its i^{th} entry. In effect, it is a combination of the invariants corresponding to both the TD-DSA and the HC-DSA. For every iteration of the main loop, a state i is either TD ($i \in [0, k)$) or HC ($i \in [k, n)$).

For a TD state, the i^{th} entry of m is either -1 or it has already been dynamically allocated in d for acceptance testing; in this case, access to state information is done via m_i that holds the state's address in d.

In the case of a HC state, the i^{th} entry of m is either $A_h - 1$ or the predicate $isDynAlloc(m_i)$ holds for $m_i \in [A_h, B_h)$. The predicate simply illustrates the fact that the hardcoded instructions that make up the state i have already been copied in the dynamic memory space referenced by the address portion $[A_h, B_h)$ where m_i is found.

The other scenarios of the algorithm can be written in the same way as the unbounded algorithm. All that is required is to use appropriate alternation statements and invoke parts of either TD or HC that handle either unbounded or bounded dynamic allocation of memory.

Having discussed in details all the algorithms derived from the DSA strategy, we provide in the next section an illustrative example of the DSA algorithm applied to TD.

4.5 Illustrative Example

In this section we illustrate how the DSA strategy can be applied to the TD algorithm provided in pseudo-code in Section 4.2 above. In order to do so, consider an automaton $M(\mathcal{V}, \mathcal{Q}, \delta, s_0, \mathcal{F})$ where $s_0 = 0$, $\mathcal{V} = \{a, b, c\}$,
$\mathcal{Q} = \mathcal{F} = \{0, 1, 2, 3, 4, 5, 6\}$, and δ is defined by a two-dimensional array, given by the left-hand table in Table 4.1. This automaton is *partially* represented in Figure 4.1, in that it only shows transitions that will be followed when the string *abcbaabcbaabcba* is being recognized. For this example, it is assumed that the TD-DSA algorithm is unbounded (thus only the second part of the algorithm is considered in our example). The string *abcbaabcbaabcba* will be processed using the TD-DSA algorithm as follows:

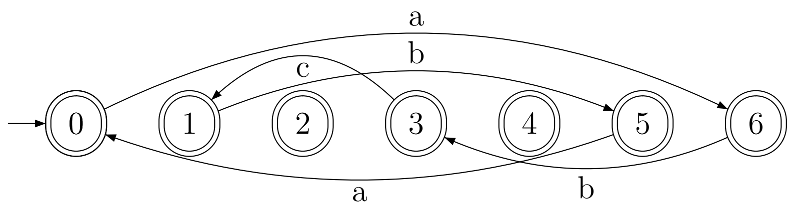

Figure 4.1. A State diagram for testing the string *abcbaabcbaabcba*

Initial phase:

After initialization the following holds:
δ is represented by the first table of Table 4.1. (Thus, $\delta(0, a) = 6$, $\delta(0, b) = 3$, etc.)
$s = abcbaabcbaabcba$
$s.len() = 15$
$m = \{-1, -1, -1, -1, -1, -1, -1\}$.

δ	a	b	c
0	6	3	1
1	2	5	4
2	1	1	2
3	3	2	1
4	4	6	0
5	0	1	3
6	1	3	5

m	
0	0
1	3
2	-1
3	2
4	-1
5	4
6	1

d	a	b	c
0	6	3	1
1	1	3	5
2	3	2	1
3	2	5	4
4	0	1	3

Table 4.1. The arrays δ, m, and d for the DSA Example

$B, q, j, p := A, 0, 0, 0$

The first iteration:

At this stage, all the conditions to enter the loop are satisfied. Therefore, the loop is executed. A test is made on m_q to see whether the state has been created or not. For $q = 0$, the first guard is selected and a new state has to be created in memory. This results in the following:
$q_0 = 0$, that is the old state 0 will occupy the first position in the new memory space. This is shown in the first row of the second table of Table 4.1, which depicts m.

The variable Z represents the memory required to store a state. It depends on the alphabet size (3 for the present example).

The instructions: $d_p = malloc(B, Z)$ and $d_{0,[0..2]} = \delta_{0,[0..2]}$ are then executed to produce $d = \{\{6, 3, 1\}\}$. This corresponds to the first row of the third table of Table 4.1.

Acceptance testing occurs in d, the new value of q is 6, p becomes 1 and j becomes 1.

Later iterations

Suppose the substring *abcba* has already been processed.

In processing the string up to this point, only four of the six automaton states would have been visited. It can easily be seen that the remaining part of the string, that is *abcbaabcba*, involves the traversal of these dynamically allocated states only. In terms of this input string, these four states constitute what we might call a *hot-spot*. In processing the remaining string only states within the *hot-spot* are traversed, thereby hopefully minimizing the need for cache swaps.

In the next section we theoretically evaluate the new DSA algorithm over their preliminary counterparts.

4.6 A theoretical assessment

In this section, we briefly consider the efficiency of the new DSA algorithms in relation to the core algorithms. We restrict the comparison to a theoretical assessment of TD in relation to TD-DSA. An assessment of HC in relation to HC-DSA would be along similar lines.

In cross-comparing TD and TD-DSA algorithms, we rely on the fact that data in cache is processed faster than the data that is in main memory. Furthermore, when data is organized in a contiguous fashion and data items are accessed sequentially, then the number of page swaps is minimized. By contrast, when data is accessed in a disorganized or random fashion, the number of cache swaps is high.

Now, as a matter of fact, ultimately neither the TD nor the TD-DSA algorithms can of themselves directly influence the way in which cache is used. They are "victims", as it were, of the strings that they are required to recognize. The following is a broad classification of the kinds of scenarios that could arise.

1. If an input string continuously drives an algorithm through a relatively small number of states such that these all remain permanently in cache, then both algorithms function optimally. Even if the input string is relatively long, the time taken to process a single symbol is optimized. Of course, in such a case, the DSA algorithm is a poor one, since it needlessly incurs the initial setup cost during the dynamically allocating phase.

2. If the input string drives an algorithm through a somewhat larger number of states, such that cache swaps have to be made, then the question is whether these cache swaps are at a minimum. Again, this behaviour is entirely dependent on the input string.

   (a) Pathological strings could be constructed to induce worst case behaviour for both the TD and DSA algorithms, where as many string symbols as possible induce a transition to a state that is not currently in cache.

   (b) Likewise, well behaved string examples could be constructed where state transitions are nicely ordered to progress from row to row in the original transition table.

   In both these extreme situations, TD would perform better than DSA, since DSA would again incur, without any real gain, the setup cost for dynamic allocation of states.

3. Under the previous scenario (i.e. where a large number of states are traversed), the DSA algorithm could potentially acquire an advantage over the TD algorithm if the input string exhibited the following characteristics:

   (a) the string tended to repeatedly exercise the same subset of states; where

   (b) these states were fairly widely distributed over the transition table rows, thus causing many cache misses under TD; but

(c) where the states were contiguously placed in d because the order of their initial usage reflected their later usage and order.

It is easy to see that under these circumstances, the hot-spot of the DSA algorithm would repeatedly be exercised in a way that minimized cache swaps, while the TD algorithm would incur a high number of cache swaps.

The claim made in Point 3 is rather general. It does not attempt to quantify how many dynamic allocations should take place, how many times the hot-spot should be exercised, how long the input string should be, how rows in the transition table should be ordered, etc. Clearly all of these factors could influence the extent to which DSA improves over TD. Indeed, at this point, it is not even clear whether, under practical conditions, the cost of dynamic state allocation is ever really likely to pay off. In Part III, experiments are described that offer some insights into these matters.

4.7 Summary of the Chapter

In this chapter, we have introduced a new strategy referred to as the DSA strategy for implementing FA-based string processors. A formal characterization is provided that expresses how the strategy may be combined with TD , HC or MM to yield various FA-based recognizer implementations. The reason for investigating such an implementation strategy was suggested by the nature of cache memory at hardware level. An example illustrated that time efficiency may be gained when using the DSA algorithm provided that the overhead caused by the dynamic allocation operation is eventually offset by gains caused by efficient use of cache. In particular, the efficiency are likely to be more significant when processing strings whose string-path tend to visit repeatedly a limited number of states. An empirical investigation of the performance of some of the algorithms provided in this chapter is reserved for Part III. The next chapter discusses yet another implementation strategy referred to the pre-ordering of states.

CHAPTER 5

STATES PRE-ORDERING

In this chapter, we discuss yet another implementation strategy of FA-based recognizer algorithms. An algorithm that relies on this strategy will be referred to as a States pre-Ordering (SpO) algorithm. Such an algorithm is based on the premise that when processing an FA that has a large number of states, then the order in which its states are organized in memory plays an important role on the efficiency of the recognizer, especially when processing a long string whose string path involves only a limited number of states. The chapter starts with a formal description of FA-based string processing based on the SpO strategy. Then follows a discussion on the implementation of the core FA-based algorithms using the SpO strategy. Also provided in this chapter is a theoretical assessment of the suggested algorithms compared to their core counterparts.

5.1 The SpO-based Characterization

When carrying out FA-based string recognition, it may happen that the string being tested for acceptance frequently visits only a small part of the whole transition graph. It may even be the case that the number of visited states is well below the overall number of states that make up the automaton. It seems sensible to account for such situations by providing a mechanism for reorganizing the transition graph so that frequently accessed states are grouped together, thus optimizing the performance of the recognizer. In effect, by putting frequently visited states next to each other, the number of cache misses is reduced, resulting in better cache utilization, and hence improved efficiency of the recognizer. The 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 (online). 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.

In order to reference the strategy in terms of the functional description given in Chapter 3, a function argument to represent the SpO strategy is introduced. As in the case of the strategy described in the previous chapter, the SpO argument indicates whether the strategy is adopted or not. If the strategy has been adopted, then the algorithm requires a preprocessing function that reorders the automaton's state before acceptance testing.

Since the general formalism of a string recognizer contains both the TD and the HC algorithms, separately, we introduce two arguments: P_t and P_h. The first is a boolean that indicates whether the TD part of the algorithm requires pre-ordering of states or not. Likewise, the second parameter is a boolean indicating whether the hardcoded part of the algorithm is based on state pre-ordering or not. Thus, when both arguments evaluate to *true* (T), then the recognizer corresponds to an MM algorithm following the SpO strategy, provided that the TD and the HC transition sets constitute a partition of the automaton's transition set.

We can now define a function which accounts for a possible SpO strategy implementation for FA-based string recognition. We call this function ρ_{CP}, the $_C$ subscript indicating that it can express any core algorithms, and the $_P$ subscript indicating that it also accommodates the S$\underline{\text{p}}$O strategy. The function is thus defined as follows:

$$\rho_{CP} : \mathcal{T} \times \mathcal{T} \times \mathbb{B} \times \mathbb{B} \times \mathcal{V}^* \nrightarrow \mathbb{B} \tag{5.1}$$

such that

$$\text{if } \begin{cases} (\Delta_t \cup \Delta_h = \Delta) \wedge (\Delta_t \cap \Delta_h = \varnothing) \\ (P_t \in \mathbb{B} \wedge P_h \in \mathbb{B}) \end{cases} \quad \text{then } \rho_{CP}(\Delta_t, \Delta_h P_t, P_h, s) = \rho(\Delta, s)$$

With the above formalism, some properties could be expressed in order to avoid ambiguity in the usage of the SpO strategy variables. The next section depicts the properties of the SpO strategy.

5.2 Properties of the SpO strategies

The following highlights a number of properties to be preserved when referring to the SpO strategy in terms of the foregoing functional notation. In fact, these refer to permissible combinations of parameter values in ρ_{CP}. In certain cases, it will be convenient to introduce specific abbreviations/terminology to reference particular combinations.

1. Each strategy depends on the cardinality of its corresponding transition set. That is:

$$\begin{cases} |\Delta_h| = 0 \Rightarrow P_h = \mathtt{F}(false) \\ \text{and} \\ |\Delta_t| = 0 \Rightarrow P_t = \mathtt{F} \end{cases}$$

2. When $P_t = \mathtt{T}$ (or $P_h = \mathtt{T}$), an auxiliary array p^t of size $|\mathcal{Q}_t|$ (or p^h of size $|\mathcal{Q}_t|$) will be assumed to hold the new position of the states of the FA.

3. The core TD algorithm can be formally expressed in terms of this new characterization by regarding the HC transition set as empty and its associated SpO strategy as $false$. The following relationship therefore holds:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_C}(\Delta_t, \varnothing, s) \equiv \rho_{_{CP}}(\Delta_t, \varnothing, \mathtt{F}, \mathtt{F}, s).$$

4. The core HC algorithm could be formally expressed in terms of this new characterization by regarding the TD transition set as empty and its associated SpO strategy as $false$. The following relationship therefore holds:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_C}(\varnothing, \Delta_h, s) \equiv \rho_{_{CP}}(\varnothing, \Delta_h, \mathtt{F}, \mathtt{F}, s).$$

5. When both TD and HC transition sets are non-empty, with their associated SpO arguments evaluated to $false$, then the formalism corresponds to that of the core mixed-mode algorithm. Therefore, following relationship holds:

$$\begin{cases} \forall s : \mathcal{V}^* \centerdot \rho_{_C}(\Delta_t, \Delta_h, s) \equiv \rho_{_{CP}}(\Delta_t, \Delta_h, \mathtt{F}, \mathtt{F}, s) \\ \text{with} \\ |\mathcal{Q}_t| + |\mathcal{Q}_h| = |\mathcal{Q}|;\ \Delta_t \cup \Delta_h = \Delta;\ \Delta_t \cap \Delta_h = \Delta \end{cases}$$

6. The TD-SpO algorithm refers to the scenario where the table-driven transition set is non-empty with an associated SpO argument evaluated to $true$, and the hardcoded transition set is empty. The following relationship holds for the TD-SpO algorithm:

$$\forall s : \mathcal{V}^* \centerdot \rho(\Delta, s) = \rho_{_{CP}}(\Delta_t, \varnothing, \mathtt{T}, \mathtt{F}, s)$$

7. Similarly, the HC-SpO algorithm refers to the scenario where the hardcoded transition set is non-empty with a corresponding SpO argument that evaluates to $true$, and the TD transition set is empty. The following relationship therefore holds for the HC-SpO algorithm:

$$\forall s : \mathcal{V}^* \centerdot \rho(\Delta, s) = \rho_{_{CP}}(\varnothing, \Delta_h, \mathtt{F}, \mathtt{T}, s)$$

8. When both TD and HC transition sets are non-empty, with at least one associated SpO argument that evaluates to *true*, the resulting formalism is that of the mixed-mode SpO algorithm. Therefore, the following relationships hold for the MM-SpO algorithm:

$$\forall s : \mathcal{V}^* \centerdot$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CP}}(\Delta_t, \Delta_h, P_t, \mathtt{F}, s) \\ P_t \neq \mathtt{F} \end{cases} \tag{5.2}$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CP}}(\Delta_t, \Delta_h, \mathtt{F}, P_h, s) \\ P_h \neq \mathtt{F} \end{cases} \tag{5.3}$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CD}}(\Delta_t, \Delta_h, P_t, P_h, s) \\ P_t \neq \mathtt{F} \land P_h \neq \mathtt{F} \end{cases} \tag{5.4}$$

9. The MM-SpO algorithm is said to be *weak on HC (or strong on TD)* if the relationship 5.2 of Property 8 holds.

10. The MM-SpO algorithm is said to be *weak on TD (or strong on HC)* if the relationship 5.3 of Property 8 holds.

11. The MM-SpO algorithm is said to be *complete* if the relationship 5.4 of Property 8 holds.

These various SpO algorithms, based on TD, HC, or MM, can be implemented and cross-compared with their core counterparts, to assess their utility for FA-based string recognition. In the sections to follow in this chapter, we provide the pseudo-code of some of the SpO algorithms. Part III will be devoted to an empirical analysis of their performance. A theoretical assessment of the SpO algorithms compared to their core counterparts is also discussed towards the end of this chapter. The next section discusses the TD-SpO algorithm.

5.3 The TD-SpO algorithm

The TD-SpO algorithm formally characterized in Property 6 refers to a table-driven implementation of FA-based recognizer that relies on the state pre-ordering strategy. As mentioned earlier, the state pre-ordering is based on the premise that the implementer is aiming to exploit the order in which states are visited at runtime so as to improve the overall performance of the recognizer. If the implementer is provided with an array, p^t, whose entries hold the order in which the states are expected to be visited, the algorithm becomes straightforward since it would consist of: a preprocessing phase that reorders the automaton's states; and a processing phase that performs the actual acceptance testing.

However, in some circumstances, there may be no information available of the order in which states are accessed. In that case, the SpO strategy cannot be applied until the information has been obtained. Clearly, one would expect that the information would have to be determined from the history of FA-based string recognition in a particular context. One might imagine the deployment of computational intelligence techniques such as artificial neural networks, genetic algorithms, etc to "learn" the most likely patterns of state utilization from past data. However, details of how such information might be found is beyond the scope of this present study.

The SpO algorithm consists of a preprocessing phase whereby, the position of each state of the automaton is reordered according to the entries available in the auxiliary array p^t provided as input. Then follows the processing phase where acceptance testing takes place.

Recall that the states are assumed to be named according to their row number in the original transition table: the transition information for state 0 is in row 0; for state 1 it is in row 1, etc. During the pre-processing phase, state information is swapped from its original position in the transition table to its new position. However, states are not renamed during this process. Thus, for example, after pre-ordering, transition information for, say, state 5 may be in the 2^{nd} row of the transition table. This will be the case if $p^t_5 = 2$. The next transition may be to, say state 7. To determine the transition table row for this state, we need to look up p^t_7, etc.

Thus, acceptance testing is similar to that of the core table-driven algorithm, access to state information in the transition table has to be obtained indirectly via p^t. This additional level of indirection introduces a slight computational penalty in relation to the core TD algorithm, but the hope is that this will be offset by gains in the more efficient utilisation of cache.

Algorithm 5.3.1 depicts the pseudocode for the SpO-TD algorithm. The variables and function used in the algorithm are as follows:

n: the number of states ;

a: the number of alphabet symbols ;

$\delta : \mathcal{Q} \times \mathcal{V} \to \mathcal{Q}$: the transition function;

s: the input string;

j: an index of s indicating the next symbol s_j to be scanned;

i: a control variable (integer) used at preprocessing phase;

$p_{[0:n)}$: the array of the new position of each state;

$c_{[0:n)}$: an auxiliary array used to keep track of the row in which a state's transition information has been placed during preprocessing, as discussed below;

$swap(x, y)$: a function used to interchange the contents of the variables x and y;

Algorithm 5.3.1 (The TD-SpO algorithm)

---

func $tdspo(\delta, p, s)$: **boolean**
  $c_{[0:n-1]} := [0..n-1]$;
  {preprocessing phase}
  $i := 0$;
  do $(i < n) \rightarrow$
    if $(p_i \neq c_i) \rightarrow$
      $swap(\delta(p_i, [0..a-1]), \delta(c_i, [0..a-1]))$;
      $swap(c_i, c_{p_i})$
    $[\!]$ $p_i = c_i \rightarrow$ **skip**
    fi;
    $i := i + 1$
  od;
  {processing phase}
  $q, j := 0, 0$;
  do $(j < s.len()) \wedge (q \geq 0) \rightarrow$
    $q, j := \delta(p_q, s_j), j + 1$
  od;
  return $(q \geq 0)$
cnuf

---

initially the array's entries (c_i $\forall i \in [0..n)$) contain the natural order of each state; The algorithm is executed through major phases:

1. *Preprocessing*: Input to this phase is the auxiliary array p whose value at the j^{th} index, p_j, represents the *required* row-position of state j in the re-ordered transition table. A loop traverses through the array p, accessing and manipulating an auxiliary array, c. The latter array is maintained in such a way that, for $j = 0, \cdots n-1$, c_j represents the *current* row-position of state j in the transition table. Thus, before the loop, c_j is initialized to j $(0 \leq j < n)$; that is, $c_i = i$. In the i^{th} iteration of the loop, an equality test is made on the entries c_i and p_i. If the test evaluates to *true*, it means that the state i is already at its required position and nothing is done. However if the test evaluates to $false$, then we swap the transition information in rows c_i and p_i. State i will then be at its desired row-position in the transition table, namely p_i. However, at that point, c_i references a row that now contains transition information for state p_i, and c_{p_i} references a row that now contains transition information for state c_i. By swapping the values of c_i and c_{p_j}, we ensure that c retains its invariant property mentioned above, namely that for $j = 0, \cdots n-1$, c_j represents the *current* row-position of state j in the transition table. At the end of the preprocessing operation, all states are at their desired position and can be accessed indirectly through p.

2. *Acceptance testing*: It is similar to the core TD algorithm. However, access to a state currently being processed is made via the array p. For example, if i is the state currently being processed, and s_j the index of the next symbol to be tested for acceptance, $\delta(p_i, s_j)$ gives the actual value to be transited to instead of $\delta(i, s_j)$ as was the case for the core TD algorithm.

The TD-SpO based algorithm takes both the transition table as well as the auxiliary array of positions to be allocated as input. Its memory requirements are principally determined by the size of these inputs. In terms of its time efficiency, although the pre-ordering operation is expensive especially for large automata, once the states have been ordered according to the implementer's need, optimum cache usage and hence performance enhancement may be expected. In the next section we discuss the HC-SpO algorithm.

5.4 The HC-SpO algorithm

The hardcoded SpO algorithm was formally characterized in Property 7. The algorithm is based on the premise that, given some knowledge on the order in which the states are visited during acceptance testing, a preprocessing function that accordingly reorders the states may be used to generate the hardcoded directly executable code before acceptance testing takes place. Thus, an array p^h that holds the new states's positions is provided as input for the preprocessing operation. Once hardcoded instructions are generated in memory, acceptance testing simply occurs on a "natural" way, as was the case for the core HC algorithm. As discussed in Section 3.2 of Chapter 3, the HC-SpO algorithm requires a preprocessing operation enabling it to generate the hardcoded instructions based on the order in which states are expected to be accessed at runtime. For the present, we assume that a variable *top*, points to the address in memory where the first instruction of the HC-SpO algorithm is to be written. Then, after generating the hardcoded instructions based on the SpO strategy, we would redirect the program counter to *top* for acceptance testing.

Algorithm 5.4.1 called *hcspog*[1] below gives pseudo-code for generating such a hardcoded recognizer, and then executing it in respect of an input string, s. As input, the generator program takes the transition function δ; the starting address of the generated instructions *top*; the number of states of the FA, $|Q| = n$; and the number of alphabet symbols $|\mathcal{V}| = a$. It also takes in as input, the array of the states positions $p_{[0:n)}$ which will be used to set up the hardcoded states in memory; the input string s to be used for acceptance testing is also provided as parameter to the algorithm.

As in the case of the TD-SpO algorithm, the best estimate of optimal state positioning is available in advance, and reflected in provided $p_{[0:n)}$. Although variations of the algorithm could be envisaged whereby the optimal contents of array p is incrementally learned. However, strategies for such learning —though clearly of potential

[1]The reader should refer to Section 3.2 of Chapter 3 for more information on the functions called in this algorithm.

importance— are beyond the scope of this thesis, and are considered outside the domain of concern in regard to the SpO strategy itself.

Algorithm 5.4.1 (Generation and direct execution of a HC-SpO string recognizer)

```
func hcspog(δ, top, n, a, p, s) : boolean
   B := top;
   gen("q, j := 0, 0;", B);
   gen("do (j < s.len()) ∧ (q ≥ 0) → ", B);
   i := 0;
   do i < n →
      if i = 0 → gen("if q = p_i → ", B)
      [] i ≠ 0 → gen("[] q = ", B); gen(p_i, B); gen(" → ", B)
      fi;
      k := 0;
      do k < a →
         if k = 0 → gen("if s_j = c_0 → ", B)
         [] k ≠ 0 → gen(" [] s_j = ", B); gen(c_k, B); gen(" → ", B)
         fi;
         gen("q, j := ", B); gen(δ(p_i, c_k), B); gen(", j + 1", B);
         k := k + 1
      od;
      gen("fi", B);
      i := i + 1
   od;
   gen("fi", B);
   gen("od;", B);
   gen("Return (q ≥ 0)", B);
   exec@(top, s)
cnuf
```

The various functions and operations referred to in the algorithm have already been described in Section 3.2 of Chapter 3. It is worth mentioning that the two algorithms are different in terms of the code generated, since the order in which the conditional statement are written is now dictated by the array p instead of the natural order in which states appear in the transition table. Thus, the k^{th} hardcoded state instruction to be written is that of state p_k $(k \in [0..n))$, instead of that of state k as was the case for the core hardcoded algorithm.

An example of the code generated by the function *hcspog*() is depicted in Algorithm 5.4.2. We use the example provided in Subsection 3.2.0.1 of Chapter 3 for illustration[2]. The generator is provided with an array of positions $p = \{0, 4, 2, 5, 1, 3\}$. In contrast to the TD-SPO algorithm where access to state information was made indirectly via the array p, the hardcoded version performs acceptance testing without requiring such indirect accesses. The pre-ordering has organized the state code contiguously, thus minimizing the probability of cache misses during acceptance testing.

It may therefore be expected that the HC-SPO algorithm will outperform its core HC counterpart. Given that in previous experiments it was found that HC outperformed TD up to a certain threshold of number of states (discussed in [Nga03]), it may also be expected that HC-SPO will improve this threshold of efficiency in relation to the core TD algorithm. Of course these expectations assume that the string paths of the strings being tested for acceptance closely follow the order in which the HC states have been encoded.

Algorithm 5.4.2 (Pseudocode HC-SPO recognizer for a given transition function)

```
func hcspo(s) : boolean
   q, j := 0, 0;
   do (j < s.len) ∧ (q ≥ 0) →
       if q = 0 →
          if s_j = 'd' →
             q, j := 5, j + 1
          [] s_j = 'i' →
             q, j := 5, j + 1
          [] s_j = 'o' →
             q, j := 5, j + 1
          [] s_j = 'v' →
             q, j := 1, j + 1
          fi
       [] q = 4 →
          q, j := 5, j + 1
       [] q = 2 →
          if s_j = 'd' ∨ s_j = 'o' ∨ s_j = 'v' →
             q, j := 5, j + 1
          [] s_j = 'i' →
             q, j := 3, q + 1
          fi
       [] q = 5 →
          q, j := −1, j + 1
       [] q = 1 →
          if s_j = 'd' ∨ s_j = 'i' ∨ s_j = 'v' →
             q, s_j := 5, j + 1
          [] s_j = 'o' →
```

[2] The reader should notice that in order to save space, some conditional branches have been combined together using the boolean OR, since the next state to be transited to are identical.

```
            q, j := 2, j + 1
        fi
        q, j := 5, j + 1
      ⫿ q = 3 →
        if s_j = 'i' ∨ s_j = 'o' ∨ s_j = 'v' →
            q, j := 5, j + 1
        ⫿ s_j = 'd' →
            q, j := 4, j + 1
        fi
      fi
   od;
   return (q ≥ 0)
cnuf
```

5.5 The MM-SpO algorithm

As shown in Property 8, there are three variations of the MM-SpO algorithm. In this section, we only discuss the case that seems to be more general, that is the so-called *complete* MM-SpO algorithm. The MM-SpO algorithm is said to be *complete* if $P_t = P_h = \mathtt{T}$. Thus, the following relationship holds:

$$\rho(\Delta, s) = \rho_{_{CD}}(\Delta_t, \Delta_h, \mathtt{T}, \mathtt{T}, s)$$

In mixed-mode, we assume that the first $m = |\mathcal{Q}_t|$ ($0 < m < n$, n being the FAs number of states) states are processed from the transition table, and the next $n - m = |\mathcal{Q}_h|$ states are hardcoded. As for the previous algorithms, the MM-SpO algorithm consists of a pre-processing phase and a processing phase and an array $p_{[0:n)}$ holding the new positions of the states is provided as input.

We assume that portion $[0, m)$ of the array holds the new position of table-driven states, and the portion $[m, n)$ holds the new position of the hardcoded states. The preprocessing phase would then consist of reordering the table-driven portion of the table-driven states, and generating hardcoded instructions according to the order in which the hardcoded states ought to be visited. During acceptance testing, reference to a table-driven state is handled by a driver piece of code that accesses state's information indirectly through p; and reference to a hardcoded state is handled by directly executable instructions without using p's entries, since the states have been reordered accordingly.

The pseudocode of Algorithm 5.5.1 named *mmspo*() takes as parameters, the automaton's transition function δ, the array of state positions p, the threshold of hardcoded/table-driven states m, and the input string s. The following variables and functions are used in the algorithm:

top: the start address in memory where the hardcoded part of the MM-SpO algorithm is to be executed;

$reorder(\delta, m, p)$: Assigns the first m states of the automaton to their final position as referenced by the first m entries of the array p.

$genhc(\delta, top, n-m, a, h, s)$: generates directly executable hardcoded instructions that make up each state of the hardcoded part of the MM-SpO algorithm. The order in which the states are generated depends on the entries in the array of positions $h_{[0:n-m-1]}$.

$\delta(i, [0..a])$: refers to all the transitions of the state i as determined by the transition function of the automaton.

$execHC(q, s_j)$ executes this hardcode of the mixed-mode recognizer. It also updates the variable q, that refers to the next state to be transited to.

In the mixed-mode-SpO algorithm, the preprocessing phase handles the pre-ordering of both hardcoded and table-driven states. For every iteration of the loop, we then test whether the control variable i is less than m or not. If i is less than m, then the preprocessing reorders the table-driven states, otherwise hardcoded states are reordered.

After all states have been reordered the hardcoded portion of the algorithm is generated and processing can then take place as shown in the algorithm.

Algorithm 5.5.1 (The *complete* mixed-mode-SpO algorithm)

---

func $mmspog(\delta, n, p, m, top, s)$: **boolean**
{ preprocessing phase}
  $reorder(\delta, m, p)$;
  $h_{[0:n-m-1]} := p_{[m:n-1]}$;
  $genhc(\delta, top, n-m, a, h, s)$;
  {processing phase}
  $q, j :=, 0, 0$;
  do $(j < s.len() \wedge q \geq 0) \rightarrow$
    if $q < m \rightarrow$ {the state is table-driven}
      $q, j := \delta(p_q, s_j), j+1$
    $[\!]$ $q \geq m \rightarrow$ {the state is hardcoded}
      $q, j := execHC(q, s_j), j+1$
    fi
  od;
  return $(q \geq 0)$
cnuf

The code depicted in Algorithm 5.5.2 gives the notional idea of how the mixed-mode implementation of the running example would evolve into code. During acceptance testing, part of the transition matrix represented by a table is accessed by a driver function whereas the hardcoded part is directly executed. In the algorithm, the first 3 states are table-driven and the remaining states are hardcoded.

Algorithm 5.5.2 (An applied MM-SPO recognizer)

```
func mmspo(δ, 3, p, s) : boolean
   q, j := 0, 0;
   do (j < s.len) ∧ (q ≥ 0) →
       if q < 3 → q, j := δ(p_q, s_j), j + 1
       [] q ≥ 3 →
          if q = 4 → q, j := 5, j + 1
          [] q = 5 → q, j := −1, j + 1
          [] q = 3 →
             if s_j = 'i' ∨ s_j = 'o' ∨ s_j = 'v' → q, j := 5, j + 1
             [] s_j = 'd' → q, j := 4, j + 1
             fi
          fi
       fi
   od;
   return (q ≥ 0)
cnuf
```

Having provided some of the variations of string processing algorithms based on the SpO strategy, we briefly discuss in the next section, theoretically, how the algorithms will perform in relation to their core counterparts.

5.6 Theoretical Assessment

In this section, we briefly discuss the advantage and disadvantage of the suggested SpO algorithms in general over their core FA-based string processors. It is common knowledge that the complexity of a string recognizer is linear to the string length. Whether we consider SpO algorithms or preliminary FA-based string processing algorithms, their complexity would remain linear in the length of the string being tested for acceptance. The main objective in the investigation of new algorithms for string processing is to take advantage of hardware capabilities. In effect, an important aspect of hardware that hampers the efficiency of algorithms is the cache. Due to cache memory reliance on locality of reference discussed in Chapter 2, the better data are organized in memory the better the latency of the algorithm.

For the core FA-based string processing algorithms, the transition function is not modified and is loaded in memory exactly as it was first designed. Such a form of data organization in memory means that entries of the transition table (whether hardcoded or not) are structured in a random fashion. If the string being processed happened to access this data contiguously according to each symbol that forms part of the string, there would be a low probability of cache misses and hence better processing speed. However if data are accessed on a random fashion the core algorithms would be subject to high probability of cache misses, resulting to poor performance.

The major drawback of the SpO algorithms is the cost of the preprocessing phase. As seen in the above algorithms the preprocessing requires $O(n \times a)$ operations (where n is the automaton's number of states and a the alphabet size). Therefore if preprocessing is to take place for each new input string, then the core algorithms would clearly outperform their SpO counterparts. However, it is envisaged that the SpO strategy will primarily be deployed in cases where a large number of similar strings are to be processed. In such cases, the pre-ordering costs are incurred only once, with the hoped-for benefit that the subsequent string processing will be more efficient.

The SpO strategy should be considered when a large number of strings are to be processed, and the following applies: *The transition table is large, and only a limited number of states —non-uniformly distributed within the table— are repeatedly visited.* The implementer should then attempt to reorganize the frequently visited states within the same memory range so as to maximize cache benefits. The SpO strategy may thus be considered when the implementer has some prior knowledge of the order in which states are visited. Although the preprocessing phase is costly, subsequent acceptance testing will hopefully amortize the costs of the reordering operation, yielding better eventual performance.

Furthermore, if a string to be processed is very long, the cache advantages of having frequently visited states spatially localized within the same memory space outweigh the cost of preprocessing, even for a single run.

5.7 Summary of the Chapter

In this chapter, we have introduced yet another strategy for implementing FA-based string recognizers. For consistency with the previous chapter, we used the variables P_t and P_h to represent the new strategies referred to as the SpO strategies. As a result to this, new TD, HC and MM algorithms based on the SpO strategies where suggested, and it was shown that these algorithms are likely to outperform their preliminary counterparts when processing large FAs, provided that the string being processed is relatively long and the number of frequently visited states is relatively small compared to the overall automaton size.

Implementing the SpO strategy is based on the premise that the FA implementer has some prior knowledge of the order in which the states are likely to be accessed. However, it is possible to provide a variation whereby the algorithm starts with the FA's original order and frequently adapts itself according to the strings processed thus far. In both cases, the core aspect of the algorithm would remain the same with more or less the same expectations in terms of performance. However, such arguments are matter of further investigations since "online ordering" do not assume prior knowledge of the order in which states would be visited.

In Part III a practical experiment is conducted in order to quantify the extent to which our theoretical assessment holds in practice. In the next chapter, we discuss another variation of FA-based string processing strategy, referred to as the allocated virtual caching strategy.

CHAPTER 6

ALLOCATED VIRTUAL CACHING

This chapter discusses an implementation strategy of FA-based string processing algorithms referred to as the Allocated Virtual Caching (AVC) strategy. The strategy is based on the designating (or allocating) as a "virtual cache", a block of memory within the portion of memory occupied by the automaton's transition table. The objective is to ensure that state transition data within the virtual cache is maximally used during input string acceptance testing. Because the size of the virtual cache is limited, it may be necessary to remove transition data for some states from the cache from time to time, and to replace this data with transition data of more frequently used states. This replacement policy comes into play when the cache is full, and could rely on policies such as a direct mapping policy, or a LRU (Least Recently Used) policy. The AVC strategy seeks to maximize locality of reference, and thus enhance performance of a string recognizer by attempting to ensure that frequently visited states are always available in the cache.

The chapter starts off by formally characterizing FA-based processors in terms of parameters associated with the AVC strategy, and indicating the range of values which those parameters can assume. Then follows a discussion on various algorithms based on the AVC strategy. Towards the end of the chapter, an illustrative example of AVC-based algorithms is provided, as well as a brief theoretical assessment of the new algorithms compared to their core counterparts.

6.1 The AVC-based Characterization

The implementation of FA-based string processing algorithms using the allocated virtual caching 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. Removing a state from the cache makes cache space available, so that the new state's information can be placed in the empty cache space. Of course, 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* [PH05] 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 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 whose information is constantly present within the virtual cache. If, in addition, the string path visits states whose information is *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.

In order to formally described the AVC strategy on a refined form of the functional description given in Chapter 3, we need to dedicate two arguments to the AVC strategy, such that one of the argument is related to the TD algorithm and the other to the HC algorithm. The datatype used to describe the AVC strategy is a natural number. It represents the maximum number of states that are to be used as virtual cache in memory. In effect, if a parameter is allocated a non-zero value (say V), then the first V states occupy the virtual cache portion that holds state information for acceptance testing. Therefore, when reference is made to a state that is not present within the virtual cache, a replacement strategy is used to swap the state in the cache. Otherwise, acceptance testing occurs on that state since its information is available in the cache. A zero value assigned to the strategy's argument simply means that the AVC strategy has not been applied to this particular algorithm.

The AVC strategy is similar to the DSA strategy discussed in Chapter 4 in the sense that it also relies on states allocation in cache. However, for the DSA strategy, the states are allocated dynamically in an initialized "free" portion of the memory, whereas the AVC strategy takes advantage of the block of the memory initially occupied by states, and makes it behave as a kind of virtual cache memory. One of the direct advantage of the AVC algorithms over the DSA algorithms would arise when processing a string that visits on a contiguous fashion, states that are already available in the virtual cache. In this case, there is no overhead caused by states replacement —hence processing at optimum, whilst the DSA algorithm would always require dynamic allocation of states in memory. As for the DSA strategy , the two arguments used for the AVC strategy, V_t and V_h, are associated with the TD and HC algorithms respectively. Unlike the DSA strategy, the AVC strategy must always be bounded since if unbounded, the virtual cache would not be considered as such, in that, states replacement would not happen at runtime; in this case the AVC strategy would be similar to a version of SpO algorithm (not covered in this thesis) whereby, states are reordered as they are being visited. Therefore, each AVC strategy should always remain strictly less than its associated number of states.

We can now define a function which accounts for the possible use of the AVC strategy when implementing FA-based string recognition. Call this function $\rho_{_{CV}}$, the $_C$ subscript indicating that it can express any of the three core algorithms, and the $_V$ subscript indicating that it also accommodates the AVC strategy. The function is thus defined as follows:

$$\rho_{_{CV}} : \mathcal{T} \times \mathcal{T} \times \mathbb{N} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B} \tag{6.1}$$

such that

$$\text{if} \begin{cases} (\Delta_t \cup \Delta_h = \Delta) \wedge (\Delta_t \cap \Delta_h = \varnothing) \\ (0 \leq V_t < |\mathcal{Q}_t|) \wedge (0 \leq V_h < |\mathcal{Q}_h|) \end{cases} \text{then } \rho_{_{CV}}(\Delta_t, \Delta_h V_t, V_h, s) = \rho(\Delta, s)$$

The conditions under which the above formalism should be used are presented as properties in the next section.

6.2 Properties of the AVC strategies

The following properties characterise the AVC strategy:

1. The AVC strategy is always *bounded*, that is, $V_t < |\mathcal{Q}_t|$ and $V_h < |\mathcal{Q}_h|$. As mentioned in the previous section, if it had been permissible that $V_t = |\mathcal{Q}_t|$ or $V_h = |\mathcal{Q}_h|$, then state swapping in the respective TD or HC part of the algorithm would only take place until the cache line had grown to its maximum value. This scenario is reminiscent of an SpO algorithm, where a preordering is first learnt, and then later applied.

2. Of course, the deployment of the strategy depends on the cardinality of its associated transition set. That is,

$$\begin{cases} |\mathcal{Q}_h| = 0 \Rightarrow V_h = 0 \\ \text{and} \\ |\mathcal{Q}_t| = 0 \Rightarrow V_t = 0 \end{cases}$$

3. The core TD algorithm can be formally expressed in terms of this new characterization by regarding the HC transition set as empty and its corresponding AVC strategy as zero. The following relationship therefore holds:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_C}(\Delta_t, \varnothing, s) \equiv \rho_{_{CV}}(\Delta_t, \varnothing, 0, 0, s).$$

4. Similarly, the core HC algorithm can be formally expressed in terms of this new characterization by regarding the TD transition set as empty and its associated AVC strategy as zero. Again, the following relationship holds:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_C}(\varnothing, \Delta_h, s) \equiv \rho_{_{CV}}(\varnothing, \Delta_h, 0, 0, s).$$

5. If both TD and HC transition sets are non-empty, and their associated AVC strategy parameters are zero, then the resulting formalism represents the core mixed-mode algorithm. The following relationship therefore holds:

$$\begin{cases} \forall s : \mathcal{V}^* \centerdot \rho_{_C}(\Delta_t, \Delta_h, s) \equiv \rho_{_{CV}}(\Delta_t, \Delta_h, 0, 0, s) \\ \text{with} \\ |\mathcal{Q}_t| + |\mathcal{Q}_h| = |\mathcal{Q}|;\ \Delta_t \cup \Delta_h = \Delta;\ \Delta_t \cap \Delta_h = \Delta \end{cases}$$

6. When the table-driven transition set is non-zero with a non-zero associated AVC strategy, and the hardcoded transition set is empty, the characterization is that of the TD-AVC algorithm. Thus the TD-AVC algorithm is characterised by the following relationship:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_{CV}}(\Delta_t, \varnothing, V_t, 0, s) = \rho(\Delta, s), \text{ with } V_t \neq 0.$$

7. When the hardcoded transition set is non-zero with a non-zero associated AVC strategy, and the table-driven transition set is empty, the characterization is that of the HC-AVC algorithm. Therefore, the following relationship holds for the formalism of the HC-AVC algorithm:

$$\forall s : \mathcal{V}^* \centerdot \rho_{_{CV}}(\varnothing, \Delta_h, 0, V_h, s) = \rho(\Delta, s), \text{ with } V_h \neq 0.$$

8. When both TD and HC transition sets are non-empty, with at least one non-zero associated AVC strategy, then the resulting formalism is that of the mixed-mode AVC (MM-AVC) algorithm. The following characterizations point to different variants of the MM-AVC algorithm:

$$\forall s : \mathcal{V}^* \centerdot$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CV}}(\Delta_t, \Delta_h, V_t, 0, s) \\ 0 < V_t < \mathcal{Q}_t \end{cases} \tag{6.2}$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CV}}(\Delta_t, \Delta_h, 0, V_h, s) \\ 0 < V_h < \mathcal{Q}_h \end{cases} \tag{6.3}$$

$$\begin{cases} \rho(\Delta, s) = \rho_{_{CV}}(\Delta_t, \Delta_h, V_t, V_h, s) \\ 0 < V_t < \mathcal{Q}_t \wedge 0 < V_h < \mathcal{Q}_h \end{cases} \tag{6.4}$$

9. The MM-AVC algorithm is said to be *weak on HC (or strong on TD)* if Equation 6.2 of the Property 8 holds.

10. The MM-AVC algorithm is said to be *weak on TD (or strong on HC)* if Equation 6.3 of the property 8 holds.

11. The MM-AVC algorithm is said to be *complete* if Equation 6.4 of the property 8 holds.

Having defined the conditions in which the AVC strategies may be used, we discuss in the following sections, various FA-based string processing algorithms that rely on different AVC strategy instantiations. We start by the TD-AVC-based string processing algorithm in the next section.

6.3 The Table-driven-AVC algorithm

The table-driven-AVC based algorithm refers to the formalism

$$\forall s : \mathcal{V}^* \bullet \rho(\Delta, s) = \rho_{_{CV}}(\Delta_t, \varnothing, V_t, 0, s) \text{ where } 0 < V_t < |\mathcal{Q}_t|$$

For this algorithm, a virtual cache of up to V_t states is reserved for acceptance testing. We use an auxiliary array $c_{[0:V_t)}$ whose entries represent states currently in the virtual cache. Furthermore, an array $i_{[0:n)}$ of boolean is used as an indicator for establishing whether a state q is in the cache or not. Therefore: $\forall q \in [0..n)$, $i_q = true$ is construed to mean that the state is in the cache; and $i_q = false$ is construed to mean that the state is not in the cache. Another auxiliary array $m_{[0:n)}$ is used to keep track of the *position* of states either out of cache (i.e. in $[V_t..n)$) or in cache (i.e. in $[0..V_t)$). Therefore, access to state information in the table is done indirectly through m. An integer ℓ referred to as *cache line controller* is used to keep track of the current cache line at runtime. Initially, $\ell = 0$; meaning that there is *virtually*[1] no state in the cache. While c's entries are initialized to point to the first V_t states, entries of i are all initialized to `F`, indicating that nothing has yet been physically moved into the cache.

The AVC strategy ensures that the first V_t different states that are encountered on the string path during acceptance testing are placed in the virtual cache exactly in that order. This ordering arises because whenever a reference is made to a state out of the cache, and the cache is not full, then that state is brought into the next position in cache (this next position being indicated by the cache line). In each such case, its information is swapped with that of the state currently in that cache line, and the cache line is advanced. The cache is said to be full when $\ell = V_t$. At this stage, state replacement based on a cache replacement policy takes place.

Algorithm 6.3.1 provides the pseudocode for the TD-AVC algorithm. At the start, the cache is regarded as empty, and therefore $i_{[0:n)}$ is initialized to `F`. However, the first V_t states are *physically* in the memory area that is to be used for cache. Therefore,

[1] At the start, state transition data is of course physically in cache but because the data has not specifically been moved there, we consider the cache to be empty at that stage.

the array $c_{[0:V_t)}$ is initialized so that $\forall j \in [0..V_t) \centerdot c_j = j$. Similarly, the auxiliary array $m_{[0:n)}$ is initialized to reflect the fact that state q is initially in position q, i.e. $\forall q \in [0..n) \centerdot m_q = q$.

The variables and function used in the algorithm are as follows:

n: the number of states in the FA;

a: the alphabet size (which does not appear explicitly in the algorithm, but is implicitly used when swapping state information);

δ: the transition function;

s: the input string to be tested for acceptance;

V_t: the size of the virtual cache;

ℓ: the cache controller which indicates how much of the cache has been used;

$m_{[0:n)}$: an auxiliary array such that at any stage, for any index q, m_q holds the position of state q in the memory;

$i_{[0:n)}$: an auxiliary array whose entries indicate whether a state q is in the cache ($i_q = \mathtt{T}$) or not ($i_q = \mathtt{F}$);

$c_{[0:V_t)}$: an auxiliary array whose entries are states currently in the memory area that is to be regarded as cache, irrespective of whether that area is currently being used as cache or not;

p: a variable used to determine the position of the state to be swapped out of the cache or to interchange state's positions in the cache;

q: the state currently being processed;

j: the position of the symbol currently being tested for acceptance in s;

$swd(\delta[m_k], \delta[m_j])$: a function that interchanges not only data at rows m_k and m_j of the transition table, but also entries m_k and m_j of the auxiliary array m;

A loop invariant to which the loop's body should conform, $inv()$, is the predicate defined below:

$$inv \triangleq \forall i[0, n) \centerdot (i_q \neq \mathtt{F}) \Rightarrow$$
$$m_q \in [0, V_t) \wedge [((\ell < V_t) \wedge isInCacheLine(m_q)) \vee ((\ell \geq V_t) \wedge isInCache(m_q))]$$

Algorithm 6.3.1 (Table-driven based on allocated virtual caching)

---

func $tdavc(\delta, V_t, s)$: **boolean**
  $q, j, p, \ell := 0, 0, 0, 0;$
  $m_{[0:n)} := [0..n);$
  $c_{[0:V_t)} := [0..V_t);$
  $i_{[0:n)} := \mathtt{F};$
  do $(j < s.len()) \wedge (q \geq 0) \rightarrow$
    if $\ell < V_t \rightarrow$
      if $\neg i_q \rightarrow$
        if $q = c_\ell \rightarrow i_q = \mathtt{T}$
        $[\!]$ $q \neq c_\ell \rightarrow$
          $p := c_\ell;$
          $swd(\delta[m_q], \delta[m_p]);$
          $i_q, i_p, c_\ell := \mathtt{T}, \mathtt{F}, q$
        fi
      $[\!]$ $i_q \rightarrow$ **skip**
      fi;
      $\ell := \ell + 1$
    $[\!]$ $\ell \geq V_t \rightarrow$
      if $\neg i_q \rightarrow$
        $p := MOD(m_q, V_t);$
        $swd(\delta[m_q], \delta[m_{c_p}]);$
        $i_q, i_{c_p}, c_p := \mathtt{T}, \mathtt{F}, q$
      $[\!]$ $i_q \rightarrow$ **skip**
      fi
    fi;
    $q, j := \delta(m_q, s_j), j + 1$
  od;
  return $(q \geq 0)$
cnuf

The loop invariant articulates the nature of $i_{[0:n)}$, namely that the q^{th} entry of i is either F in this case it is out of cache, or: (1) the cache is not full and the row m_q of the transition table is in the appropriate cache line in the cache; (2) the cache is full and the row m_q of the transition table is simply in the cache.
In the point (1) above, the condition for matching the current cache line when the cache is not full is made by the predicate $isInCacheLine(m_q)$, which ensures that the first V_t states to be processed in the cache are ordered on a contiguous fashion. However, when full, point (2) above articulates that there is no more need to ensure ordering of states. This is made by the predicate $isInCache(m_q)$, that uses replacement policy to swap m_q is the cache before acceptance testing.

In the algorithm, for every iteration of the main loop, a test is made to check whether the cache is full or not. This is done by using a variable ℓ that keeps track of the number of states already processed in the cache. The variable is also used to make sure that the data of the first V_t states are in the cache.

In order to process an arbitrary next state q, if the cache controller ℓ is less than V_t, then the cache is not full. In this case, a test evaluates whether the state q is already flagged as being in the cache or not. If the state has not been flagged as being in the cache, then a further test is carried out to see whether the state q happens to correspond to the state currently in the cache line. If it does (i.e. if $q = c_\ell$), then nothing further needs to happen, apart from flagging this fact by setting i_q appropriately. If the state is not in the cache, since the cache is not full, the state's information is swapped with that of the state c_ℓ, currently in that cache line. As a result, the arrays i, c and m are updated accordingly. That is, i_q is now set to `T`, and i_p is set to `F`, since the state p has been swapped out of the cache. Also, the array c and m are updated accordingly.

If the state is in the cache, nothing is done —**skip**. At the end of this part of the algorithm, the cache line controller is incremented.

If the cache controller ℓ is greater than or equal to V_t, then the cache is full. At this stage, the ordering of states according to the string path is no longer respected. Thus, the way in which to handle a state out of the cache is determined by means of some preselected replacement policy. For simplicity, our algorithm above uses the direct mapping policy. The processing of a state under this condition requires yet another test. If the state is in the cache, nothing is done (**skip**). Otherwise, the direct mapping policy entails the computation of the modulus division of m_q by V_t yielding some index into cache, say p, which points to the state that will be swapped out of cache. The function $swd(\delta[m_q], \delta[m_{c_p}])$ is invoked to interchange the rows m_q and m_{c_p} of the transition table as well as to interchange entries m_q and m_{c_p} of the array m. The indicators i_{c_p} and i_q are switched to `F` and `T` respectively, and the entry c_p of the array c is changed to q so as to reflect the fact that state q is now in the p^{th} position in cache.

At the end of either of the above controls, acceptance testing takes place on the row m_q of δ followed by the update of the current index j of the string s. The algorithm ends when no more string is being processed or the automaton has reached a sink-state.

This version of the TD-AVC algorithm not only ensures that states being processed are available in the cache, but also that acceptance testing occurs in cache for every state to be processed. A variation of the algorithm could be provided such that states are swapped into the cache until the cache is full. Thereafter, no more swapping into cache takes place. This approach will be considered when dealing with performance in Part III. The hardcoded version of the TD-AVC algorithm is discussed in the next section.

6.4 The hardcoded-AVC algorithm

The HC-AVC based algorithm corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho(\Delta, s) = \rho_{_{CV}}(\varnothing, \Delta_h, 0, V_h, s) \text{ where } 0 < V_h < |\mathcal{Q}_h|$$

Its implementation is based on the same principle as that of the TD-AVC algorithm. In the HC-AVC algorithm, a virtual cache is allocated in the memory block that holds directly executable instructions of the automaton's states. A cache controller is used to ensure that instructions relating to the first V_h states of the string path are in the cache. When the cache is full and a reference is made to a state whose address is out of the allocated virtual cache, then the direct mapping policy is used to determine the state within the virtual cache whose information should be swapped with that of the current state.

Since in hardcoding states are referenced by the starting address of their directly executable instructions, the auxiliary array used to reference the state's position in the virtual cache and in memory still hold automaton states instead of addresses —as might be expected. This mechanism is used for simplicity. Having the state's index enables one to determine its address in memory as explained below.

Consider an automaton of n states, whose first state is hardcoded at address top, and the size of directly executable instructions for each state is Z bytes. If V_h represents the maximum number of states to be available in the virtual cache, and the cache is initially in $[0..V_h)$, therefore, the memory address range it occupies is in $[top, top + Z * V_t)$. As for the TD-AVC algorithm, we use the arrays $i_{[0:n)}$ whose entries indicate whether a state is in the cache or not. The arrays $c_{[0:V_t)}$ and $m_{[0:n)}$ would still be used to hold the states currently in the cache and their position in memory respectively. Therefore, the address of a state q which is at position m_q is $top + Z * m_q$. States swapping in hardcoding is different from that of the table-driven version in that, not all the instructions are swapped during the process. Since hardcoded states are made of directly executable instructions, many fields that make up an instruction are similar from one state to another. Therefore in practice, only the fields representing the state to be transited to ought to be swapped and not the whole set of instructions that describes a state transition.

Algorithm 6.4.1 gives the pseudocode for the HC-AVC algorithm. The variables, used in the algorithm are similar to those of the TD-AVC algorithm and additional functions and variables used are as follows:

top: the start address where the first state is hardcoded in memory;

Z: the size of the block of directly executable instruction that make up a state;

A and B: variables used to calculate the address of states in memory;

$exec@(B)$: executes the instructions starting at address B and returns new value for q;

$genhc(\delta, n, top, s)$: generates the hardcoded instructions of the recognizer starting from the memory address top;

$swv(A, B)$: This instruction interchanges the *necessary fields*, as explained below, of the instruction that make up the state at address A with those of the state at address B.

$sw(m_q, m_p)$: the standard function that interchanges the entries m_q and m_p of the array m.

By *necessary fields* in relation to the *swv* function mentioned above, we mean fields that are specific to a given state. In the hardcoded context, state transitions are described in terms of blocks of instructions of identical size for the whole FA. Many of these instructions are identical from one state to the next, while some instructions, such as *jumping to a next state*, *jumping to a rejecting state*, vary from one state to another. This because the latter instructions represent actions that are proper to the state itself. Thus, when swapping states, instead of swapping all the state instructions, in is more efficient to only interchange fields that contain instructions that do indeed change.

Algorithm 6.4.1 (Hardcoded based on allocated virtual caching)

---

func $hcavc(\delta, V_h, top, Z, s)$: **boolean**
  $q, j, p, \ell := 0, 0, 0, 0;$
  $c_{[0:V_t)} := [0..V_t);$
  $m_{[0:n)} := [0..n);$
  $i_{[0:n)} := \mathtt{F};$
  $genhc(\delta, n, top, s);$
  do $(j < s.len()) \wedge (q \geq 0) \rightarrow$
    if $\ell < V_h \rightarrow$
      if $i_q \rightarrow$ **skip**
      $[\!]$ $\neg i_q \rightarrow$
        if $q = c_\ell \rightarrow i_q = \mathtt{T}$
        $[\!]$ $q \neq c_\ell \rightarrow$
          $p, A, B := c_\ell, top + Z * m_q, top + Z * m_p;$
          $swv(A, B); sw(m_q, m_p); i_q, i_p, c_\ell := \mathtt{T}, \mathtt{F}, q$
        fi
      fi;
      $\ell := \ell + 1$
    $[\!]$ $\ell \geq V_h$
      if $\neg i_q \rightarrow$
        $p, A, B := MOD(m_q, V_t), top + Z * m_q, top + Z * m_{c_p};$
        $swv(A, B); sw(m_q, m_p);$
        $i_q, i_{c_p}, c_p := \mathtt{T}, \mathtt{F}, q$
      $[\!]$ $i_q \rightarrow$ **skip**
      fi
    fi;
    $B := top + Z * m_q; q, j := exec@(B), j + 1$
  od;
  return $(q \geq 0)$
cnuf

A loop invariant to which the loop body should conform is similar to the invariant discussed in relation to the TD-AVC algorithm. For every iteration of the main loop, when the cache is not full, a state must be in the cache and more precisely in the appropriate cache line for acceptance testing to take place. However, when the cache is full, a state must simply be in the cache for acceptance testing to take place.

In practice, the HC-AVC algorithm would be quite challenging to implement in a high-level language, since it requires self modification of directly executable instructions at runtime. Techniques such as writing self-modifying code [Hyd03] could be used to overcome this complexity. However, the most plausible approach is to use assembly language, as it enables one to make sure that the states' code size remains identical at runtime without any compiler intervention.

Another variation of the implementation of the hardcoded AVC algorithm is to only generate hardcoded states within the portion of memory specified by the virtual cache such that reference to a state that has not been generated could be copied in the cache using information in the table. This approach may reduce instruction size load and further improve performance. The performance of the HC-AVC algorithm is briefly discussed in part III. In the next section, another variation of the AVC-based algorithm that combines both HC-AVC and TD-AVC, referred to as the MM-AVC algorithm, is discussed.

6.5 The mixed-mode-AVC algorithm

The various formalisms of the MM-AVC algorithms were given in Property 8. In this section, we discuss the *complete* MM-AVC algorithm that corresponds to the case where both TD and HC have their strategies strictly less than their respective number of states. For an n states automaton, the MM-AVC algorithm requires the transition set to be split into two disjoints subsets such that one is hardcoded and the other is table-driven. For this algorithm, we assume that the first k states of the automaton are table-driven, and the remaining $n - k$ states are hardcoded[2]. Two arguments V_t and V_h are used to represent the threshold of the allocated virtual caches for both TD and HC respectively. Furthermore, two auxiliary arrays $m^t_{[0:k)}$ and $m^h_{[0:n-k)}$ are used to hold the positions of the states in memory in order to determine whether a visited state is part of the virtual cache or not. The variables ℓ_t and ℓ_h are the cache line controllers for the TD and HC part of the algorithm respectively. The following scenarios are envisaged upon accessing a state in mixed-mode:

The state is hardcoded: in this case, the portion of the code related to hardcoding is invoked. The logic of this code corresponds to the logic of the HC-AVC algorithm discussed in the previous section.

The state is table-driven: in this case, the string is processed in the table-driven portion of the code. Again, the logic of this corresponds to that of the TD-AVC algorithm discussed above.

[2]The assumption obviously means that $k = |\mathcal{Q}_t|$ and $n - k = |\mathcal{Q}_h|$, since $n = |\mathcal{Q}_m| = |\mathcal{Q}|$.

Algorithm 6.5.1 provides a simplified pseudocode of the *complete* MM-AVC algorithm. The algorithm basically consists of a table-driven part and a hardcoded part. At start, a test is first made as to check whether the current state is hardcoded or not. The table-driven function $td(\delta, V_t, \ell_t, k, m^t, i^t, c^t, s, q, j)$ is invoked when the state q currently being processed is less than k. In this case, acceptance testing takes place in the table using the transition function δ. The function processing is similar to that of the TD-AVC algorithm; at the end of the operation, the various auxiliary arrays as well as the value of q and j are assumed to have been updated. If the next state to be processed is greater than k, then the hardcoded function

$$hc(V_h, \ell_h, top, Z, n-k, A, B, i^h, c^h, m^h, s, q, j)$$

is invoked, taking as parameter all the necessary auxiliary arrays, the address *top* where the first hardcoded state has been written, the size of a hardcoded state Z in bytes, the variables A and B used for states' address calculation, as well as the current state q and the current string index j. At the end of a hardcoded state processing, the variables q and j point to the next state to be processed and the next symbol to be scanned respectively.

We have chosen to provide a simplistic version of the algorithm without providing details of both $hc()$ and $td()$ function for space economy. In effect, the details of both functions are straightforward as they merely correspond to the TD-AVC and HC-AVC algorithms, respectively. The variables and functions used in this algorithm correspond to those used in both TD-AVC and HC-AVC algorithms. The variables subscripted by a $_t$ represent those related to the table-driven portion of the code and those subscripted by an $_h$ corresponds to the hardcoded part. The reader should notice that in depicting the $hc()$ function, the array m^h is defined within $[0..n-k)$; and is used to hold the position of the hardcoded states that are every state $q \in [k, n-k)$. Therefore, the corresponding entry of a state q in m^h would be $m^h[q-m]$ (or simply m^h_{q-m}) instead of $m^h[q]$ (i.e. m^h_q). In the algorithm, the two variables that hold the virtual cache thresholds (V_t and V_h) are independent of one another. Also, the replacement policy used for states swapping for either virtual cache may be different from one another. In the present algorithm however, we use the same replacement policy for both TD and HC.

Of course changing replacement policy may yield different latency in terms of performance. The function $genhc()$ in the algorithm generates the hardcoded directly executable instructions of the $n-k$ states of the automaton that are supposed to be hardcoded. The address of a hardcoded state at m_q is obtained using the formula $top + Z * m_q$ as was the case for the HC-AVC algorithm.

Algorithm 6.5.1 (Mixed-mode based on allocated virtual caching)

func $mmavc(\delta, V_t, V_h, top, Z, k, s)$: **boolean**
$\quad q, j, p, \ell_t, \ell_h := 0, 0, 0, 0, 0;$
$\quad m^t_{[0:k)}, c^t_{[0:V_t)} := [0..k), [0..V_t);$
$\quad i^t_{[0:k)} := \mathtt{F};$
$\quad m^h_{[0:n-k)}, c^h_{[0:V_h)} := [0..n-k), [0..V_h);$

$i^h_{[0:n-k)} := \mathtt{F}$;
$genhc(\delta, n-k, V_h, top, s)$;
do $(j < s.len() \wedge q \geq 0) \rightarrow$
  if $q < k \rightarrow$ {table-driven}
    $td(\delta, V_t, \ell_t, k, m^t, i^t, c^t, s, q, j)$
  $[\!]$ $q \geq m \rightarrow$ {hardcoded}
    $hc(V_h, \ell_h, top, Z, n-k, A, B, i^h, c^h, m^h, s, q)$
  fi
od;
return $(q \geq 0)$
cnuf

The *complete* MM-AVC algorithm provided in this section is merely for illustrative purposes. We could have discussed other variations such as the *strong on TD/HC* MM-AVC algorithms. However, their implementations are straightforward and would only be of importance for practical reasons, i.e. when analyzing the performance of the algorithms. In the next section we provide an illustrative example of the FA-based string processing algorithms based on the AVC strategy.

6.6 Illustrative example

In this section, we use a practical example to show how the TD-AVC (and therefore the HC-AVC or MM-AVC) algorithm works. Consider for example an automaton $M(\mathcal{Q}, \mathcal{V}, \Delta, s_0, \mathcal{F})$ where $s_0 = 0$, $\mathcal{V} = \{a, b, c\}$,
$Q = F = \{0, 1, 2, 3, 4, 5, 6, 7\}$, and δ is defined by a two-dimensional array, given by the left-hand table in Table 6.1. This automaton is *partially* represented in Figure 6.1, in that it only shows transitions that will be followed when the string *abcabcaabccaabcc* is being recognized. For this example, we assume that our virtual cache can hold a maximum of four ($V_t = 4$) states. That is, the virtual cache in memory is in the range [0..3]; therefore the states 0,1,2 and 3 are all physically in the cache at start, but not necessarily ordered according to the string path.

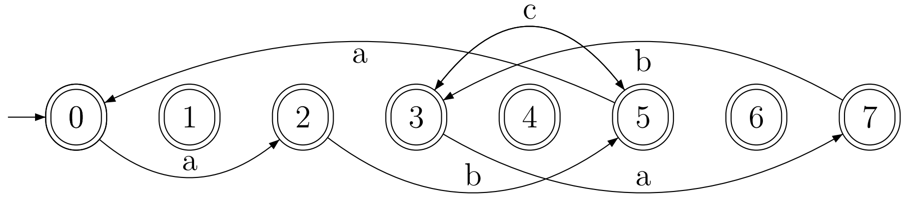

Figure 6.1. A State diagram for testing the string *abcabcaabccaabcc*

The strings *abcabcaabccaabcc* can be processed using the TD-AVC algorithm as follows:

δ			
	a	b	c
0	2	3	1
1	2	5	4
2	1	5	2
3	7	2	5
4	4	6	0
5	0	6	3
6	1	5	3
7	7	3	4

m	
0	0
1	1
2	2
3	3
4	4
5	5
6	6
7	7

i	
0	F
1	F
2	F
3	F
4	F
5	F
6	F
7	F

c	
0	0
1	1
2	2
3	3

Table 6.1. The arrays δ, m, i, and c initially for the AVC example

Initial phase:

After initialization the following holds:
δ is the first table of Table 6.1. (Thus, $\delta(0, a) = 2$, $\delta(0, b) = 3$, etc.)
$s = abcabcaabccaabcc$
$s.len() = 16$
$q, j, p, l := 0, 0, 0, 0$
$i = \{\mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F}, \mathtt{F}\}$
$c = \{0, 1, 2, 3\}$
$m = \{0, 1, 2, 3, 4, 5, 6\}$
$V_t = 4$

The first iteration:

At this stage, all the conditions to enter the loop are satisfied. Therefore, the loop is executed. A test is made on ℓ to see whether the cache is not full. Since $\ell = 0 < V_t$, another test is made to see whether the current state is in the cache or not. Since $i_q = i_0 = \mathtt{F}$, the state is assumed not to be in the cache. However, it turns out that there is a match between the current state and the state in the cache line. Therefore, i_0 is switched to F, the cache line controller is incremented to 1 and acceptance testing takes place in the cache as follows: q is assigned $\delta(m_0, s_0) = \delta(0, a) = 2$ and
$j = j + 1 = 1$ is incremented to point to the next symbol.

Second iteration

At this stage, $\ell = 1$, $q = 2$, and $j = 1$. A test on the cache line shows that the cache is not full. However, the current state is not in the cache line pointed to by $\ell = 1$, since $m_2 \neq c_1$. Thus, the following instructions must be executed:
The variable p is assigned c_ℓ ie $p = 1$
The function $sw(c_{m_q}, c_\ell)$ since $m_q = m_2 = 2$ and $c_\ell = 1$, the function corresponds to $sw(c_2, c_1)$ resulting in $c_2 = 1$ and $c_1 = 2$
The function $sw(\delta[m_q], \delta[m_p])$ is invoked;

since $m_q = m_2 = 2$, and $m_p = m_1 = 1$, the entries corresponding to rows 2 and 1 of the transition table are swapped. Furthermore, m_2 now holds the value 1 and m_1 holds the value 2. Table 6.2 shows the content of δ and m, c and i after the above sequence of instructions have been executed.

δ	a	b	c
0	2	3	1
1	**1**	**5**	**2**
2	**2**	**5**	**4**
3	7	2	5
4	4	6	0
5	0	6	3
6	1	5	3
7	7	3	4

m	
0	0
1	**2**
2	**1**
3	3
4	4
5	5
6	6
7	7

i	
0	*T*
1	F
2	T
3	F
4	F
5	F
6	F
7	F

c	
0	0
1	**2**
2	**1**
3	3

Table 6.2. The arrays δ, m, i, and c after the second iteration for the AVC example

The cache line controller is now incremented to 2, and acceptance takes place within the virtual cache on state 2. The current symbol to be processed is b, which triggers a transition to state 5, and j is incremented to 2.

Third to sixth iterations

At this stage, $j = 2$, $\ell = 2$, and $q = 5$. The cache is not full, but $i_5 = F$ indicates that the state in not in the cache. Thus the following instructions are executed:
The variable p is assigned c_ℓ that is, $p = c_2 = 1$ which represents the state to be swapped out of the cache.
The function $swd(\delta[m_5], \delta[m_1])$ is invoked in order to swap states information. Now $m_5 = 5$, but $m_1 = 2$; meaning that, information on state 1 are now at position 2 in δ. Thus those information are swapped, so are entries $m_q = m_5$ and $m_p = m_1$ of the table.
The variables, $i_q = i_5$, $i_p = i_1$, and $c_\ell = c_2$ are updated respectively to `T`, `F`, and 5. Meaning that the state 5 is now in the cache, and the state 1 is out of the cache. Table 6.3 shows the content of the δ and m, c and i after states interchanges.

The cache line controller is now incremented to 3, acceptance takes place within the virtual cache on state 5. The current symbol to be processed is c, that triggers a transition to state 3, and the string index is incremented to 3. The next state to be transited to is 3 which is in the virtual cache and matches the state in the current cache line ($q = 3 = c_3$). Thus, the state indicator is switched to 0, and the cache line is incremented to 4. At this stage, the next symbol to be processed is a which triggers a transition for state 3 to state 7, and the string index is incremented to 4. This later case was the fourth iteration of the main loop.

δ			
	a	b	c
0	2	3	1
1	*1*	*5*	*2*
2	**0**	**6**	**3**
3	7	2	5
4	4	6	0
5	**2**	**5**	**4**
6	1	5	3
7	7	3	4

m	
0	0
1	**5**
2	*1*
3	3
4	4
5	**2**
6	6
7	7

i	
0	T
1	F
2	F
3	F
4	F
5	T
6	F
7	F

c	
0	0
1	*2*
2	**5**
3	3

Table 6.3. The arrays δ, m, i, and c after the third iteration for the AVC example

The reader may verify that after the fifth and sixth iterations, the contents of δ and m, c and i are as depicted by Table 6.4 and Table 6.5 respectively. At this stage the substring *abcabca* has already been processed.

δ			
	a	b	c
0	2	3	1
1	*1*	*5*	*2*
2	*0*	*6*	*3*
3	**7**	**3**	**4**
4	4	6	0
5	*2*	*5*	*4*
6	1	5	3
7	**7**	**2**	**5**

m	
0	0
1	*5*
2	*1*
3	**7**
4	4
5	*2*
6	6
7	**3**

i	
0	T
1	*F*
2	F
3	F
4	F
5	*T*
6	F
7	T

c	
0	0
1	*2*
2	*5*
3	**7**

Table 6.4. The arrays δ, m, i, and c after the fifth iteration for the AVC example

Later iterations

After processing the substring *abcabca*, all the necessary states are now in the cache and grouped together on a contiguous fashion. It then follows that subsequent iterations would no longer require state replacement. This results in processing the remaining substring at optimum with the advantage that, states that should be accessed are well organized as to minimize cache misses.

The illustrative example provided in this section relies on the TD-AVC algorithm. However, the same could be applied on both HC-AVC and MM-AVC algorithms. The only difference would be the implementation approach which would require a slightly more verbose explanation than the present one.

In the next section, we briefly discuss the advantages and drawbacks of the AVC algorithms compared to their preliminary counterparts.

δ	a	b	c
0	2	3	1
1	*1*	*5*	*2*
2	*0*	*6*	*3*
3	**7**	**2**	**5**
4	4	6	0
5	*2*	*5*	*4*
6	1	5	3
7	**7**	**3**	**4**

m	
0	0
1	*5*
2	*1*
3	**3**
4	4
5	*2*
6	6
7	**7**

i	
0	T
1	*F*
2	T
3	T
4	F
5	*T*
6	F
7	F

c	
0	0
1	*2*
2	*5*
3	**3**

Table 6.5. The arrays δ, m, i, and c after the sixth iteration for the AVC

6.7 Theoretical assessment

In this section, a brief survey is made of the conditions that may lead to a better performance of the AVC-based algorithms over their core counterparts and vice-versa.

One of the obvious drawback of the AVC algorithms discussed in this chapter is the time required, not only to perform state replacement but also to access a transition. In effect, when core algorithms are considered, accessing a state's information is done directly using the offset to that state in memory. Furthermore, performing acceptance testing is straightforward, provided that both the state and the symbol to be tested are available. In this regard, for a given state and a given symbol, testing whether a transition exists or not is done directly by accessing the desired information in memory with no further additional operation. Such an approach ensures that information access is at optimum, whereas the AVC algorithms require the use of an auxiliary array. In the same context, the core algorithms do not allow for state interchange, which ensures optimality compared to the AVC algorithms where, upon entering a state, a test is made on whether the cache is full or not, followed by various tests on whether the state is in the appropriate cache line when the cache is not full, as well as test on whether a state is in the cache when the cache is full. These various test are indeed time consuming.

Another non-negligible drawback of the AVC algorithm is the replacement strategy used. The modulus operation used to make a decision on which state to swap out of the cache may not always be the best one. A poor performance may be observed if a state has to be constantly swapped in and out of the cache. Of course other approaches such as the LRU, associative mapping could be used to minimized such unwanted effects.

The AVC algorithms should then be used carefully and under the following two broad conditions:

Strings made of long sequences: If the string being processed is made of long sequences such that part of it frequently visits the same set of states, the AVC algorithm may be of interest in the sense that, the cost for replacing the states for their contiguous organization, and accessing states' information via an auxiliary array may be minimized. In effect, states replacement could be useful in organizing the virtual cache such that information that are frequently present are those frequently used. If this is the case, an AVC-based algorithm might outperform its core counterpart. As a result, the performance of the AVC algorithm in relation to its TD counterpart heavily depends on the kind of string being tested for acceptance.

Unorganized transition table: When processing a string whose states to be visited are organized in memory on a non-uniform fashion, the probability of cache misses increases, with a consequent degradation of performance when using core algorithms. The AVC algorithms may outperform their core counterparts in such a context since the virtual cache is designed as to hold frequently visited states together, provided that a good replacement policy is used. The size of the cache, as well as the kind of strings being processed appear in this context to be of importance in shaping an AVC algorithm in order to avoid states constantly being swopped in and out of the cache.

Further exploration of the AVC strategy for FA-based string processing is thus of importance in that it may be possible to characterise more precisely the contexts in which efficiency is obtained. The context depends not only on the layout of states to be visited by the string during acceptance testing, but also the kind of strings to be processed. A summary of the chapter is provided in the next section.

6.8 Summary of the Chapter

In this chapter, we have suggested new FA-based string processing algorithms that use the Allocated Virtual Caching strategy to perform acceptance testing. After characterizing the strategy, various pseudocode algorithms were provided. The AVC algorithms appear to be of importance in testing strings made of long sequence, so that the time-effects of state replacement and indirect states access may be amortized over the long run, provided that a limited number of states are continuously visited. The AVC strategy appears to relate to the DSA strategy discussed in Chapter 4; however AVC exploits a portion of the memory already in used by the FA instead of creating memory blocks on the fly as is the case for the DSA strategy.

With this chapter, we have completed the list of the various algorithms under investigation. The performance of most of the algorithms is to be discussed in Part III where an empirical investigation is made on the advantages and drawbacks of each of the selected algorithms.

The provision of a formal characterization for each of the algorithms investigated thus far can be extended to indicate how the various strategies discussed to date can be combined to produce new algorithms. The next chapter on taxonomy addresses these issues.

CHAPTER 7

TAXONOMY OF FA-BASED STRING PROCESSORS

In this chapter the characterization of FA-based string processing algorithms is redefined, taking in consideration the various implementation strategies discussed thus far. As a result, a unified formalism is obtained that serves as basis for the construction of a taxonomy of FA-based string processing algorithms. The taxonomy comprises not only algorithms already discussed, but also new ones obtained by combining existing implementation strategies.

The chapter starts with a section on the related work on taxonomy, in which, the term *taxonomy* is defined, as well as discussions on its construction, implementation and usage. The section also depicts a brief survey of some related work on taxonomy of various algorithmic-related problem domains. By problem domain, we mean an algorithmic (computational) problem that admits several and different solutions, but somehow related to each other since their primary goal is to solve the problem. The difference between the solutions may be their performances, their implementation strategies, their datastructures, etc. The purpose of a taxonomy is then to establish the relationships between the solutions, and present them within a single framework for further usage.

The chapter then continues with discussions on the general characterization of FA-based string processing used to construct the taxonomy tree, whose nodes (algorithms) are derived using cross-product of sets. Such a characterization results in derivations of various formalisms, each corresponding to an FA-based string processing algorithm. The taxonomy graph constructed toward the end of the chapter forms the basis for the design of a high level class-diagram which represents an architectural view of the toolkit, and its implementation discussed in Chapter 8.

7.1 Related Work

The McGraw-Hill dictionary defines *taxonomy* as

> *A study aimed at producing a hierarchical system of classification of organisms which best reflects the totally of similarities and differences* [MHP02].

Although the definition is biology-oriented, the term taxonomy may refer in general to either the classification of objects, or the principles underlying the classification. Therefore, almost anything (animate objects, inanimate objects, places, and events, etc.) may be classified according to some taxonomic scheme. Taxonomies are frequently hierarchical in structure. However taxonomy may also refer to relationship schemes other than hierarchies, such as network structures, class hierarchies, software components classification, and the like. A taxonomy might also be a simple organization of objects into families, groups, or even an alphabetical list.

In the context of software construction and algorithmics, a taxonomy could be thought of as *a hierarchical classification of algorithms pertaining to a problem domain*. The relationships defined between the different algorithms are usually their *performances*, the underlying *datastructures*, or simply the *strategies* used in the elaboration of the algorithms. For example, consider a problem P that can be solved algorithmically in three different ways through algorithms A, B, and C. Assume that algorithm B was obtained from A through some sort of improvement of A's datastructures, and that C was obtained through some improvement of A's efficiency. The solutions of P may be classified on a hierarchical fashion such that, A is at the top of the hierarchy, and B and C are at the bottom of the hierarchy. The two solutions at the bottom of the hierarchy are said to be the *derivations* of A, that were constructed by *refining* A. Refinement simply means that more details are used to modify A in order to obtain B and C[1]. Algorithms B and C are only related in that, they share a common goal (solution to the problem); since it may not be possible to obtain B from C and vice-versa. Nonetheless they both have a strong relationship with A. Such a hierarchical classification of the solutions to the problem P is referred to as the taxonomy of P's algorithms.

In general, an author working on a taxonomy of algorithmic problem domains usually starts by formulating a naïve solution of the problem, and then deriving various alternative solutions that could be viewed as children of the original solution. The taxonomy obtained could be presented in the form of a tree structure with a root node and several parents and children. In a taxonomy tree, terminal nodes are considered as solutions that do not yet have further refinements. However, the level of derivation depends on the author, since a different person working on the same kind of problem may provide more refinement strategies on those same "terminal nodes", resulting to even more nodes in the taxonomy tree.

Taxonomy construction in algorithmics usually relies on an intensive literature survey. Given a computational problem, the naïve solution is considered as the root in the taxonomy tree. Then follows other solutions that have been found in the literature. These have to be associated with nodes in the taxonomic tree according to the relational structure of the tree being built. In certain circumstances, the taxonomy builder may end-up with an isolated node in the tree, not directly related to the root nor to any other children of the root. This happens when the underlying relationship does not apply to the node, because a different approach has been used to make up that node. Since the taxonomy tree is built from existing solutions in the literature, one may think of its design as a bottom-up exercise, as opposed to the top-down approach whereby the root algorithm is used to derive nodes one level down the hierarchy and so forth. In this case, it is often impossible to end up with isolated nodes since the derivation has been applied in a logically constrained and formal manner.

[1]The term refinement is used here in a rather loose sense, and not in the formal sense of some or other refinement calculus, as for example the as defined by [Mor94]

After constructing the taxonomy tree, a unified implementation approach is needed to make it computationally usable. This is helpful for the practical exploitation of the algorithms. The implementation is done through a class-library implementation, or simply a toolkit implementation in the object-oriented terminology. In order to obtain the corresponding toolkit, the taxonomy designer maps the taxonomy graph to a high level class diagram. Each node of the taxonomy becomes a class component with data members and member functions. Some of the classes in the diagram may be abstract classes, necessary in the structure for the implementation of its derived concrete classes. Once mapping is completed, its implementation is relatively straightforward using the so-called generic programming [Ale01]. The implemented toolkit may be accessed by means of a Domain Specific Langauge (DSL) built on top of it to allow users that are not concerned with implementation details to exploit the toolkit. Chapter 8 is devoted to discussions on toolkit design.

Our work on a taxonomy of FA-based string processors was inspired by that of Watson in [Wat95b]. In his work, he discussed the taxonomies and toolkits of various regular language algorithms whereby, using a naïve solution for each of the problem-domain, and after a intensive literature survey, a taxonomy tree was constructed according to the relationship between the naïve solution and algorithms found in the literature. In the process, new algorithms were derived either by improving existing algorithms, or by filling the gaps between them. His taxonomy graph was further converted into a toolkit, and algorithms implemented and compared according to some benchmarks. An extension of Watson's work by Cleophas et al. in [CW05] introduced the notion of DSL design and implementation on top of the toolkit, in order to accommodate users that are not interested on toolkit's implementation details.
Turpin et al. in [TPS05] suggested a taxonomy of suffix array construction algorithms. Unlike Watson's approach, the taxonomy suggested was based on the *appearance date* of various suffix tree array algorithms in the literature. Therefore, the relationship in the classification was more on the appearance date rather than implementation details.
The taxonomy of sorting algorithms was proposed by Broy in [Bro83] where the complete list of the various sorting algorithms is obtained based on a naïve solution. A taxonomy of garbage collectors was also suggested by Jonker in [Jon82].

Unlike the above approaches, a taxonomy of FA-based string processing algorithms cannot only rely on a literature survey. The reason is that very little has been done in exploring FA-based string processors, probably because users/researchers have not been sufficiently interested to explore other alternatives, particulary at the strategy (and therefore cache) level as it is the case in this work. Moreover the alternative hardcoded approach suggested by Thompson [Tho68] and further extended by Knuth [KMP77] appears to be efficient only for automata of relatively small size [Nga03]. Therefore, to the best of our knowledge, only two implementation approaches of FA-based string processors have been proposed in the literature to date. This clearly suggests that a preliminary taxonomy could not be proposed through literature survey[2].

Our taxonomy differs from the previous ones in that it is derived from the implementation strategies that have been discussed, rather than from data structures, appearance date, or performance. The refinement rule used for the derivation of the algorithms is the addition of new strategies to existing nodes in order to obtain new nodes in the taxonomy graph. Furthermore, we only rely on the so-called core algorithms (TD and/or HC) found in the literature for the derivation of new algorithms. In order to do so, we use the formal characterization of string recognizers in which the strategies are integrated to produce new formalisms —and therefore, new algorithms. The combination between the identified strategies yields even more formalisms, and therefore more algorithms. Although the constructed taxonomy graph is regarded as a preliminary one, it does not contain isolated nodes, and can be viewed as a *trie* rather than the more general *acyclic* graph produced by the Watson's taxonomy. In the next section, we provide a unified characterization of FA-based string processing algorithms that serves as basis for the construction of our taxonomy.

7.2 New Characterization of FA-based String Processors

In previous chapters we provided various formalisms for characterizing string recognizers. It was pointed out that the core formalism that characterized TD, HC and MM could be viewed as special cases of each of the formalisms discussed. The formalisms discussed resulted to new algorithms that were in theory, conjectured to be more efficient than their core counterparts —depending on the input string and the automaton's size. However, the characterizations were discussed independently, one chapter dealing with a single strategy. This section discusses a generalized formalism that may be used to characterize FA-based string recognizers, taking into consideration all the strategies previously described. The new characterization is used to derive not only all algorithms previously discussed, but also new ones.

Previous characterization of FA-based string recognizers revealed that a recognizer is function of its input string, its transition sets, and its associated strategy, be it C (for Core), D (for DSA), P (for SpO), or V (for AVC). By putting all the strategies together, we may now characterize a recognizer ρ_{CDPV} as a new function of: its input string s; its transition sets Δ_t and Δ_h; and all its strategies (D_t, D_h, P_t, P_h, V_t and V_h). Therefore, once the transitions sets have been specified, given an arbitrary input string, one may choose to implement ρ_{CDPV} using any of the strategies or some combination of them, as long as the necessary conditions on strategies are respected. The recognizer' denotational semantics is now formally expressed in general as follows:

$$\rho_{CDPV} : \mathcal{T} \times \mathcal{T} \times \mathbb{N} \times \mathbb{N} \times \mathbb{B} \times \mathbb{B} \times \mathbb{N} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B} \tag{7.1}$$

[2]Various unpublished FA-based string processors may exist in the private domain, within various organizations, but not accessible to researchers.

such that
if

$$\begin{cases} (\Delta_t \cup \Delta_h = \Delta) \wedge (\Delta_t \cap \Delta_h = \varnothing) \\ (0 \leq D_t \leq |\mathcal{Q}_t|) \wedge (0 \leq D_h \leq |\mathcal{Q}_h|) \\ (P_t \in \mathbb{B}) \wedge (P_h \in \mathbb{B}) \\ (0 \leq V_t < |\mathcal{Q}_t|) \wedge (0 \leq V_h < |\mathcal{Q}_h|) \end{cases}$$

then

$$\rho_{_{CDPV}}(\Delta_t, \Delta_h D_t, D_h, P_t, P_h, V_t, V_h, s) = \rho(\Delta, s)$$

The recognizer defined as such shows that, the strategies used depend on the nature of the transition set itself. Arguments subscripted with $_t$ are associated to the TD algorithm, whereas those subscripted with $_h$ are associated to the HC algorithm. It follows that when either of the transition set provided in the characterization is empty, there is no need to use its strategies. Therefore, depending on the transition set's cardinality we may derive three characterizations that are independent of each other, as given below:

1. *The TD characterization* is obtained if $\Delta_h = \varnothing$ since it implies $D_h = 0$, $P_h = \mathtt{F}$ and $V_h = 0$. Therefore, there is no need to use the strategy variables associated to HC when no state is hardcoded as specified in the problem domain. Without loss of generality, we may introduce a new function ρ_t, which is a table-driven recognizer that takes as input a string s, the strategy arguments D_t, P_t, and V_t, and returns a boolean as follows:

$$\rho_t : \mathcal{T} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B} \tag{7.2}$$

   such that

$$\text{if } \begin{cases} \Delta_t = \Delta \\ 0 \leq D_t \leq |\mathcal{Q}_t| \\ P_t \in \mathbb{B} \\ 0 \leq V_t < |\mathcal{Q}_t| \end{cases} \text{ then } \rho_t(\Delta_t, D_t, P_t, V_t, s) = \rho(\Delta, s)$$

   Furthermore, the following relationship holds:

$$\forall s : \mathcal{V}^* \bullet \rho_t(\Delta_t, D_t, P_t, V_t) \equiv \rho_{_{CDPV}}(\Delta_t, \varnothing, D_t, 0, P_t, \mathtt{F}, V_t, 0, s).$$

2. *The HC characterization* is obtained if $\Delta_t = \varnothing$ since it implies $D_t = 0$, $P_t = \mathtt{F}$ and $V_t = 0$. Therefore, there is no need to use the strategy variables associated to TD when no state is table-driven as specified in the problem domain. Without loss of generality, we may introduce a new function ρ_h, which is a table-driven

recognizer that takes as input a string s, the strategy arguments D_h, P_h, and V_h, and returns a boolean as follows:

$$\rho_h : \mathcal{T} \times \mathbb{N} \times \mathbb{B} \times \mathbb{N} \times \mathcal{V}^* \nrightarrow \mathbb{B} \tag{7.3}$$

such that

$$\text{if } \begin{cases} \Delta_h = \Delta \\ 0 \le D_h \le |\mathcal{Q}_h| \\ P_h \in \mathbb{B} \\ 0 \le V_h < |\mathcal{Q}_h| \end{cases} \text{ then } \rho_h(\Delta_h, D_h, P_h, V_h, s) = \rho(\Delta, s)$$

Furthermore, the following relationship holds:

$$\forall s : \mathcal{V}^* \bullet \rho_h(\Delta_h, D_h, P_h, V_h, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, D_h, \mathtt{F}, P_h, 0, V_h, s).$$

3. *The MM characterization* is obtained if $\Delta_t \neq \varnothing$ and $\Delta_h \neq \varnothing$. Therefore, Equation 7.1 holds, and the function ρ_m which characterizes the mixed-mode algorithm is identical to $\rho_{_{CDPV}}$. Thus, the following relationship holds:

$$\forall s : \mathcal{V}^* \bullet \tag{7.4}$$

$$\begin{cases} \rho_m(\Delta_t, \Delta_h, D_t, D_h, P_t, P_h, V_t, V_h, s) = \rho_{_{CDPV}}(\Delta_t, \Delta_h, D_t, D_h, P_t, P_h, V_t, V_h, s) \\ \rho_m(\Delta_t, \Delta_h, D_t, D_h, P_t, P_h, V_t, V_h, s) = \rho(\Delta, s) \end{cases}$$

The high-level formalisms associated with each type of algorithm may be used in the derivation of new algorithms using appropriate instances of their associated strategy variables. Furthermore, the strategies may also be combined with the aim of producing new formalisms —and therefore new algorithms. In the following subsections we discuss the derivations from each specialized algorithm. The TD derivations are elaborated in more detail than the HC and MM versions, since the logic used for the derivations remains essentially the same.

7.2.1 Derivation of the TD algorithms

The TD characterization is obtained when the HC transition set is empty. This results in the TD formalism being a function of five variables, with the variables Δ_t and s considered constants. Therefore, we may only instantiate the arguments D_t, P_t and V_t following their basic conditions. The boolean nature of the variable P_t implies that it is either *true* or $false$ ($P_t \in P = \{\mathtt{T}, \mathtt{F}\}$). In general, $D_t \in D = \{0, d, n\}$; that is the variable may hold the value 0 to mean that there is no DSA, a value $0 < d < |\mathcal{Q}_t|$ to mean that the DSA strategy used is *bounded*, or, the value n ($n = |\mathcal{Q}_t|$) to mean that the DSA strategy is *unbounded*. The argument $V_t \in V = \{0, v\}$; may be assigned the value 0 to mean that there is no AVC strategy, or a value $0 < v < |Q_t|$ since it is always bounded.

Table 7.1 depicts $3 \times 2 \times 2 = 12$ different algorithms based on the combination of the values that may be assigned to each strategy argument. The combination of strategy parameters has resulted not only in algorithms discussed in previous chapters, but also in new ones, to be discussed below. For ease of reference, the last column in the table informs the reader on the section in this thesis where the algorithm has been discussed. The first column in the table represents the triplets of the form (D_t, P_t, V_t), indicating instances of the strategy variables involved in the formalism. Each of those triplets corresponds either to an algorithm that has already been discussed in previous chapters, or to one that is to be discussed below. It is worth mentioning that each strategy argument that is assigned a "*neutral*" value is construed to mean that the strategy of concern is not involved in the construction of the algorithm. By neutral value, we mean value such as 0, for both DSA and AVC and $\mathtt{F}$ $(false)$ for the SpO strategy. The second column in the table lists the strategy involved in the algorithm, and the third column refers to the name associated to each algorithm.

The name given to each algorithm starts with the letter t followed by the concatenation of the numbers assigned to each active strategy as follows: the number 1 is assigned to the DSA strategy; the number 2 is assigned to the SpO strategy; and the number 3 is assigned to the AVC strategy. However, since there are two variations to the DSA strategy, we chose to prefix its number with: u for the unbounded case, and b for the bounded case. For example, t_{u1} refers to the table-driven algorithm based on the unbounded DSA strategy, t_{b123} refers to the tabled-driven algorithm based on the bounded DSA strategy combined with the SpO and AVC strategies; and of course t refers to the core table-driven algorithm.

Combination	Active strategy	Name	Reference
$(0, \mathtt{T}, 0)$	SpO	t_2	5.3
$(0, \mathtt{T}, v)$	SpO and AVC	t_{23}	7.2.1.1
$(0, \mathtt{F}, 0)$	None	t	3.1
$(0, \mathtt{F}, v)$	AVC	t_3	6.3
$(d, \mathtt{T}, 0)$	bounded DSA and SpO	t_{b12}	7.2.1.2
$(d, \mathtt{T}, v)$	bounded DSA, SpO and AVC	t_{b123}	7.2.1.3
$(d, \mathtt{F}, 0)$	bounded DSA	t_{b1}	4.2
$(d, \mathtt{F}, v)$	bounded DSA and AVC	t_{b13}	7.2.1.4
$(n, \mathtt{T}, 0)$	unbounded DSA and SpO	t_{u12}	7.2.1.5
$(n, \mathtt{T}, v)$	unbounded DSA, SpO and AVC	t_{u123}	7.2.1.6
$(n, \mathtt{F}, 0)$	unbounded DSA	t_{u1}	4.2
$(n, \mathtt{F}, v)$	unbounded DSA and AVC	t_{u13}	7.2.1.7

Table 7.1. The derived TD-based algorithms

A discussion of the new algorithms that have been suggested by the extended formalism that was introduced in this chapter, is given in the next subsections.

7.2.1.1 The TD-SpO-AVC algorithm

This algorithm, referred to as t_{23} in the table, corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_t(\Delta_t, 0, \mathtt{T}, v, s) \equiv \rho_{_{CDPV}}(\Delta_t, \varnothing, 0, 0, \mathtt{T}, \mathtt{F}, v, 0, s) \text{ with } 0 < v < n$$

It relies on both SpO and AVC strategies. A naïve approach for its implementation would be to first reorder the automaton's states using the function $reorder(\delta, p)$, and then invoke a function $tdAvc(\delta, p, m, c, i, \ell, V_t, j, q, s)$ (for every iteration of the main loop) that updates the next state q to be transited to, as well as the next index j of the string s currently being processed. This latter function also takes as parameters the arrays m, c, and i as well as the cache line controller, ℓ, previously described in Chapter 6. Moreover, access to the original information of a state is made via entries of the array p. The algorithm below gives the pseudo-code for algorithm t_{23}.

Algorithm 7.2.1 (The TD-SpO-AVC algorithm)

---

func $t_{23}(\delta, p, v, s)$: **boolean**
  $reorder(\delta, p)$;
  $q, j, \ell := 0, 0, 0, 0$;
  $m_{[0:n)}, c_{[0:v)}, i_{[0:n)} := [0..n), [0..v), \mathtt{F}$;
  do $(q < s.len) \wedge (q \geq 0) \rightarrow$
    $tdAvc(\delta, p, m, c, i, \ell, v, j, q, s)$
  od;
  return $(q \geq 0)$
cnuf

7.2.1.2 The bounded TD-DSA-SpO algorithm

Algorithm t_{b12} corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_t(\Delta_t, d, \mathtt{T}, 0, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_t, d, 0, \mathtt{T}, \mathtt{F}, 0, 0, s) \text{ with } 0 < d < n$$

It is based on the premise that both the bounded DSA and SpO strategies are applied on the table-driven FA-based processing string algorithm. Prior knowledge of the new order of states allows for the reordering of states at the algorithm's preprocessing phase, which is followed by acceptance testing based on the bounded dynamic allocation of states in memory.

The pseudocode for the bounded TD-DSA-SpO algorithm is given below (Algorithm 7.2.2). At the start, entries of the array p are used to reorder rows of δ using the function $reorder(\delta, p)$. Then follows acceptance testing for every iteration of the main loop using the function $btddsa(\delta, p, m, d, A, Z, B, j, q, s)$ that updates the next state q to be transited to as well as the next index j of the string to be tested. The function also takes as parameters the array p that reference the actual rows in the table where states are located, as well as the various parameters $m_{[0:n)}$, d, A, B, and Z previously discussed in Chapter 4.

Algorithm 7.2.2 (The bounded TD-DSA-SpO algorithm)

func $t_{b12}(\delta, p, d, A, Z, s)$: **boolean**
  $reorder(\delta, p)$;
  $B, q, j, k, m_{[0:n)} := A, 0, 0, 0, -1$;
  do $(j < s.len() \wedge q \geq 0) \rightarrow$
    $btddsa(\delta, p, d, A, Z, B, j, q, s)$
  od;
  return $(q \geq 0)$
cnuf

7.2.1.3 The bounded TD-DSA-SpO-AVC algorithm

Algorithm t_{b123} corresponds to the formalism

$$\forall s : \mathcal{V}^* \bullet \rho_t(\Delta_t, d, \mathtt{T}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_t, d, 0, \mathtt{T}, \mathtt{F}, v, 0, s) \text{ with } 0 < d < n \wedge 0 < v < n$$

The formalism is construed to mean that the TD is implemented by combining the bounded DSA strategy with the other two strategies (SpO and AVC). It would then consists of: a preprocessing phase whereby the automaton's states are reordered according to their new position kept in the array $p_{[0:n)}$; and a processing phase where acceptance testing takes place based on both bounded DSA and AVC strategies. The bounded nature of the DSA strategy requires that when the memory block reserved for dynamic allocation is full, a replacement policy is used to copy a state currently being processed in the dynamically allocated memory space. The same principle applies for the virtual cache to be defined within the block of memory initially occupied by the states. For this algorithm, we adopt a simple policy for acceptance testing in order to establish which states are processed dynamically and which ones are processed in the cache. The policy is as follows:

- The first v $(0 < v < n)$ states in memory are considered to be in the virtual cache;
- Only the first k states $(0 < v < k < n)$, can be processed in the virtual cache: those states are said to be *cacheable*;
- States in $[k..n-k)$ are processed in a dynamically allocated memory space, that may only hold up to d $(0 < d < n)$ states.
- When allocating dynamically the $n - k$ states, if the threshold d has been reached, a replacement policy is used to swap a state out of the memory and replace it with the state currently being processed.

The algorithm below depicts the pseudocode for the bounded TD-DSA-SpO-AVC algorithm. At start, the invocation of the function $reorder(\delta, p)$ results in the states of the transition table being reordered according to the entries of the array $p_{[0:n)}$. It then follows proper acceptance testing based on either bounded DSA or AVC strategy. Therefore, when reference is made to a cacheable state q, (i.e. $q \in [0..k)$), the function $tdavc(\delta, p, m, c, i, \ell, v, k, j, q, s)$ is invoked, resulting to the new q being updated, as well as the new pointer j of the next symbol to be tested; the parameters m, c, i, ℓ of the function have already been discussed in Chapter 6. If the current state q is not cacheable, then it ought to be processed based on the bounded DSA strategy; it follows that the function $btddsa(\delta, p, d, A, Z, B, j, q, s)$ is invoked, resulting in q and j being updated. In these last two functions, information about states in the transition table are accessed via p. It should be noted that for state replacement, we may still maintain the direct mapping policy for the two functions. However, there is no dependency between the replacement policies used for bounded DSA and that used for AVC. Therefore, any policy could be used for any strategy without affecting the overall principle of the algorithm. Of course, one may choose not to adopt any replacement policy at all for performance enhancement if necessary.

Algorithm 7.2.3 (The bounded TD-DSA-SpO-AVC algorithm)

---

func $t_{b123}(\delta, p, s, v, k, d, A, Z)$: **boolean**
  $reorder(\delta, p)$;
  { $Initializations$ }
  do $(q < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow tdavc(\delta, p, m, c, i, \ell, v, j, q, s)$
    $[\!]$ $q \geq k \rightarrow btddsa(\delta, p, d, A, Z, B, q, j, s)$
    fi
  od;
  return $(q \geq 0)$
cnuf

7.2.1.4 The bounded TD-DSA-AVC algorithm

For the t_{b13} algorithm, both the bounded dynamic state allocation and the allocated virtual caching strategies are used. The algorithm corresponds to the formalism

$$\forall s : \mathcal{V}^* \bullet \rho_t(\Delta_t, d, \mathtt{F}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_t, d, 0, \mathtt{F}, \mathtt{F}, v, 0, s) \text{ with } 0 < d < n \wedge 0 < v < n$$

It is implemented by dedicating a number of cacheable states say $0 < k < n$, and a number of states that should be processed based on the DSA strategy, say $n - k$. Furthermore, the bounded nature of both strategies requires the algorithm to be provided the variables v and d that represent threshold for virtual caching and DSA respectively. Before processing a state q, a test is made as to see whether the state should be processed in the virtual cache or following the dynamic state allocation strategy; this is done by comparing q to the variable k representing the threshold of cacheable states and states to be dynamically allocated in memory. The algorithm is thus similar to algorithm t_{b123} previously described with the difference that there is no more indirect access to states information via p. The pseudocode of t_{b13} is given below.

Algorithm 7.2.4 (The bounded TD-DSA-AVC algorithm)

---

func $t_{b13}(\delta, k, v, d, A, Z, s)$: **boolean**
  { *Initializations* }
  $m_{[0:n)}, c_{[0:v)}, i_{[0:n)} := [0..n), [0..v), \texttt{F}$;
  do $(j < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow tdavc(\delta, m, c, i, \ell, v, j, q, s)$
    $[\!]$ $q \geq k \rightarrow btddsa(\delta, d, A, Z, B, q, j, s)$
    fi
  od;
  return $(q \geq 0)$
cnuf

7.2.1.5 The unbounded TD-DSA-SpO algorithm

This algorithm corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_t(\Delta_t, n, \texttt{T}, 0, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_t, n, 0, \texttt{T}, \texttt{F}, 0, 0, s)$$

It is similar to its bounded counterpart. However, unlike the bounded algorithm, no restriction is made on the number of dynamically allocated states. Therefore, during processing, any visited state that has not yet been dynamically allocated in memory would be allocated as the string is being processed. As for its counterpart, the algorithm start by a preprocessing operation that reorders the states, and access to states' information for dynamic allocation is made indirectly via the auxiliary array $p_{[0:n)}$. The pseudocode for the t_{u12} algorithm is given below.

Algorithm 7.2.5 (The unbounded TD-DSA-SpO algorithm)

---

func $t_{u12}(\delta, p, A, Z, s)$: **boolean**
  $reorder(\delta, p)$;
  $B, q, j, k, m_{[0:n)} := A, 0, 0, 0, -1$;
  do $(j < s.len() \wedge q \geq 0) \rightarrow$
    $utddsa(\delta, p, A, Z, B, j, q, s)$
  od;
  return $(q \geq 0)$
cnuf

7.2.1.6 The unbounded TD-DSA-SpO-AVC algorithm

Similar to its bounded counterpart, algorithm t_{u123} corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_t(\Delta_t, n, \mathtt{T}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_t, n, 0, \mathtt{T}, \mathtt{F}, v, 0, s) \text{ with } 0 < v < n$$

Therefore, no restriction is made on the number of states to be dynamically allocated when a state is to be processed based on the DSA strategy.

As for its bounded counterpart, the algorithm consists of a preprocessing phase whereby rows of the transition table δ are reordered according to information contained in the array $p_{[0:n)}$. Then follows proper acceptance testing whereby, when reference is made to a cacheable state, the function $tdavc(\delta, p, m, c, i, \ell, v, k, j, q, s)$ is invoked, resulting to the new q being updated, as well as the new pointer j of the next symbol to be tested. However, if reference is made to a non-cacheable state, the function $utddsa(\delta, p, A, Z, B, j, q, s)$ is used to determine the next state to be transited to, as well as the next symbol to be processed. The pseudo-code of the algorithm is given below:

Algorithm 7.2.6 (The unbounded TD-DSA-SpO-AVC algorithm)

---

func $t_{u123}(\delta, p, s, v, k, A, Z)$: **boolean**
  $reorder(\delta, p)$;
  { *Initializations* }
  do $(q < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow tdavc(\delta, p, m, c, i, \ell, v, j, q, s)$
    $[\!]$ $q \geq k \rightarrow utddsa(\delta, p, A, Z, B, q, j, s)$
    fi
  od;
  return $(q \geq 0)$
cnuf

7.2.1.7 The unbounded TD-DSA-AVC algorithm

Similar to its bounded counterpart, the algorithm corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_t(\Delta_t, n, \mathtt{F}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_t, n, 0, \mathtt{F}, \mathtt{F}, v, 0, s) \text{ with } 0 < v < n$$

It combines virtual caching and unbounded dynamic state allocation. The memory block reserved for dynamic allocation is unlimited and therefore, can hold up to the automaton's number of states. Of course since acceptance testing occurs either in the cache or in the dynamic memory block, the number of dynamically allocated states may never reach the total number of states. As for its bounded counterpart, we define a virtual cache that holds the first v states, where only the first k $(0 < v < k < n)$ states are processed in the cache. The remaining $n - k$ states are therefore, processed through dynamic allocation. The pseudo-code of algorithm t_{u13} is given below, and the variable used in the function are similar to those previously described.

Algorithm 7.2.7 (The unbounded TD-DSA-AVC algorithm)

---

func $t_{u13}(\delta, k, v, A, Z, s)$: **boolean**
  { *Initializations* }
  $m_{[0:n)}, c_{[0:v)}, i_{[0:n)} := [0..n), [0..v), \mathtt{F}$;
  do $(j < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow tdavc(\delta, m, c, i, \ell, v, j, q, s)$
    $[\!]$ $q \geq k \rightarrow utddsa(\delta, A, Z, B, q, j, s)$
    fi
  od;
  return $(q \geq 0)$
cnuf

This subsection concludes the discussions on new table-driven-based algorithms using the combination of the various strategies discussed in previous chapters. In the next subsection, we discuss the derivation of the hardcoded algorithms.

7.2.2 Derivation of the hardcoded algorithms

The derivation of the hardcoded algorithms is similar to that of the table-driven algorithms discussed in the previous subsection. The formalism of the HC characterization given in Equation 7.3 is used to derive all possible formalisms —and therefore algorithms, by instantiating arguments of the hardcoded strategies.
The strategy parameter D_h may be assigned either the value 0 to mean that there is no DSA strategy, a value d $(0 < d < n)$ to mean that DSA strategy is bounded to d states, or the value n to mean that the DSA strategy is unbounded i.e. up to n states may be dynamically allocated in memory. Therefore, $D_h \in D = \{0, d, n\}$ with $0 < d < n$. The strategy argument P_h is boolean; therefore, $P_h \in P = \{\mathtt{T}, \mathtt{F}\}$. The argument V_h may be assigned either the value 0 to mean that there is no virtual caching, or a value v $(0 < v < n)$ to mean that the AVC strategy is bounded. Therefore, $V_h \in V = \{0, v\}$ with $0 < v < n$.

It follows that up to $3 \times 2 \times 2 = 12$ different triplets of the form (D_h, P_h, V_h) corresponding to all the formalisms associated to the algorithms based on HC could be derived. Again, as for the TD algorithms, Table 7.2 depicts the range of the derived hardcoded algorithms. We follow the same naming convention by using the letter h to prefix an algorithm's name, and reference in the text of where the algorithm is discussed is provided in the fourth column of the table.

The various algorithms not previously mentioned are discussed in the subsections below:

Combination	Active strategy	Name	Reference
$(0, \mathtt{T}, 0)$	SpO	h_2	5.4
$(0, \mathtt{T}, v)$	SpO and AVC	h_{23}	7.2.2.1
$(0, \mathtt{F}, 0)$	None	h	3.2
$(0, \mathtt{F}, v)$	AVC	h_3	6.4
$(d, \mathtt{T}, 0)$	bounded DSA and SpO	h_{b12}	7.2.2.2
$(d, \mathtt{T}, v)$	bounded DSA, SpO and AVC	h_{b123}	7.2.2.3
$(d, \mathtt{F}, 0)$	bounded DSA	h_{b1}	4.3
$(d, \mathtt{F}, v)$	bounded DSA and AVC	h_{b13}	7.2.2.4
$(n, \mathtt{T}, 0)$	unbounded DSA and SpO	h_{u12}	7.2.2.5
$(n, \mathtt{T}, v)$	unbounded DSA, SpO and AVC	h_{u123}	7.2.2.7
$(n, \mathtt{F}, 0)$	unbounded DSA	h_{u1}	4.3
$(n, \mathtt{F}, v)$	unbounded DSA and AVC	h_{u13}	7.2.2.6

Table 7.2. The range of HC-based algorithms

7.2.2.1 The HC-SpO-AVC algorithm

The h_{23} algorithm corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_h(\Delta_h, 0, \mathtt{T}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, 0, \mathtt{F}, \mathtt{T}, 0, v, s);\ \text{where } 0 < v < n.$$

It is implemented using both state pre-ordering and allocated virtual caching. A simple implementation strategy was described for its TD counterpart t_{23}. The pseudocode of the algorithm is given below (Algorithm 7.2.8). At start, the function $reorder(\delta, p)$ is used to reorder the rows of the transition function according to p's entries. Then follows the generation of the hardcoded directly executable instructions using the function $hcgen(\delta, p, top)$ —previously discussed in Chapter 3— that generates hardcoded directly executable instructions starting from the address top in memory. Proper acceptance testing occurs when the function

$$hcavc(\delta, p, m, c, i, l, v, top, Z, B, j, q, s)$$

is invoked for every iteration of the main loop. This latter function updates both the next state to be transited to, as well as the index j of the next symbol to be tested. Further details on the function may be found in Chapter 6. In the function the parameter Z represents the size in bytes of a hardcoded state, and the variable B is used to calculate the address to the state to be transited to for the direct execution of the instruction at that address.

Algorithm 7.2.8 (The HC-SpO-AVC algorithm)

func $h_{23}(\delta, p, top, Z, B, v, s)$: **boolean**
  $reorder(\delta, p)$;
  $hcgen(\delta, p, top)$;
  $q, j, \ell, B := 0, 0, 0, top$;

$m_{[0:n)}, c_{[0:v)}, i_{[0:n)} := [0..n), [0..v), \mathtt{F};$
do $(q < s.len) \wedge (q \geq 0) \rightarrow$
$hcavc(\delta, p, m, c, i, \ell, v, top, Z, B, j, q, s)$
od;
return $(q \geq 0)$
cnuf

7.2.2.2 The bounded HC-DSA-SpO algorithm

Almost similar to the HC-SpO-AVC algorithm, algorithm h_{b12} corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_h(\Delta_h, d, \mathtt{T}, 0, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, d, \mathtt{F}, \mathtt{T}, 0, 0, s); \text{ where } 0 < d < n.$$

The algorithm is the combination of the bounded DSA strategy and the SpO strategy. It then consists of a preprocessing phase whereby, rows of the transition function δ are reordered according to the entries of that auxiliary array p. Acceptance testing occurs at every iteration of the main loop, whereby the function

$$bhcdsa(\delta, p, m, d, top, Z, B, j, q, s)$$

is executed following the bounded DSA strategy described in Chapter 4. The pseudocode of the algorithm is given in Algorithm 7.2.9 below.

Algorithm 7.2.9 (The bounded HC-DSA-SpO algorithm)

---

func $h_{b12}(\delta, p, top, Z, B, d, s)$: **boolean**
$reorder(\delta, p);$
$q, j, B := 0, 0, 0, top;$
$m_{[0:n)} := -1;$
do $(q < s.len) \wedge (q \geq 0) \rightarrow$
$bhcdsa(\delta, p, m, d, top, Z, B, j, q, s)$
od;
return $(q \geq 0)$
cnuf

The reordering operation enables to write directly executable instructions of each state at an address determined by the ordering policy. The bounded nature of the DSA strategy requires that before acceptance testing, the threshold of the memory used for dynamic state allocation be defined. Therefore, acceptance testing occurs only on that portion of the memory, provided that the threshold has not been reached. When the memory block reserved is full, a replacement policy is used to swap a state out of memory, replacing it with the non visited state currently being processed. An auxiliary array $m_{[0:n)}$ is used to hold the addresses of states that have already been visited for further usage. Since acceptance testing of the current symbol at a given state only occurs in the dynamic memory, when visiting a state, a test is first made to see whether it has already been dynamically allocated in memory. If that is the case, directly executable instructions referred to by the state's address in $m_{[0:n)}$ are processed. Otherwise, state's instruction are first allocated in memory —by calculating its position in the table based on the array $p_{[0:n)}$— before being executed, provided that the memory is not full.

7.2.2.3 The bounded HC-DSA-SpO-AVC algorithm

Algorithm h_{b123} consists of the combination of the bounded DSA strategy and both the SpO and the AVC strategies. It corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_h(\Delta_h, d, \mathtt{T}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, d, \mathtt{F}, \mathtt{T}, 0, v, s);$$
$$\text{where } 0 < d < n, \text{ and } 0 < v < n.$$

For its implementation, a naïve approach would be to choose both d and v such that, some states are processed in the virtual cache and others are dynamically allocated in a bounded memory block. The SpO strategy is used here at the preprocessing phase for states reordering. Once the states are reordered, a number of directly executable instructions of the states that would be processed in the virtual cache are generated. It follows that, when accessing a state, a test is made as to see whether it is a cacheable state or a DSA state[3]. If it is a cacheable state, a test is further made to see whether it is in the virtual cache or not. A failure means that, replacement should take place between the current state and a state candidate in the cache. Then follows proper acceptance testing in the cache. If the state is a DSA state, its directly executable instructions are written in memory for processing, provided that the state has not yet been visited. Otherwise, acceptance testing occurs at that state's address in memory. The fact that the dynamic memory is bounded to a threshold requires that, when the allocated memory block is full, a replacement policy is used for state swapping; the same principle applies for the states processed in the virtual cache to swap in states out of the cache. The pseudo-code of the algorithm is given below (Algorithm 7.2.10); notice that the functions and variables used are similar to those previously described.

Algorithm 7.2.10 (The bounded HC-DSA-SpO-AVC algorithm)

func $h_{b123}(\delta, p, s, v, k, d, A, Z, top)$: **boolean**
  $reorder(\delta, p)$;
  $hcgen(\delta, p, k, top)$;
  { $Initializations$ }
  do $(q < s.len \wedge q \geq 0) \rightarrow$

[3] A DSA state in this context is a state which is supposed to be processed through DSA and a cacheable state is a state meant to be processed within the allocated virtual cache.

```
        if q < k → hcavc(δ, m, c, i, ℓ, v, j, q, s)
        ▯ q ≥ k → bhcdsa(δ, p, d, A, Z, B, q, j, s)
        fi
    od;
    return (q ≥ 0)
cnuf
```

In the algorithm, the function $hcgen(\delta, p, k, top)$ only generates the first k cacheable states. The generation of the DSA states is performed implicitly within the function $bhcdsa(\delta, p, d, A, Z, B, q, j, s)$. Information of a DSA state that has not yet been visited are accessed indirectly through $p_{[0:n)}$.

7.2.2.4 The bounded HC-DSA-AVC algorithm

The h_{b13} algorithm is similar to the t_{b13} algorithm and corresponds to the formalism

$$\forall s : \mathcal{V}^* \centerdot \rho_h(\Delta_h, d, \mathtt{F}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, d, \mathtt{F}, \mathtt{F}, 0, v, s);$$
$$\text{where } 0 < d < n, \text{ and } 0 < v < n.$$

It relies on both bounded DSA and AVC strategies for acceptance testing. The algorithm is thus a variation of algorithm t_{b123} whereby the function $reorder(\delta, p)$ is not accounted for. Its pseudo-code is given below (Algorithm 7.2.11).

Algorithm 7.2.11 (The bounded HC-DSA-AVC algorithm)

```
func hb13(δ, s, v, k, d, A, Z, top) : boolean
    hcgen(δ, k, top);
    { Initializations }
    do (q < s.len ∧ q ≥ 0) →
        if q < k → hcavc(δ, m, c, i, ℓ, v, j, q, s)
        ▯ q ≥ k → bhcdsa(δ, d, A, Z, B, q, j, s)
        fi
    od;
    return (q ≥ 0)
cnuf
```

7.2.2.5 The unbounded HC-DSA-SpO algorithm

Algorithm h_{u12} corresponds to the formalism below:

$$\forall s : \mathcal{V}^* \centerdot \rho_h(\Delta_h, n, \mathtt{T}, 0, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, n, \mathtt{F}, \mathtt{T}, 0, 0, s).$$

It is a combination of the bounded DSA strategy and the SpO strategy. The algorithm is similar to h_{b12} with the difference that no threshold is required when allocating state dynamically in the reserved free memory space. The pseudo-code of the algorithm is given in Algorithm 7.2.12 below:

Algorithm 7.2.12 (The unbounded HC-DSA-SpO algorithm)

---

func $h_{u12}(\delta, p, top, Z, B, d, s)$: **boolean**
  $reorder(\delta, p)$;
  $q, j, B := 0, 0, 0, top$;
  $m_{[0:n)} := -1$;
  do $(q < s.len) \wedge (q \geq 0) \rightarrow$
    $uhcdsa(\delta, p, m, d, top, Z, B, j, q, s)$
  od;
  return $(q \geq 0)$
cnuf

7.2.2.6 The unbounded HC-DSA-AVC algorithm

The algorithm is a combination of the unbounded DSA strategy and the AVC strategy. It corresponds to the formalism

$$\forall s : \mathcal{V}^* \bullet \rho_h(\Delta_h, n, \mathtt{F}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, n, \mathtt{F}, \mathtt{F}, 0, v, s);\ \text{where } 0 < v < n.$$

The algorithm is similar to its bounded counterpart with a difference that no restriction is made on the number of states to be dynamically allocated in memory when dealing with non-cacheable states. The pseudo-code of the algorithm is given in Algorithm 7.2.13 below:

Algorithm 7.2.13 (The unbounded HC-DSA-AVC algorithm)

---

func $h_{u13}(\delta, s, v, k, d, A, Z, top)$: **boolean**
  $hcgen(\delta, k, top)$;
  $\{\ Initializations\ \}$
  do $(q < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow hcavc(\delta, m, c, i, \ell, v, j, q, s)$
    $[\!]\ \ q \geq k \rightarrow uhcdsa(\delta, A, Z, B, q, j, s)$
    fi
  od;
  return $(q \geq 0)$
cnuf

7.2.2.7 The unbounded HC-DSA-SpO-AVC algorithm

Algorithm t_{u123} corresponds to the formalism:

$$\forall s : \mathcal{V}^* \bullet \rho_h(\Delta_h, n, \mathtt{T}, v, s) \equiv \rho_{_{CDPV}}(\varnothing, \Delta_h, 0, n, \mathtt{F}, \mathtt{T}, 0, v, s);\ \text{where } 0 < v < n.$$

It is similar to its bounded counterpart (t_{b123}). Therefore, for its implementation, no restriction is made on the number of states to be dynamically allocated when dealing with non-cacheable states. Its pseudo-code is given below (Algorithm 7.2.14):

Algorithm 7.2.14 (The unbounded HC-DSA-SpO-AVC algorithm)

func $h_{u123}(\delta, p, s, v, k, A, Z, top)$: **boolean**
  $reorder(\delta, p)$;
  $hcgen(\delta, p, k, top)$;
  $\{$ *Initializations* $\}$
  do $(q < s.len \wedge q \geq 0) \rightarrow$
    if $q < k \rightarrow hcavc(\delta, m, c, i, \ell, v, j, q, s)$
    $[\!]$ $q \geq k \rightarrow uhcdsa(\delta, p, A, Z, B, q, j, s)$
    fi
  od;
  return $(q \geq 0)$
cnuf

This subsection concludes discussion on hardcoded algorithms investigated to date. In the next section we discuss the derivation of the mixed-mode algorithms.

7.2.3 Derivation of the mixed-mode algorithms

The derivation of the mixed-mode algorithms is similar to that of the TD and HC algorithms with the difference that more strategies are used in the formalisms. We have already discussed mixed-mode implementations in various forms in previous chapters. In this subsection, we only enumerate some of the algorithms derived from the general mixed-mode characterization. Discussions of new algorithms is beyond the scope of this thesis but, of importance in constructing the taxonomy as well as subsequent tools necessary for its usage.

The general formalism of the mixed-mode characterization is given in Equation 7.1. A mixed-mode recognizer is thus a function of nine variables, where three of them, namely Δ_t, Δ_h, and s are known (since the transition sets are known to be non-empty and the input string is user specific —also assumed non-empty). The variables susceptible to be used for implicit instantiation are only the strategies discussed in previous chapters. Each derived mixed-mode algorithm will be formally described using a combination of strategies in the form of 6-tuples $(D_t, D_h, P_t, P_h, V_t, V_h)$. It follows that up to $3 \times 3 \times 2 \times 2 \times 2 \times 2 = 144$ different combinations could be obtained by instantiating each strategy parameter in the 6-tuple, resulting therefore to 144 algorithms. Among the derived algorithms are those that have already been studied in previous chapters and new ones.

Recall that in previous chapters, more than one algorithm was characterized for each strategy being studied, although we only discussed one of the algorithms rather than all of them. Instead of depicting all the 144 mixed-mode algorithms, we provide in Table 7.3 some of the algorithms derived by associating each group of strategies without further combination. The last column of the table informs the reader on the section in the text where the algorithm was discussed —if at all.

The naming convention used for the algorithm is similar to that of both TD and HC algorithm, with each name prefixed with the letter m, that stands for mixed-mode. As for the previous naming convention, we assign the number 2 and 3 to the SpO strategy and the AVC strategy respectively. However the DSA strategy is no more represented with the number 1, but by the letters b (bounded DSA) and u (unbounded DSA) respectively. When a strategy is involved in a combination, the letters t (TD) and h (HC) may be used if there is a difference between the way TD and HC would be implemented. The following are some examples: m_{btuh23} represents a mixed-mode algorithm involving all the three strategies, such that the DSA strategy is bounded on TD, and unbounded on HC; m_{ut2} represents a mixed-mode algorithm that relies on both DSA and SpO strategies in which the DSA strategy is bounded on TD. m_{uh2t3h} represents a mixed-mode algorithm involving all the three strategies such that, the DSA strategy is unbounded on HC, the SpO strategy only applied on TD, and the AVC strategy only applies on HC. It is worth mentioning when a strategy involved in the algorithm applies for both HC and TD on a similar way, it is no more necessary to suffix their associated number/letter with a t or h. For example, m_{b23} involves all three strategies such that the DSA strategy is bounded on TD and HC, and the other strategy are applied on both TD and HC.

Combination	Active strategies	Name	Reference
$(0, 0, \mathtt{F}, \mathtt{F}, 0, 0)$	None	m	3.3
$(0, d_h, \mathtt{F}, \mathtt{F}, 0, 0)$	bounded DSA on HC	m_{bh}	None
$(0, \lvert\mathcal{Q}_h\rvert, \mathtt{F}, \mathtt{F}, 0, 0)$	unbounded DSA on HC	m_{hu}	None
$(d_t, 0, \mathtt{F}, \mathtt{F}, 0, 0)$	bounded DSA on TD	m_{bt}	None
$(d_t, d_h, \mathtt{F}, \mathtt{F}, 0, 0)$	bounded DSA on TD and HC	m_b	4.4
$(d_t, \lvert\mathcal{Q}_h\rvert, \mathtt{F}, \mathtt{F}, 0, 0)$	bounded TD-DSA & unbounded HC-DSA	m_{btuh}	None
$(\lvert\mathcal{Q}_t\rvert, 0, \mathtt{F}, \mathtt{F}, 0, 0)$	unbounded DSA on TD	m_{ut}	None
$(\lvert\mathcal{Q}_t\rvert, d_h, \mathtt{F}, \mathtt{F}, 0, 0)$	unbounded TD-DSA & bounded HC-DSA	m_{utbh}	None
$(\lvert\mathcal{Q}_t\rvert, \lvert\mathcal{Q}_h\rvert, \mathtt{F}, \mathtt{F}, 0, 0)$	unbounded DSA on TD and HC	m_u	None
$(0, 0, \mathtt{T}, \mathtt{T}, 0, 0)$	SpO on TD and HC	m_2	5.5
$(0, 0, \mathtt{T}, \mathtt{F}, 0, 0)$	SpO on TD	m_{2t}	None
$(0, 0, \mathtt{F}, \mathtt{T}, 0, 0)$	SpO on HC	m_{2h}	None
$(0, 0, \mathtt{F}, \mathtt{F}, 0, v_h)$	AVC on HC	m_{3h}	None
$(0, 0, \mathtt{F}, \mathtt{F}, v_t, 0)$	AVC on TD	m_{3t}	None
$(0, 0, \mathtt{F}, \mathtt{F}, v_t, v_h)$	AVC on TD and HC	m_3	6.5

Table 7.3. A range of MM algorithms

The table depicts the formalism of 15 algorithms obtained as previously described. The remaining 129 would be obtained from the various combination between the instances associated to each strategy. However, their study is left as a matter of future work.

This subsection has concluded our approach used to exploring new algorithms without relying on literature survey for taxonomy construction. It is indeed interesting to realize that, based on only three implementation strategies for FA-based string processing algorithms, up to 168 different algorithms could be generated. Of course, among the generated algorithms some may be of interest because of their efficiency, and others may be very bad in any form. In any case, the discovery of many algorithms as always been useful for research, and also for pedagogical purposes. The taxonomy construction of FA-based string processing algorithms is discussed in the next section.

7.3 The taxonomy

The previous sections were devoted to the derivations of not only existing algorithms, but also new algorithms for FA-based string processors. Some of the algorithms have been covered in the literature, whereas others are new, and require further analysis. In this section, we provide a taxonomy of FA-based string processors. We use an abstraction of the problem-domain definition to derive the core algorithms. The algorithms are further refined by adding more details (implementation strategies) in order to obtain new algorithms. The end result is presented in the form of a taxonomy tree, whose nodes represent variations of FA-based string processing. Unlike the other taxonomies in the literature our taxonomy graph is constructed in a top-down fashion using well defined refinement rules at each stage of the derivation. Each node of the algorithm corresponds to an algorithm, be it abstract or concrete.

The FA-based string processing problem can be described as *the problem of determining whether as string s is part of the language modelled by a finite automaton M or not*. For this problem, our aim is to provide several implementations approaches, by refining existing algorithms, in order to derive various algorithms whose performance metrics may be further analyzed for benchmarking. 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 existing taxonomies, the root node is often considered as a naïve solution to the problem. However, due to the nature of the problem being solved and the formalism employed in characterizing FA-based recognizers, we choose to consider our root node as 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 algorithm whose implementation could be provided. 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 our

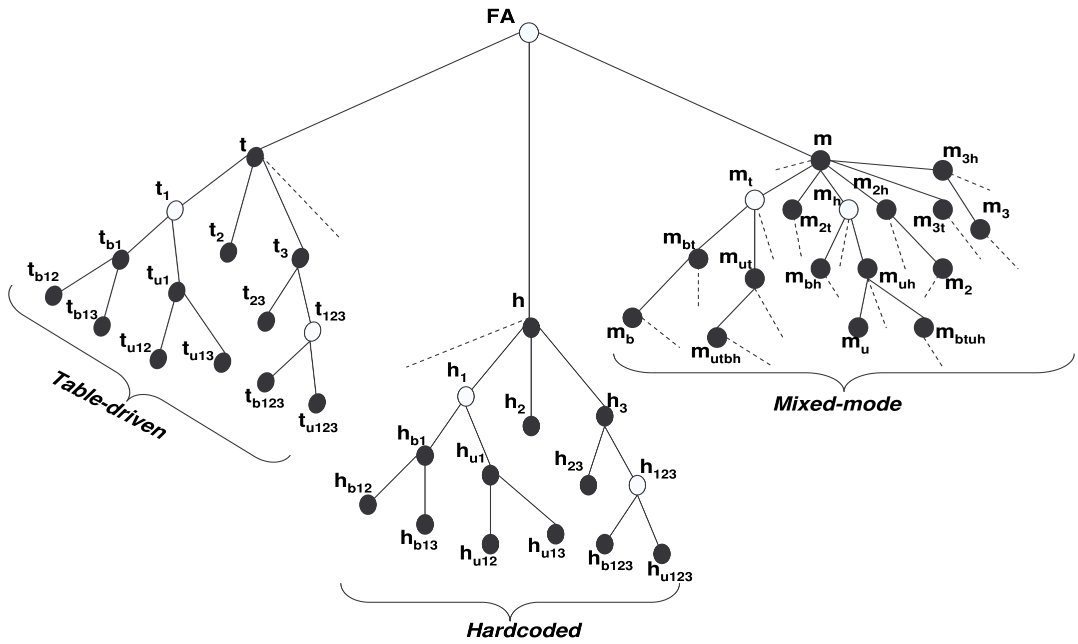

Figure 7.1. A taxonomy of FA-based String Processing Algorithms.

taxonomy, the refinement rules are the strategies applied on parents in order to obtain children.

The taxonomy graph suggested in this work is preliminary in the sense that many other implementation strategies may be suggested in order to produce several new algorithms not discussed here. Figure 7.1 depicts our taxonomy graph. The root node labelled FA represents the problem specification, more precisely that of the transition sets (table-driven and hardcoded). 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 hardcoded transition set empty, the derived algorithm is that of the table-driven (t). If provided with the table-driven transition set empty, the derived algorithm is that of the the hardcoded algorithm (h). However, if both of the transition sets are non-empty, the derived algorithm is that of the mixed-mode algorithm (m).

The three children of the root node represent the core algorithms discussed in Chapter 3. Further refinement may be used in either of the nodes at that level to produce various algorithms. Since our taxonomy is a preliminary one, many other new strategies may be applied on any of the preliminary algorithm in order to derive new algorithms not discussed in this work. This is the reason for dashed edges on various nodes in the graph.

In our taxonomy graph, the nodes in dark represent concrete algorithms, whereas the others are nondeterministic algorithms whose concrete implementations are provided by either of the children.

The characterization of FA-based implementation leads to the derivation of up to 168 different algorithms. For consistency, the taxonomy graph depicts only some of them, namely those discussed in previous sections. The overall approach used to derived algorithms 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.* The derivation of the TD algorithms is given below:

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. Notice that the algorithm could have been the child of t_2; but we have chosen deliberately to derive it from t_3. In the same way, *algorithm* t_{123} is derived from t_3 by applying simultaneously DSA and SpO strategies. Again, we could have chosen to derive it from node t_2 or t_1. Moreover, 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. The nodes were deliberately chosen to be children of t_{b1}. However, putting t_{b12} as a child of t_2, and t_{b13} as a child of t_3 makes perfect sense.

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.

In our taxonomy graph, the derivation of the hardcoded algorithms as well as the mixed-mode algorithms follows the same principles as those described for the table-driven algorithm. In effect each table-driven algorithm has its hardcoded version as given in the taxonomy graph. For example, the h_{b123} algorithm is the hardcoded version of the t_{b123} algorithm. Of course, a hardcoded counterpart of a given table-driven algorithm is not just a direct translation of that algorithm into hardcode. Although the underlying strategy used to implement the algorithm is the same, more effort should be given in hardcoding algorithms since states are made of directly executable codes and not simple data. The manipulation of instructions requires good knowledge of instruction formats, displacements calculation between addresses, and the like. This makes hardcoding of most of the children of the node h a complex task, but it can be of interest in terms of performance and hardware implementation of FA-based recognizers.

For brevity, not all the mixed-mode algorithms are given in the taxonomy graph. The scope of this thesis does not require a complete study of every single algorithm in the taxonomy, but rather, it lays down a foundation for further taxonomy study through integration of new strategies in the general formalism of FA-based recognizers, as well as analysis for most of the algorithms not discussed here. Nonetheless any algorithm derived from the combination of the basic strategies discussed in previous chapters are of interest since they are implementable. It is only after they have been implemented and tested that one can assess their importance, and decide on their applicability in relation to certain types of input strings.

7.4 Summary of the Chapter

In this chapter, we have introduced a new approach for constructing the taxonomy of a problem domain, and more precisely that of FA-based string processing algorithms using the following steps:

- Provision of a formalism of the problem domain used to represent high-level abstract solutions;
- Intuitive investigation of refinement rules applied on the abstract solution to derive more solutions;
- Combining of identified refinement rules to produce more solutions;
- Generation of the taxonomy graph made of abstract nodes and concrete ones. Each node representing an algorithm.

The derived taxonomy graph is an extensible and reusable one, made of 168 different concrete algorithms with a number of abstract algorithms that appear to be important in the literature of FA-based string processors. The taxonomy graph constructed is the starting point for toolkit design and implementation as well as a domain specific language necessary in exploiting of the toolkit. In the next chapter, we discuss the mapping of the taxonomy graph to a class diagram, that represents the architecture of our future toolkit.

CHAPTER 8

TOOLKIT DESIGN FOR FA-BASED STRING RECOGNIZERS

The aim of this chapter is to produce an architecture for an FA-based string recognizer toolkit. The architecture is based on the taxonomy tree constructed in Chapter 7. Such a toolkit may also be referred to as a *class library* or simply a *library* for FA-based string recognizers. Using the taxonomy graph, each node is mapped into a class whose attributes and operations are defined, enabling its exploitation by various application programs that rely on FA-based string recognizers. We rely on the Unified Modelling Language (UML) for the design of our class diagram. In the design process, emphasis is given on the relationships between classes, and to some extent, to the specification of class *attributes* and *operations*. The complete implementation of the toolkit is beyond the scope of this thesis. Instead, for each class in the systems, we briefly provide guidelines for implementing some of the operations and suggest suitable datatypes that may be associated with attributes. Such implementation guidelines point to the way in which a working toolkit system could be exploited by application programs.

The chapter starts with some introductory notes on toolkit design and implementation, as well as a brief survey of some related work in the literature. Then follows the architectural description of the toolkit, using a top-down design approach: i.e. a high-level view of the system is first suggested, followed by detailed diagrams representing various parts of the system. When detailing system's parts, the nature of the association between classes is given as well as a description of class attributes and operations. A detailed view of the system provided towards the end of the chapter indicates how the system may be exploited by external application programs.

8.1 Motivation and Related Work

The toolkit envisaged here is a self contained package of implemented and directly executable 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 toolkit, since such an application typically needs to test whether a given string pattern is part of the language modelled by a well specified automaton. Moreover, the toolkit may also be used for educational and research purposes by supporting experimentation and benchmarking of the various algorithms.

Class libraries for client applications are widely used in software construction. Most general purpose programming languages provide libraries to be used to produce software [VM03, Ale01]. In principal, class libraries could be provided for various problem domains such as FA-based string processing, pattern matching, sorting, and the like. In fact, toolkits for the symbolic computation of finite automata do exist, including the following:

- The **Grail** system [RP93]. 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 [JPTW90]. 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 [CH91]. 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** [Wat94]. It is an implementation of all the algorithms that appear in the taxonomy or regular expression algorithms [Wat95b]. A somewhat smaller version referred to as **FIRE Lite** is proposed in [Wat95b]. The aim of **FIRE lite** was to provide a variety of algorithms to the user that 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 [WC04] 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.

- The **SPARE Time** system [CW05] is a string pattern recognition toolkit design in C++ using the Taxonomy-based software construction (TABASCO) approach.

- The **FIRE Station** system [Fri05] is a finite automata and regular expression utility for various purposes such as automata minimization, regular expression rewriting, and various other transformations.

Here, an architectural design of a toolkit is proposed, instead of a complete ready to use package as suggested by the above packages in the literature. As already mentioned, its implementation is left for future work. This chapter focuses on the specifications and design of most of the toolkit's components, and more precisely on its classes, as well as important class attributes and operations. Although **SPARE**

Parts is a complete package consisting of the implementation of various pattern matching algorithms available in the literature with the possibility of being invoked by external applications, many existing toolkits merely provide for symbolic computation of automata rather than for the actual practical processing of string for various computational needs. Our proposed toolkit differs from the others in that classes correspond to the various implementation strategies discussed in previous chapters, with the single intention of providing for acceptance testing, rather than for automata transformations or operations on them. It is thus clear that, any FA-based string processing problem whose solution may be modelled by a transition set and a given input string could exploit our toolkit for acceptance testing.

The next section depicts the mapping from the taxonomy tree to a class diagram along with the definition of classes and class components in the process.

8.2 The Architectural Design

In this section, we depict the toolkit's architecture in a top-down fashion. That is, we first provide a high level view of the architecture and systematically discuss the structure diagram of each of the components present in the high-level diagram. The detailed architecture of each component is further combined to reflect a more detailed view of the system, which is considered as our toolkit's detailed class-diagram. In the process of depicting various views of the systems, components such as classes, class attributes and operations are described, as well as the relationships that hold in regard to the interaction between various classes in the the system.

At a higher level, a toolkit may be regarded as a set of interacting packages which in turn are made of interacting classes that represent *self contained algorithms* used for string processing. By self-contained algorithms, we mean algorithms that may be regarded as objects, requiring in the process of instantiating them, proper construction based on their attributes and proper invocation using directly executable definition of their main operation used for acceptance testing.

The taxonomy in Chapter 7 is a tree structure whose root represents a simple specification of an FA-based string recognizer. The root node of the taxonomy graph was further refined, resulting in three children at the first level of the tree, corresponding to the table-driven, the hardcoded, and the mixed-mode algorithms respectively. At a level down the hierarchy, were the children of the core TD, HC and MM algorithms, corresponding to various implementation strategies. It follows that a toolkit could be regarded at higher level as a system consisting of the following four interacting packages:

- The `PkgRecognizer`, consisting of the whole problem domain specification; that is the transition sets and the input string. The package interacts with:
- the `PkgTableDriver` which embodies the various table-driven algorithms that were obtained by using the various implementation strategies to modify the core table-driven algorithm;

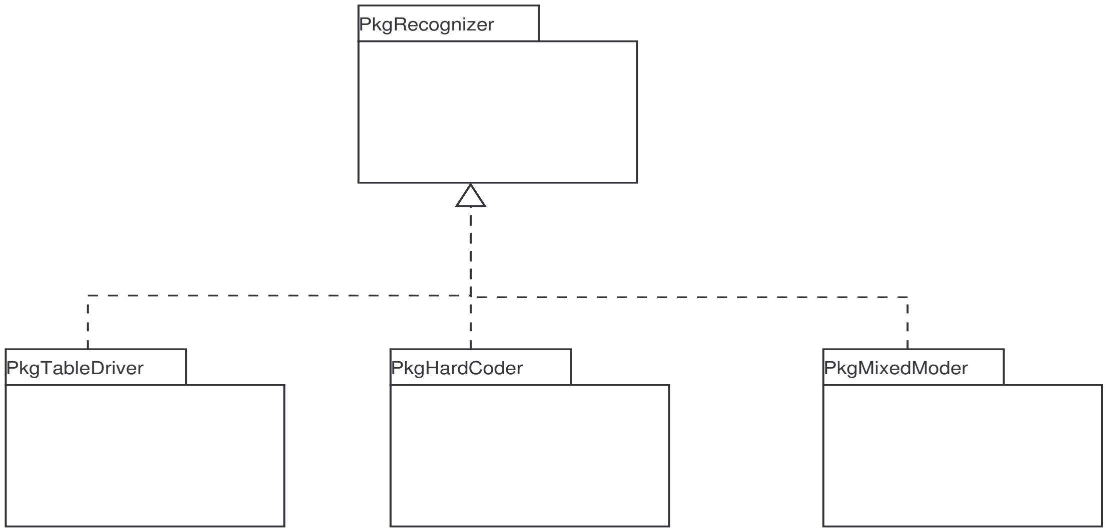

Figure 8.1. A high-level toolkit's view based on interacting packages.

- the `PkgHardCoder` which consists of the various hardcoded algorithms that are derivatives of the original hardcoded algorithm; and
- the `PkgMixedModer`: consisting of the algorithms that are derivatives of the mixed-mode core algorithm characterised by various combinations of the constraints.

Figure 8.1 depicts such a high-level view of the toolkit. There is a *dependency* relationship between the *root package* and its children; which literally means that the packages down the hierarchy are sub-packages of `PkgRecognizer`. Therefore any class in any of the sub-packages inherits from a base class in the the root package. However, there will be only one class (that we shall refer to it 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 emphasise 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 of a root class representing the specification of the problem domain, from which any other classes down the hierarchy inherit. The subsections below elaborate on each package of the system, discussing in the process 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.

8.2.1 The Package PkgRecognizer

Figure 8.2 depicts the class-diagram that make up `PkgRecognizer`. The package consists of the following classes: `Recognizer`, `State`, `Transition` and `AlphabetObject`.

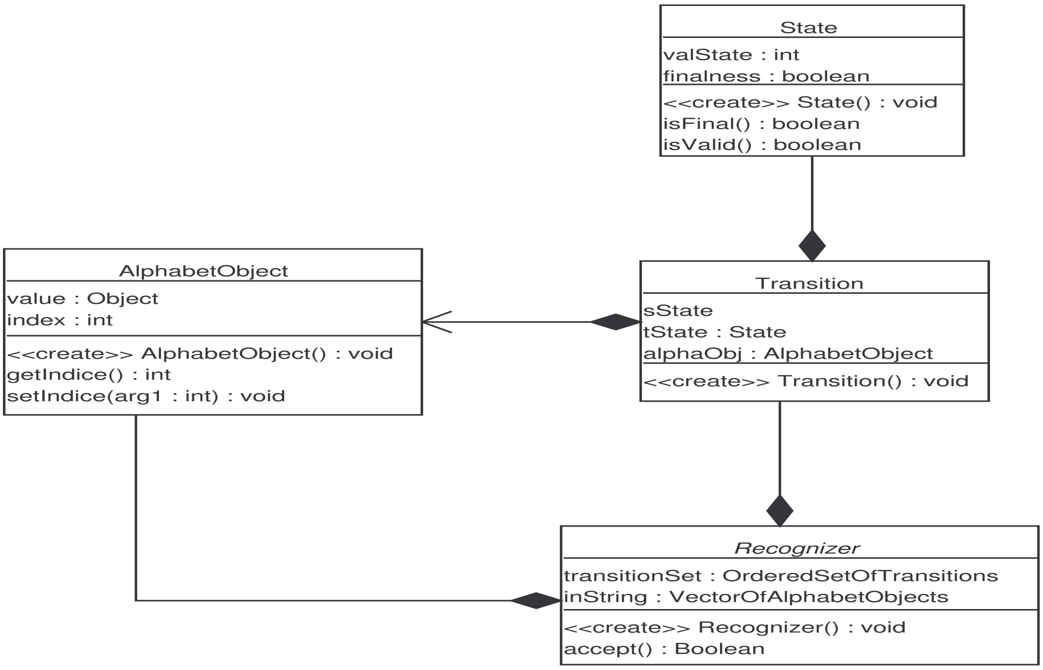

Figure 8.2. The class diagrams of `PkgRecognizer`.

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 inputs 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*(). Further specifications need to be considered for the class's attributes since they are not simple datatypes.

The attribute *transitionSet* of the class `Recognizer` may be regarded as an ordered set of transitions. Using a set to represent the transitions guarantees that there are no duplicate elements. We also require the set to be ordered for easy information retrieval based on sequential or direct access. A transition is a triple 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. Furthermore, the $finalness$ attribute is of no use for a state which is *a priori* a rejecting state. 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. A constructor, a copy constructors, and a destructor are required for `AlphabetObject`. 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 allows 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 lack of explicit definition of the implementation strategy to be used for acceptance testing makes the operation *accept*() virtual (like in C++ for example). Therefore, the class `Recognizer` is just an abstract class and cannot be instantiated. Explicit definitions of implementation strategies are provided further down the hierarchy of the toolkit. However, the class `Recognizer` is regarded as the root class from which major classes down the hierarchy are derived. The other classes in the package only contribute to the base class attributes. Various other 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. Those operations are useful in ascertaining that the construction of objects down the hierarchy are well defined. The next section depicts the contents of the table-driven package.

8.2.2 The Package PkgTableDriver

The `TableDriver` class in a package also called `PkgTableDriver` inherits from the class `Recognizer` in the `PkgRecognizer` package. The package consists of a hierarchy of classes whose base class, `TableDriver`, implements the core table-driven algorithm in the operation *accept*(). The class inherits all attributes of `Recognizer` necessary for its instantiation. An additional attribute *tdNumStates* which is an integer that holds the number of states for a table-driven implementation is used for consistency checking during instantiation of a `TableDriver` object. Such consistency checking enables one to ensure that the number of states provided in the transition set matches with the value of the attribute *tdNumStates*.

In order to do so, an operation such as *assert*() that compares *tdNumStates* and the number of states in the transition set is invoked before construction of an object of type `TableDriver`.

For example, after constructing an instance of `TableDriver` say `TD`, that generates the table-driven recognizer, we simply use the statement `TD.accept()` which returns a boolean to test whether the input string is part of the language modelled by the FA or not. Figure 8.3 depicts the class diagram contained in the table-driven package.

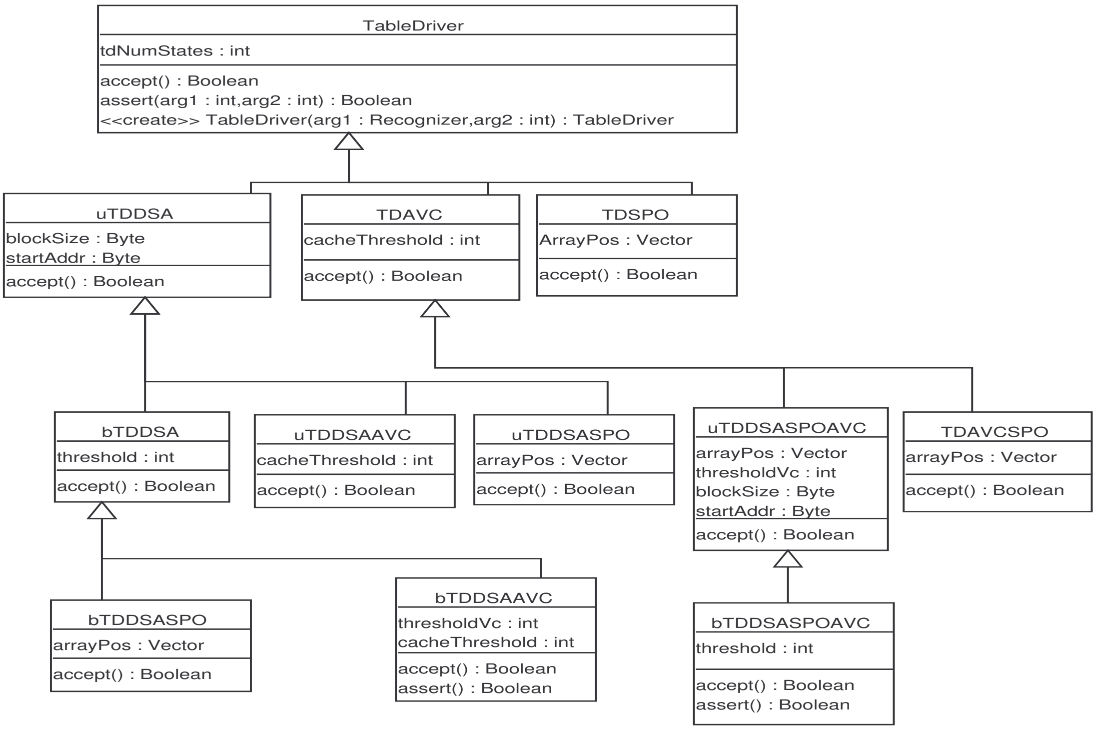

Figure 8.3. The `TableDriver` class diagram.

The diagram was obtained by mapping the nodes of the table-driven part of the taxonomy graph to classes. The `TableDriver` class is the base class in the package, and all other classes directly on indirectly inherit from it. As shown in the figure, classes are given in three levels of the hierarchy. Each level of the hierarchy will now be discussed in the subsections to follow.

8.2.2.1 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 is a form of specialization of the `TableDriver` class that inherits all attributes of table-driven (as well as those of `Recognizer` indirectly) but also requires an additional attribute *cacheThreshold* that ensures its specialization. The unique attribute of `TDAVC` supports input string processing that is based on the allocated virtual caching strategy. The threshold holds an integer value that specifies the last state that falls within the virtual cache starting from state 0. Therefore, 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 such as $assert()$ is thus required in the class for checking the validity of the fundamental condition of AVC. Once the condition is satisfied, the operation $accept()$ could be invoked for acceptance testing.

The class `TDSPO` holds the implementation of a table-driven recognizer based on 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*. The operation $accept()$ could then be invoked for acceptance testing. As for the `TDAVC` class, an operation such as $assert()$ may be required to ensure that the new positions of the states have indeed been provided in $arrayPos$.

Unlike the classes `TDSPO` and `TDAVC` that were obtained by mapping the concrete nodes t_2 and t_3 of the taxonomy graph (see Chapter 7, Figure 7.1) to concrete classes, the node t_1 represents an abstract class which is nondeterministic, representing the DSA strategy in a broad sense. The t_1 node is concretely represented by either the t_{b1} node or the t_{u1} node in the taxonomy graph, representing the bounded and unbounded DSA strategies respectively. It follows that for simplicity in the class diagram, the node t_1 should be removed and replaced by either of its child node. In order to decide whether both concrete nodes may directly inherit from the node t, or only one of them is suitable to inherit from t, we analyze the attributes of their associated classes.

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 as a derived class of `TableDriver` in the absence of the abstract class `TDDSA`. For the construction of an instance of `uTDDSA`, an operation $assert()$ 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 *statAddr* is a valid memory address or not. Once the $assert()$ is satisfied, the operation $accept()$ that returns a boolean could be invoked for acceptance testing.

8.2.2.2 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`.

As mentioned earlier, 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 based class. The attribute holds the maximum number of states to be dynamically allocated. It follows that an operation $assert()$ 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. When all the necessary conditions are satisfied, the operation $accept()$, returning a boolean for acceptance testing may be invoked.

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 reflects multiple inheritance. However, it can either be considered as a specialization of `uTDDSA` or that 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 chose 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 *assert*(). The operation *accept*() is used to invoke the generated table-driven algorithm based on both unbounded DSA and state pre-ordering.

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 on 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 *assert*() 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 $arrayPos \neq \varnothing$, and 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 operation *accept*() would of course then be used for acceptance testing.

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 *assert*() operation is used at construction time to ensure that the array is indeed provided. The operation *accept*() is used to invoke the table-driven algorithm based on both SPO and AVC strategies.

8.2.2.3 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`, equally as we might have chosen to make it a subclass of `TDSPO`, or as deriving form 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. An operation *assert*() is required to ensure that *arrayPos* is provided before constructing an instance of the class. The directly executable table-driven algorithm based on both bounded DSA and SPO strategy may be generated at construction time. The operation *accept*(), that returns a boolean, is used to test whether the string is part of the language modelled by the automaton.

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. In order to ascertain that the condition in which both bounded DSA and AVC strategy may be used is satisfied, an operation *assert*() is used to check whether the condition $0 < cacheThreshold < thresholdVc < tdNumStates$ holds. The bounded nature of the class requires that replacement could also be performed during dynamic allocation of states. Again, the method *accept*() is used for acceptance testing.

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. The operation *accept*() is used to test whether the input string is part of the language modelled by the FA, by invoking the generated bounded table-driven DSA-SPO-AVC algorithm.

For each class discussed in the table-driven package, there is a hardcoded counterpart. We briefly discuss the `pkgHardCoder` package in the next subsection.

8.2.3 The Package PkgHardCoder

The **pkgHardCoder** package is made of a class diagram which is a hierarchical structure in which the main relationships between classes is that of inheritance.

On top of the hierarchy, is the root class `HardCoder` from which any other class down the hierarchy inherits directly or indirectly. The class in turn is a kind of recognizer. Therefore in the whole toolkit class diagram, it is derived class of the class `Recognizer`. The inheriting class adds the attribute *hcNumStates* that holds the total number of hardcoded states to be constructed. An operation $assert()$ is used to check whether the total number of hardcoded states is equal to that of the states in the automaton transition set provided while constructing the recognizer. If that is the case, acceptance testing may be performed by invoking the method *accept*().

Figure 8.4 depicts the class diagram obtained by mapping the hardcoded part of the taxonomy tree onto classes. The diagram is similar to the that of the table-driven class diagram with the only difference being the change of class names and of course, the way in each acceptance testing takes place (now based on instructions instead of data).

At the first level of the `HardCoder` class-diagram are the classes `uHCDSA`, `HCAVC`, and `HCSPO` that correspond to the nodes h_{u1}, h_3, and h_2 of the taxonomy graph respectively. As for the table-driven package, we have removed the abstract class `HCDSA` from the diagram. The class has been replaced by `uHCDSA` which indeed inherits from `HardCoder`, and holds additional attributes *blockSize*, and *startAddr* described in the previous subsection.

The second level of the `HardCoder` class diagram comprises the classes `bHCDSA`, `uHCDSAAVC`, `uHCDSASPO`, `uHCDSASPOAVC`, and `HCAVCSPO`, that correspond to the nodes

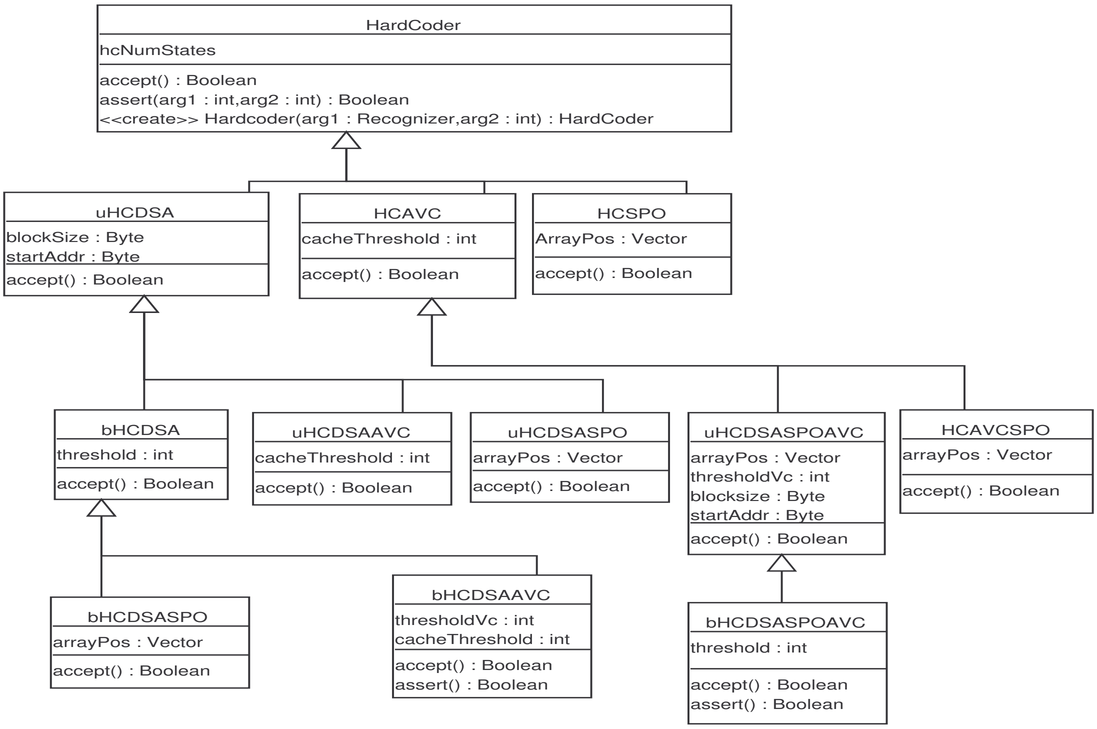

Figure 8.4. The `HardCoder` class diagram.

h_{b1}, h_{u13}, h_{u12}, h_{u123}, and h_{23} of the taxonomy graph respectively. The first three classes are subclasses of `uHCDSA` and the remaining two inherit from `HCAVC`. As explained in the previous subsection, we have chosen to conform with single inheritance as opposed to multiple inheritance. Again, we could have chosen to make `HCAVCSPO` a subclass of `HCSPO`, and `uHCDSASPOAVC` a subclass of any of the class at the first level of the hierarchy, provided that changes are made on the attributes of the subclasses according to the attributes of their base classes.

At the last level of the hierarchy, are the classes `bHCDSASPO`, `bHCDSAAVC`, and `uHCDSASPOAVC`. The first two classes inherits from `bHCDSA` and the last one is a subclass of `uHCDSASPOAVC` corresponding to the nodes h_{b13}, h_{b13}, and h_{b123} of the taxonomy tree respectively. We could have made the last class a subclass of any class at the second level of the hierarchy, except the `bHCDSAAVC` class. The same applies for the first two classes which could have be subclasses of `HCSPO` (for `bHCDSASPO`), and `HCAVC` (for `bHCDSAAVC`).

A discussion of the mixed-mode package follows in the next subsection.

8.2.4 The Package PkgMixedModer

The package `PkgMixedModer` comprises the `MixedModer` class diagram containing implementations of the mixed-mode algorithms. In our taxonomy graph only a portion of the mixed-mode nodes are given. This subsection discusses the mapping of those nodes into concrete classes.

The interaction of the PkgMixedModer package and the recognizer package is materialized by an inheritance relationship between the main class MixedModer in PkgMixedModer, and the main class Recognizer in PkgRecognizer. A mixed-mode algorithm is a string recognizer; therefore, the class MixedModer is regarded as a subclass of Recognizer. In order to instantiate an object of the class MixedModer, the following attributes are required: *tdNumStates* that holds the total number of table-driven states, and *hcNumstates* that holds the total number of hardcoded states. At construction time, an operation *assert*() is required to ascertain that the sum of the hardcoded states and the table-driven states is equal to the total number of the states in the automaton's transition set. Acceptance testing takes place by invoking the method *accept*() that returns a boolean. The MixedModer class that corresponds to the node m of the taxonomy graph is considered as the base class in the mixed-mode package. Any class within the diagram inherits from it directly or indirectly.

Figure 8.5 depicts the class-diagram for the mixed-mode algorithms. The diagram is structured in the form of a hierarchical structure consisting of three levels below the base class. The class diagram was obtained by mapping the taxonomy's nodes into classes, and preserving the inheritance relationships in the process.

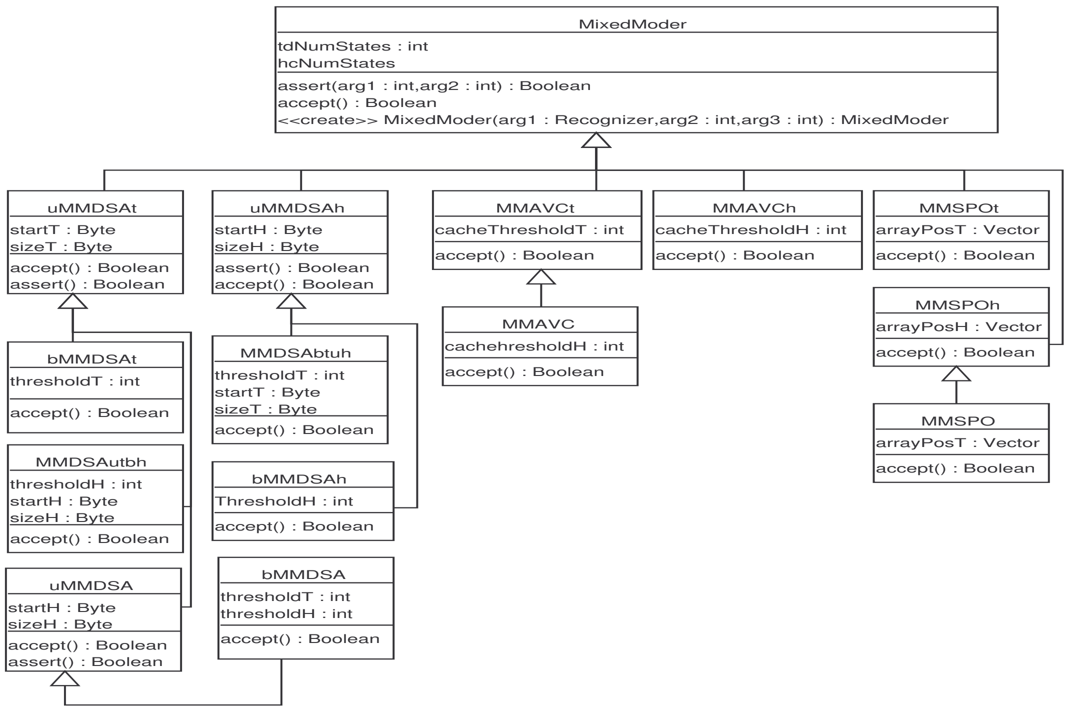

Figure 8.5. An extract MixedModer class diagram.

At the first level of the hierarchy, are the classes uMMDSAt, uMMDSAh, MMAVCt, MMAVCh, MMSPOt, and MMSPOh. These classes respectively correspond to the nodes m_{ut}, m_{uh}, m_{3t}, m_{3h}, m_{2t}, and m_{2h}. As we may observe, during the mapping, the abstract nodes m_t/m_h were removed in the class diagram and replaced by its children

(m_{ut} and m_{uh}). An explanation to this is similar to the one provided when discussing the previous packages. It makes sense to have the unbounded-DSA mixed-mode classes inheriting directly from the base class because it reflects inheritance, and the other DSA subclasses down the hierarchy are regarded as specialized unbounded DSA classes. The same applies for the remaining four classes at this level, whereby, their respective parent nodes (m_3 and m_2) have been replaced with the nodes representing the partial algorithms (AVC on HC/TD and SpO on HC/TD). The `MMAVCt` is partial in the sense that virtual cache allocation only happens on the table-driven states. The partiality of the `MMAVCh` class is justified by the fact that virtual caching only occurs on the hardcoded states. It would then be better to have the two classes as subclasses of `MixedModer` rather than the total `MMAVC` class. The same applies for both `MMSPOt` and `MMSPOh` classes, now considered as subclasses of `MixedModer` rather than the class `MMSPO` which implements the total SpO algorithm. In general attributes used for each class are the same as those used for the `HardCoder` and `TableDriver` classes. When necessary, if the the mixed-mode class relies partially on either of the algorithms, only the attribute(s) associated to the algorithm under consideration is used. The operations, *assert*() and *accept*() are used for attribute(s) validation and acceptance testing respectively.

At the second level of the hierarchy, are the classes `bMMDSAt`, `MMDSAutbh`, `uMMDSA`, `MMDSAbtuh`, `bMMDSAh`, `MMAVC`, and `MMSPO` corresponding to the following respective nodes in the taxonomy tree: m_{bt}, m_{utbh}, m_u, m_{btuh}, m_{bh}, m_3, and m_2. The class `MMAVC` may inherit from either of the AVC classes at the first level of the hierarchy. It can also holds a multiple inheritance relationship with the two AVC parent classes. However, we have chosen to have it derived from the `MMAVCt` class. As a result, the attribute *cacheThresholdH* that holds the threshold of the virtual cache for the hardcoded states is required in the subclass. An *assert*() operation is required in order to check the validity of the class' attribute before acceptance testing could take place using the operation *accept*().

The class `MMSPO` could also be a subclass of either of the SpO classes in the first level of the hierarchy. We chose to have it as a specialized class of `MMSPOh`. Therefore, the attribute *arrayPosT* that holds the new positions of the table-driven states to be reordered is required in the class. Both *assert*() and *accept*() are required for attribute validity checking and acceptance testing respectively.

The class `bMMDSAt` (respectively `bMMDSAh`) inherits from the class `uMMDSAt` (respectively `uMMDSAh`). Its attribute *thresholdT* (respectively *thresholdH*) holds the threshold of the number of states to be dynamically allocated. It requires the operations *assert*() and *accept*() for validity checking and acceptance testing.

The class `uMMDSA` was made a subclass of the class `uMMDSAt`. It could have also been made subclass of `uMMDSAh`. For this reason, the attributes *startH* and *sizeH* are used to reference the start address of the hardcoded states, as well as the size of the memory block reserved for dynamic state allocation. The validity of the attributes' values is performed at construction time using the operation *assert*(). Acceptance testing is performed using the operation *accept*() that returns a boolean.

The class `MMDSAutbh` is a specialized class of `uMMDSAt`. It holds the implementation of the unbounded DSA on TD, and also holds the bounded DSA on HC. The attributes

thresholdH, *startH*, and *sizeH* are required to characterize the bounded dynamic allocation of the hardcoded states. As for the other classes, *assert*() and *accept*() are equally important for this class.
The class `MMDSAbtuh` is a subclass of the `uMMDSAh`. It holds the implementation of the unbounded DSA on HC with a bounded DSA on TD. The attributes *thresholdT*, *startT*, and *sizeH* are used to characterize the bounded TD DSA part of the algorithm. The operations *assert*() and *accept*() are necessary for validity checking and acceptance testing.

The last level of the hierarchy only contains the class `bMMDSA` which corresponds to the node m_b. It is derived from the class `uMMDSA`. Its attributes *thresholdT* and *thresholdH* hold the threshold of the number of states to be dynamically allocated for table-driven and hardcoded respectively. As for the previous classes, the operations *assert*() and *accept*() are used for validity checking and acceptance testing.

Having discussed each of the package, we may now represent the class diagram as a whole, depicting all inheritances and composition relationships previously discussed; this is addressed in the next subsection.

8.2.5 A Detailed Toolkit's Architecture

The architecture of the toolkit is composed of a set of interacting classes. The relationships in the class-diagram are mainly composition and inheritance. We used composition to depict those of the classes containing other classes, and inheritance was mainly used to show the derivation from one class to the other. Figure 8.6 depicts the overall architectural view of the toolkit derived from the taxonomy tree constructed in the previous chapter. Assuming that the toolkit is a working system, the following process can be followed for the invocation of any of the algorithms within the diagram:

1. **Problem domain specification** that entails constructing the input string as well as the transition set to be used for the partial construction of the recognizer;

2. **Choice of the algorithm** that requires the implementer to decide whether to rely on one of the core algorithms or on one of its derived subclasses instead;

3. **Construction of an instance of the chosen algorithm** that entails providing the class constructor with all information required. Of course in the construction process, it should be noticed that when the instance being constructed is further down the hierarchy, all its parent classes should be provided with appropriate attributes, especially those from which the class being instantiated inherits (directly or indirectly) from;

4. **Generation of the corresponding algorithm**. In effect, each class containing a concrete operation *accept*() must be equipped with a generator; that is, a function that generates either of the algorithm of concern provided that all information required is available. We have not explicitly mentioned the presence of the function when describing classes, the reason being that it may be embedded in the operation *accept*() depending on the taxonomy implementer's taste.

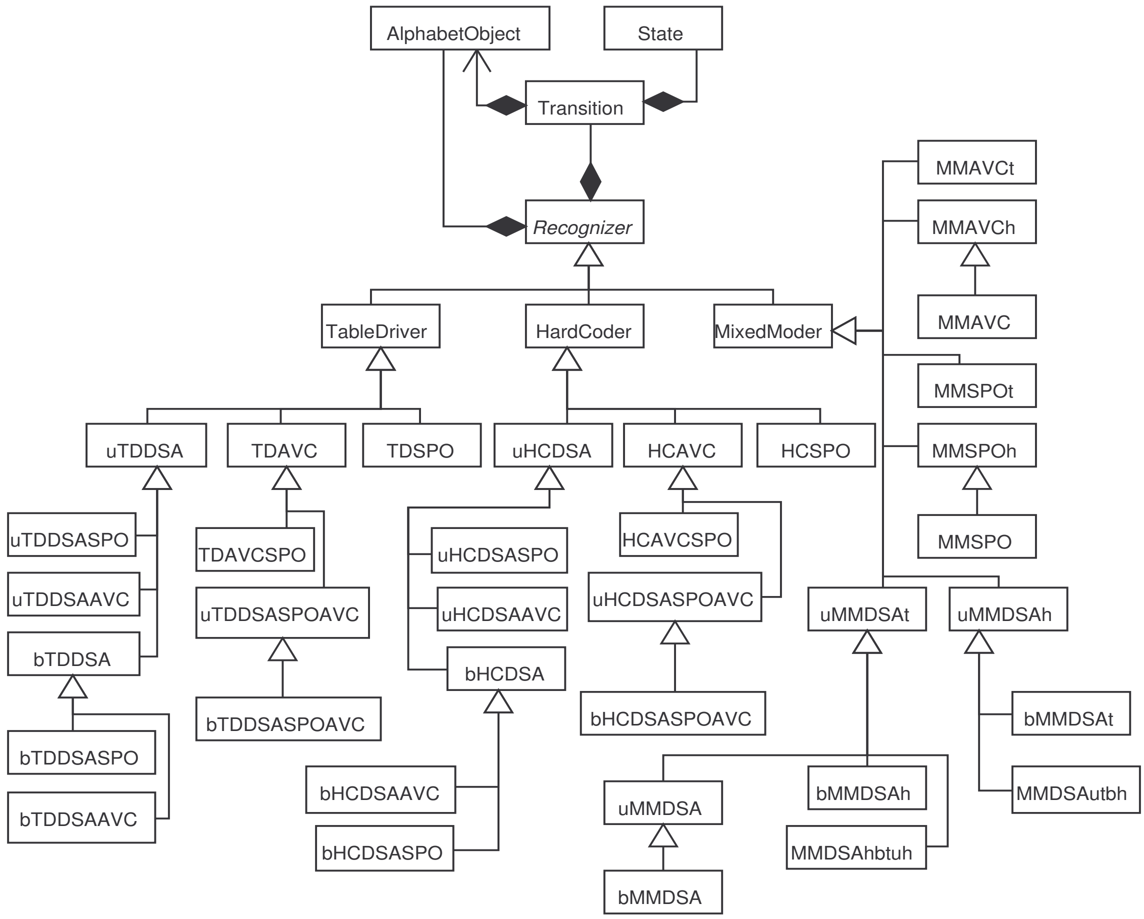

Figure 8.6. An extract FA-based String Recognizers class-diagram.

However, putting it in the operation may constitute a performance bottleneck although it may not be a complex task to implement such a generator;

5. **Perform acceptance testing** which only requires the invocation of the function *accept*() that returns a boolean.

The process above is used under the assumption that the toolkit user is not only familiar with the algorithms, but also knows how to instantiate the classes and understands their behaviour. Therefore, unexperienced users may not find the toolkit user friendly because of the underlying technicalities required for its efficient usage. In order to address such issue, we may write on top of the toolkit a *little language* commonly referred to as Domain Specific Language(DSL) that assists the user to combine all the above mentioned steps into one. Since this chapter only dealt with the design of the toolkit, the design and implementation of a DSL is also a matter of future work.

8.3 Summary of the Chapter

In this chapter, we have proposed an architectural design of a toolkit for FA-based string recognizers by mapping when necessary, each node of the taxonomy graph to a class. In the process, the relationships between nodes were further refined in order to facilitate inheritance relationships between classes. The class-diagram produced is incomplete since many more classes, especially those pertaining to the mixed-mode package, could have been added as suggested during taxonomy construction. With an architectural design provided, the implementation programming language is a matter of choice. In effect, any object-oriented programming language is suitable for the implementation of the toolkit. However, the most cumbersome task would be that of generating the different algorithm for acceptance testing within each class. In previous experiments, most of the algorithms were implemented in assembly language. Therefore, no matter the programming language used for the implementation of the toolkit, provision should be given for the generation of directly executable recognizers (i.e. the operation *accept*() in each class) that has already present all information on the transition function (hardcoded, tabled-driven or mixed-mode) as well as information on the string. Previous experiments also revealed that hardcoded recognizers in high-level language are very inefficient. It would then be better to encode the operation *accept*() in all classes in assembler and obtain its direct executable code.

This chapter concluded the part on characterization of FA-based recognizers. In this part, the unified formalism used for the characterization of the core algorithms leaded to a more general characterization of the recognizers based on each of the core algorithms. Various implementation strategies (constraints) investigated applied on each of the implementation strategies resulted to new algorithms. Further investigations revealed that the identified constraints could be instantiated and the resulting instances were combined together, leading to the derivation of new formalisms —and therefore new algorithms. Using the algorithms, a taxonomy graph was constructed based on the relationship between the nodes (algorithms). In the constructed taxonomy the main refinement rule used was that of constraint integration and constraint combination. The nodes in the taxonomy were further mapped into classes, and in the process, we produced an architectural view of a system for FA-based string processing commonly referred to as toolkit. Although the actual implementation of the toolkit is beyond the scope of this thesis, in the next part, we study the performance of some selected algorithms in order to capture the extend to which some algorithms outperform their core counterparts.