kBehavior: Algorithms and Complexity

A Compendium to [1]

Leonardo Mariani[†] and Fabrizio Pastore[‡] and Mauro Pezzè[†‡]

[†]University of Milano Bicocca,
Department of Informatics, Systems and Communication,
viale Sarca 336, 20126 Milano, Italy

[‡]University of Lugano,
Faculty of Informatics,
via Buffi, 13 6900 Lugano, Switzerland

mariani@disco.unimib.it, fabrizio.pastore@usi.ch, mauro.pezze@usi.ch

1 Introduction

Different software testing and analysis techniques use finite state automata (FSA) inferred from execution traces to model the behavior of a software system and help software engineers to re-engineer [2], debug [3, 4], or identify failures and anomalies in the software [5–7]. FSA inference algorithms generate FSA that accept and generalize strings of symbols received as input [8–17].

We described the kBehavior inference engine in [1]. kBehavior has been successfully adopted to infer FSAs that model the behavior of different software systems. In particular, it has been successfully applied to derive FSAs from both traces collected by monitoring the activity of software components [1, 6, 18] and standard log files produced by Enterprise systems [19–21].

This report analyzes kBehavior complexity and complements the algorithms description in [1]. The report proceeds as follow: Section 2 recalls some background on Finite State Automata, Section 3 overviews the kBehavior algorithm, Section 4 presents the analysis of kBehavior complexity, Section 4 concludes the report.

2 Finite State Automata

This Section recalls some definitions required to understand the kBehavior algorithm and calculate its complexity.

A *Non-Deterministic Finite State Automaton* is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a non-empty finite set of states,
- Σ is a non-empty finite set of input symbols or input alphabet,
- $\delta : Q \times \Sigma \rightarrow \wp(Q)$ is the transition function that maps pairs of ⟨state, symbol⟩ into subsets of states,
- $q_0 \in Q$ is the initial state,
- $F \subseteq Q$ is the set of accepting states

If the function δ maps every pair ⟨state, symbol⟩ to a single state, that is $\delta : Q \times \Sigma \rightarrow Q$, the automaton is *deterministic*. Σ^* denotes the set of all input strings over Σ. Given $\alpha \in \Sigma^*$, $|\alpha|$ denotes the length of α. Given $\alpha = \beta\gamma \in \Sigma^*$, we say that β is a *prefix* and γ is a *suffix* of α. δ^* denotes the straightforward extension of the transition function δ to strings:
$\delta^*(q, \epsilon) = q$ where ϵ is the empty string,
$\delta^*(q, a\alpha) = \delta^*(\delta(q, a), \alpha)$ where $q \in Q$, $a \in \Sigma$, and $\alpha \in \Sigma^*$.

An input string α is *accepted* by a finite state automaton (either deterministic or not) if $\delta^*(q_0, \alpha) \cap F \neq \emptyset$. $L(A)$ denotes the *language* accepted by a FSA A, that is, the set of strings accepted by A.

3 Overview of kBehavior

kBehavior is an inference engine that given a set of strings S, builds a finite state automaton A, whose language $L(A)$ includes S.

The algorithms that implement kBehavior are illustrated in the following boxes: Algorithm 1 shows the main algorithm that extends an inferred automaton to accept all the input strings one after the other; Algorithm 2 shows *extendFSA* the algorithm that initializes the automaton in case it is empty, Algorithm 3 presents *addString* which extends the given automaton to accept a string; finally Algorithms 4, 5 and 6 define the *pattern*, *merge* and *addTail* functions that are used in *extendFSA* and *addString*.

Algorithm 1 FSA kBehavior(S, k, k_ρ)

Require: a non-empty set of strings S that are processed by the algorithm
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ or $k_\rho \geq k$
Ensure: an FSA A that accepts all strings in S: $S \subseteq L(A)$
1: $A \leftarrow \varnothing$
2: **for all** $s \in S$ **do**
3: $A \leftarrow$ extendFSA(A, s, k, k_ρ) *//extend A by incrementally adding the string s*
4: **end for**
5: **return** A

Algorithm 2 FSA extendFSA(A, s, k, k_ρ)

Require: an automaton A with initial state q_0
Require: a string $s = \langle s_1, \ldots, s_n \rangle$
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ or $k_\rho \geq k$
Ensure: an automaton A' that accepts both the language accepted by A and s: $s \cup L(A) \subseteq L(A')$
1: **if** $A == \varnothing$ **and** s empty **then**
2: **return** an automaton with 2 states connected by an ϵ transition
3: **end if**
4: **if** $A == \varnothing$ **then**
5: $l \leftarrow min(k, n)$ *//select the first k symbols, if exist*
6: $A \leftarrow$ linear automaton with $l + 1$ states $(q_i | 0 \leq i \leq l)$ and l transitions labeled with the first l symbols of s $((\langle q_{i-1}, q_i \rangle, s_i) | 1 \leq i \leq l)$
7: $q \leftarrow q_l$ *//model generation continues from this state*
8: $s \leftarrow \langle s_{l+1}, \ldots, s_n \rangle$ *//the remaining portion of the string*
9: **else**
10: $q \leftarrow q_0$ *//model generation continues from the initial state, no changes to the string*
11: **end if**
12: **if** s **not** empty **then**
13: $A \leftarrow$ addString(A, q, s, k, k_ρ) *//extend A from state q with string s*
14: **end if**
15: **return** A

kBehavior starts with an empty FSA, and processes strings incrementally. Given a string $s \in S$, the incremental step *extendFSA* modifies the input automaton A to extend $L(A)$ with the input string s. When invoked with an empty automaton, *extendFSA* initializes the automaton to a simple linear automaton that accepts the prefix of length k of s and removes the prefix from s (line 6

Algorithm 3 FSA addString(A, q, s, k ,k_ρ)

Require: an automaton $A = (Q, \Sigma, \delta, q_0, F)$
Require: a state $q \in Q$ that is the starting point to extend A
Require: a non-empty string $s = \langle s_1, \ldots, s_n \rangle$
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ or $k_\rho \geq k$
Ensure: an FSA A' that extends A to accept s from state q

//Prefix step
$\beta \leftarrow$ the longest $\langle s_1, \ldots, s_{max} \rangle$ accepted by A from state q
if β **not** empty **then**
  $s \leftarrow \langle s_{max+1}, \ldots, s_n \rangle$
end if
$ls \leftarrow n - max$
if s is empty **then**
  return A
end if
$q \leftarrow \delta^*(q, \beta)$ *//accepting state reached when accepting* β
//Pattern step
$\langle \mu, \rho, \nu, q_\rho \rangle \leftarrow$ pattern(A, s, k, k_ρ)
//Merge step
if ρ is empty **then** *//does not exist a subautomaton of* A *that accepts a substring of* s
  $A_s \leftarrow$ extendFSA($\varnothing, s$)
  $A \leftarrow$ addTail(A, q, A_s)
  return A
end if
//ρ *follows* μ *in* s*, and is accepted from state* q_ρ *of* A*.* μ *is not accepted by any subautomaton of* A
$B \leftarrow$ extendFSA($\varnothing$, μ, k, k_ρ) *//build an automaton that accepts* μ
$A \leftarrow$ merge(A, q, q_ρ, B) *//merge* B *with* A *at states* q *and* q_ρ
$q \leftarrow \delta^*(q, \mu\rho)$ *//accepting state reached when accepting* $\mu\rho$
if ν is **not** empty **then** *//*ν *is a tail of* s *still not accepted by* A
  $A \leftarrow$ addString(A, q, ν, k, k_ρ) *//extend* A *according to the remaining trace* ν
end if
return A

Algorithm 4 $\langle \mu, \rho, \nu, q_\rho \rangle$ pattern(A, s, k, k_ρ)

Require: an automaton $A = (Q, \Sigma, \delta, q_0, F)$
Require: a string $s = \langle s_1, \ldots, s_n \rangle$
Require: two integers k and k_ρ, with $k > 0$ and either $k_\rho = 0$ (best matching mode) or $k_\rho \geq k$ (optimized mode)
Ensure: returns three strings μ, ρ and ν and a state q_ρ. If there exists a decomposition of $s = \mu\rho\nu$, such that ρ is a substring longer than or equal to k_ρ (optimized mode) or the longest substring longer than k (best matching mode) accepted by A from a state q_ρ, ρ is non empty; otherwise, ρ is empty and the value of μ, ν and q_ρ are not relevant.

1: $\rho \leftarrow$ empty
2: $length_\rho \leftarrow 0$ *//the length of* ρ
3: **for** $i = 1$ to n **do**
4: **for all** $q_j \in Q$ **do**
5: $l \leftarrow$ the longest $\langle s_i, \ldots s_{l+i} \rangle$ accepted by A from state q_j
6: **if** $l > length_\rho$ **and** $l \geq k$ **then**
7: $length_\rho \leftarrow l$
8: $\mu \leftarrow \langle s_1, \ldots s_{i-1} \rangle$
9: $\rho \leftarrow \langle s_i, \ldots s_{l+i} \rangle$
10: $\nu \leftarrow \langle s_{l+i+1}, \ldots s_n \rangle$
11: $q_\rho \leftarrow q_j$
12: **end if**
 //if $k_\rho \neq 0$ *we use the* optimized *mode that returns a pattern when finding a string of length* $l \geq k_\rho$
13: **if** $l \geq k_\rho$ **and** $k_\rho \neq 0$ **then**
14: **return** $\langle \mu, \rho, \nu, q_\rho \rangle$ *//optimized mode*
15: **end if**
16: **end for**
17: **end for**
18: **return** $\langle \mu, \rho, \nu, q_\rho \rangle$

in Algorithm 2). If s is empty, kB-Inc initializes the automaton to the trivial automaton with two states connected by an ϵ transition (line 2 in Algorithm 2). If s is shorter than k, *extendFSA* just returns the sequential automaton that accepts s. If s is longer than k, kB-Inc removes the first k symbols from s and invokes the recursive step *addString* with the newly created linear automaton, the tail of the input string, and the parameter q set to the final state of the linear automaton just created (line 13 in Algorithm 2). The parameter q is the state from which kB-Rec starts searching for a subautomaton that accepts a substring of the given string s to extend the automaton with s itself, as discussed below.

When invoked with a non-empty automaton, *extendFSA* simply invokes *addString* (line 13 in Algorithm 2) with the parameter q initialized to the initial state q_0 of A (line 10 in Algorithm 2).

kB-Rec recursively processes an FSA A, a state q, and a string s in three main steps: *Prefix* (from line 1 to line 9 in Algorithm 3), *Pattern* (line 10 in Algorithm 3), and *Merge* (from line 11 to line 18 in Algorithm 3).

In the *Prefix* step, *addString* identifies the longest prefix of s accepted by A starting from q(line 1).

Algorithm 5 FSA merge(A, q_1, q_2, B)

Require: an automaton $A = (Q, \Sigma, \delta, q_0, F)$ that will be augmented with B
Require: a state $q_1 \in Q$ that will be connected to the initial state of B
Require: a state $q_2 \in Q$ that will be connected to the final state of B
Require: an automaton $B = (Q', \Sigma', \delta', q'_0, F')$ that will be merged in A
Ensure: an FSA A' that extends A with B

//avoid spurious loops: if the two states q_1 and q_2 are the same state q, function split creates two new states q_1 and q_2, such that all transitions entering q enter q_1 and all transitions exiting q exit q_2

1: **if** $q_1 == q_2$ **then**
2: $Q \leftarrow Q \cup q_{new}$ *//extend Q with a new state q_{new}*
3: for each transition from q_1 to q_i, extend δ with a transition from q_{new} to q_i and the same label
4: eliminate from δ all transitions from q_1 to any state
5: extend δ with an ϵ transition from q_1 to q_{new}
6: $q_2 \leftarrow q_{new}$
7: **end if**
8: $Q \leftarrow Q \cup Q'$
9: merge $q'_0 \in Q'$ with $q_1 \in Q$
10: merge each state in F' with $q_2 \in Q$
11: $\Sigma \leftarrow \Sigma \cup \Sigma'$
12: $\delta \leftarrow \delta \cup \delta'$ *//add transitions in δ' to δ*
13: **return** A

Algorithm 6 FSA addTail(A, q, B)

Require: an automaton $A = (Q, \Sigma, \delta, q_0, F)$
Require: a state $q \in Q$
Require: an automaton $B = (Q', \Sigma', \delta', q'_0, F')$
Ensure: return A extended by merging q'_0 with q and adding F' to F

1: $Q \leftarrow Q \cup Q'$
2: $q'_0 \in Q$ is merged with $q \in Q$
3: $F \leftarrow F \cup F'$
4: $\Sigma \leftarrow \Sigma \cup \Sigma'$
5: $\delta \leftarrow \delta \cup \delta'$ *//add transitions in δ' to δ*
6: **return** A

In the *Pattern* step then *addString* looks for a substring ρ of s longer than k and accepted by a subautomaton of A. This action is performed by the function *pattern* (line 10 in Algorithm 3) that is detailed in Algorithm 4. The pattern function can work in two modes, according to the value of parameter k_ρ. If k_ρ is equal to 0 *pattern* works in *best matching* mode: it scans s sequentially and looks for the longest substring accepted by a subautomaton of A. If k_ρ is greather than k, *pattern* works in *optimized* mode: it stops looking for substrings accepted by a subautomaton of s when it finds a substring not shorter than k_ρ.

In the *Merge* step *addString* first modifies A to accept the prefix $\mu\rho$ of s from state q and then it extends A to accept the whole string.

If ρ is empty, there is no subautomaton of A that accepts a substring of s with at least k symbols. Thus, the *Merge* step of kB-Rec adds a tail to automaton

A by creating an automaton that accepts s (line 11 to line 15 in Algorithm 3) and connecting q to that automaton by invoking method *addTail* described in Algorithm 6.

If ρ is not empty, the *Merge* step of kB-Rec first connects q to q_ρ by means of an automaton B that accepts μ, thus including $\mu\rho$ in $L(A)$, and then it continues processing ν (line 16 to line 18 in Algorithm 3). *addString* generates an automaton B that accepts μ by calling *extendFSA* with μ and an empty automaton as parameters (line 16 in Algorithm 3). Then it includes B in A by merging q with the initial state of B, and the final state of B with q_ρ (call of the *merge* function at line 17 in Algorithm 3).

Algorithm 5 shows the details of the *merge* function, which is straightforward except when invoked with $q = q_\rho$. In this case simply inserting B by merging the initial and final states of B with the same state creates a spurious loop. To avoid creating this spurious loop, the *merge* function splits q into two states connected with an empty (ϵ) transition, and merges B with these two states (lines 1 to 7 in Algorithm 5).

Once the current FSA has been extended with B, kB-rec extends the automaton to accept the remaining substring ν. If ν is empty *addString* can just return A because it already accepts the string received as input. If ν is not empty *addString* invokes recursively *addString* with the new FSA, the string ν, and q as the initial state (line 20 in Algorithm 3). This extends A to accept ν from q. kB-rec then returns A.

4 Complexity Analysis

This Section analyzes the complexity of the kBehavior algorithm in detail. In the analysis we assume that the following operations can be completed with constant time and can be considered as unitary when computing the algorithmic complexity: adding a transition to an automaton (this includes adding the initial or the final state), checking if a given state accepts a given symbol and merging two states.

In this Section we refer to the cost of computing the functions that are part of kBehavior by using the notation $T_f(args)$, where T_f denotes the cost of computing the function f, and $args$ are the length l of the trace to be processed and the number q of states in the automaton.

Following paragraphs estimate the best case complexity, an over-approximation of the worst case complexity when the inferred automaton is deterministic, an over-approximation of the worst case complexity when the inferred automaton is non-deterministic (which seldom occurs as discussed in Section 5) and an over-approximation of the average complexity of the algorithm.

Best Case Complexity

The best case is given by the analysis of a trace of length l that is already accepted by the automaton. This trace is processed in l steps by prefix acceptance (lines 1 to 7 of the Algorithm 3). The best case complexity is thus linear with respect to the length of the trace, which is the optimum algorithm to verify that a word of l symbols belongs to the language accepted by a given automaton.

Worst Case Complexity When the Inferred Automaton Is Deterministic

To estimate the worst case complexity, we must estimate the complexity of the algorithms `merge`, `addTail`, `pattern`, *extendFSA* and ***`addString`*** .

The `merge` (Algorithm 5) and `addTail` (Algorithm 6) algorithms merge two pairs of states and one pair of states, respectively. They can be completed in constant time and do not significantly affect the overall complexity of the algorithm.

Given a substring s and an automaton A, the `pattern` algorithm identifies the subautomaton of A that accepts the longest substring of s. The initial state of the identified subautomaton can be any state of A (line 4 in Algorithm 4), and the initial element of the substring can be any element of s (line 3 in Algorithm 4). The complexity of `pattern` depends on the number of matches required to identify a subautomaton and a corresponding substring. Given a generic pair $\langle state, index \rangle$ that represents a state *state* of A and position *index* in s, the amount of required matches is maximum when the subautomaton with *state* as initial state accepts $k_\rho - 1$ symbols of the substring that starts at position *index*. In fact, if the subautomaton accepts more than $k_\rho - 1$ symbols, function `pattern` terminates and does not consider further pairs of states and positions, while if it accepts less than $k_\rho - 1$ symbols, it skips the subautomaton and proceeds to the next one.

The worst case scenario is the (unrealistic) case of an automaton A that, from all its states, accepts all the possible substrings of length $k_\rho - 1$ of s. In this case $T_{pattern}(q, l) < k_\rho q l$. Note that here we consider the case of deterministic automata, where at any step there is at the most one transition labelled with the considered symbol. We will consider the case of nondeterministic automata later in this appendix.

The analysis of the complexity of the algorithm *extendFSA* (Algorithm 2) is quite simple. When the input automaton is empty, in the worst case *extendFSA* creates a linear automaton with k transitions and invokes *addString* with the newly created automaton and the string without the first k symbols, thus, $T_{extendFSA}(0, l) = k + T_{addString}(k, l - k)$, where k is the kBehavior constant. When the input automaton is non empty, in the worst case *extendFSA* initializes a variable and then invokes *addString*, thus $T_{extendFSA}(q \neq 0, l) = T_{addString}(q, l)$.

In the worst case, *addString* finds an empty prefix of the string s accepted by the automaton. In this case, *addString* looks for a substring ρ of s with at least k elements. If it finds such a substring, it decomposes s in $\mu\rho\nu$, invokes *extendFSA* to generate the automaton that accepts μ and invokes *addString* recursively to analyse ν with the automaton extended with the new transitions and states added by *extendFSA* (Algorithm 3). When it does not find a substring ρ with at least k elements accepted by the automaton, *extendFSA* adds the tail of s to the automaton (line 11 in Algorithm 3). We discus this case later in this section. In the following, we use the symbols μ, ρ and ν to denote both the substrings of s to be processed and their length.

The worst case scenario leads to the following worst case complexity of *addString*:

$T_{addString}(q, l) = T_{\texttt{pattern}}(q, l) + T_{extendFSA}(0, \mu) + T_{addString}(q + q_{new}, \nu) =$

$= T_{\texttt{pattern}}(q, l) + k + T_{addString}(k, \mu - k) + T_{addString}(q + q_{new}, \nu)$

where q_{new} is the number of states added to the current automaton by invoking

extendFSA$(0, \mu)$.

The elements of the formula satisfy the following properties:

1. μ, ρ ν is a partition of l, thus $l = \mu + \rho + \nu$
2. Let p be the number of symbols removed from s in each iteration, then $p = \mu + \rho$ where ρ is the substring already accepted by the current automaton of length at least k, and μ is the (possibly empty) substring accepted by the automaton that *extendFSA*$(0, \mu)$ adds to the current automaton. $\rho \geq k$ and $p \geq k$.
3. Let n be the number of iterations of `kBehavior`, then $l = np$, where p is the number of symbols removed at each iteration from the string s. Property 2, implies that $n = \frac{l}{p} \leq \frac{l}{k}$
4. Property 2 implies that $\mu = p - \rho \leq p - k \Rightarrow \mu - k \leq p - 2k$. We can thus introduce the following over-approximation that increases the size of the problem $T_{addString}(k, \mu - k) \leq T_{addString}(k, p - 2k)$
5. `kBehavior` adds at most q_{new} states while processing a string of length μ, thus the maximum number of states added to the automaton corresponds to generating a linear branch with μ states. This implies $q_{new} \leq \mu = p - \rho \leq p - k$
6. Properties 1, 2 and 3 imply that $\nu = l - p = np - p = (n - 1)p$.
7. Properties 5 and 6 imply that $T_{addString}(q + q_{new}, \nu) \leq T_{addString}(q + p - k, (n - 1)p)$

Given the worst case complexity of the algorithm `pattern`, we can use properties 4 and 7 to over-approximate the worst case complexity of *addString* as follows:
$T_{addString}(q, l = np) < k_\rho ql + k + T_{addString}(k, p - 2k) + T_{addString}(q + p - k, (n - 1)p)$.

Since at every step we advance of p symbols and $p - 2k < p$, *addString*$(k, p - 2k)$ is processed in one step after the execution of the function `pattern`. It derives that $T_{addString}(k, p - 2k) = T_{pattern}(k, p - 2k) < kk_\rho(p - 2k) \leq kk_\rho p$.

We can update the worst case formula as $T_{addString}(q, l = np) < k_\rho ql + k + kk_\rho p + T_{addString}(q + p - k, (n - 1)p)$.

We now unfold a few steps of the recursion. $T_{addString}(q, l = np) < k_\rho ql + k + kk_\rho p + T_{addString}(q + p - k, (n - 1)p) < k_\rho ql + k + kk_\rho p + k_\rho(q + p - k)(n - 1)p + k + kk_\rho p + T_{addString}(q + 2p - 2k, (n - 2)p) < k_\rho ql + k + kk_\rho p + k_\rho(q + p - k)(n - 1)p + k + kk_\rho p + k_\rho(q + 2p - 2k)(n - 2)p + k + kk_\rho p + T_{addString}(q + 3p - 3k, (n - 3)p)$. Since we know that the recursion consists of at most $\frac{l}{p}$ iterations, we can transform the recursion into a sum.

$T_{addString}(q, l = np) < k_\rho ql + \frac{l}{p}k + kk_\rho \frac{l}{p}p + k_\rho \sum_{i=1}^{\frac{l}{p}}(q + ip - ik)(n - i)p = k_\rho ql + \frac{l}{p}k + kk_\rho l + k_\rho \sum_{i=1}^{\frac{l}{p}}(q + ip - ik)(n - i)p$.

Since $\forall i = 1, \ldots \frac{l}{p}$, $q + ip \geq ik$ and $n \geq i$, we can identify an upper bound for the sum as follow.

$T_{addString}(q, l = np) < k_\rho ql + \frac{l}{p}k + kk_\rho l + k_\rho \sum_{i=1}^{\frac{l}{p}}(q + ip)np < k_\rho ql + \frac{l}{p}k + kk_\rho l + k_\rho \sum_{i=1}^{\frac{l}{p}} qnp + k_\rho \sum_{i=1}^{\frac{l}{p}} inp^2 = k_\rho ql + \frac{l}{p}k + kk_\rho l + k_\rho \frac{ql^2}{p} + \frac{k_\rho}{2} lp \frac{l}{p}(\frac{l}{p} + 1)$

In the worst case, the number of symbols removed from a trace in one iteration is the smallest, that is $p = k$. We thus have $T_{addString(q,l)} = O(ql^2 + l^3)$.

We still have to consider the complexity when the function `pattern` does not find a decomposition and kBehavior generates an automaton to be merged as a tail. In this case, the algorithm produces a linear automaton with k symbols, invokes *addString* with the newly generated linear automaton and the string without k symbols and merges the final result as a tail (line 11-15 of Algorithm 3).

When we add a tail, the complexity is $T_{addString}(q, l) < k_\rho ql + k + T_{addString}(k, l - k)$. If *addString* finds some patterns, we already know that the worst case complexity of $T_{addString}(k, l - k)$ is $O(k(l - k)^2 + (l - k)^3) = O(l^3)$. Otherwise, if the algorithm continues adding tails, we have $T_{addString}(k, l - k) < k_\rho k(l - k) + k + T_{addString}(k, l - 2k)$. If we transform the recursion into a sum and we simplify the sum, we obtain $T_{addString}(k, l - k) < \frac{l^2}{p} k k_\rho + \frac{l}{p} k$. If we replace this result in the formula for the cost of the *addString* function for tail addition we have $T_{addString}(q, l) < k_\rho ql + k + \frac{l^2}{p} k k_\rho + \frac{l}{p} k$. If again we substitute p with k, we have $T_{addString}(q, l) = O(ql + l^2)$.

Thus, the worst case complexity of *addString* is upper bounded by $O(ql^2 + l^3)$.

Since kBehavior processes traces by executing once the algorithm *extendFSA* once for each of the traces, and the computational cost of the algorithm *extendFSA* differs from *addString* only for a constant value, the worst case complexity of kBehavior when processing a trace has an upper bound of $O(ql^2 + l^3)$.

Worst Case Complexity When the Inferred Automaton Is Non-Deterministic

To compute the worst case complexity in the non-deterministic case, we start from the general worst case upper bound $T_{addString}(q, l) = T_{\texttt{pattern}}(q, l) + k + T_{addString}(k, \mu - k) + T_{addString}(q + q_{new}, \nu)$

If we consider the worst case of an automaton with all the states non-deterministically connected to every other state in the automaton, the cost of the pattern function increases. In the deterministic case, the pattern function checks at most k_ρ symbols of the string for every pair of state in the automaton and position in the string, leading to a worst case complexity of qlk_ρ. In the non deterministic case, after the first check, kBehavior must consider q and not 1 state. Note that the case of a non-deterministic automaton with all the states completely connected with non-deterministic transitions to every other state in the automaton is not realistic, but represents a manageable over-approximation of the worst case scenario. In this case, we can derive an upper bound $T_{\texttt{pattern}}(q, l) \leq qlqk_\rho = k_\rho q^2 l$.

All the relations that we have identified in the deterministic case are still true in the non-deterministic case, we can thus refine the formula with $T_{addString}(q, l = np) < k_\rho q^2 l + k + k^2 k_\rho p + T_{addString}(q + p - k, (n - 1)p)$.

By unfolding the recursion for some steps we obtain $T_{addString}(q, l = np) \leq k_\rho q^2 l + k + k^2 k_\rho p + T_{addString}(q + p - k, (n - 1)p) = k_\rho q^2 l + k + k^2 k_\rho p + k_\rho (q + p - k)^2 (n - 1)p + k + k^2 k_\rho p + T_{addString}(q + 2p - 2k, (n - 2)p) = k_\rho q^2 l + k + k^2 k_\rho p + k_\rho (q + p - k)^2 (n - 1)p + k + k^2 k_\rho p + k_\rho (q + 2p - 2k)^2 (n - 2)p + k + k^2 k_\rho p + T_{addString}(q + 3p - 3k, (n - 3)p)$. Since we know that the recursion consists of at most $\frac{l}{p}$ iterations, we can transform the recursion into a sum as follows.

$T_{addString(q,l=np)} \leq k_\rho q^2 l + \frac{l}{p}k + k^2 k_\rho \frac{l}{p}p + k_\rho \sum_{i=1}^{\frac{l}{p}}(q+ip-ik)^2(n-i)p = k_\rho q^2 l + \frac{l}{p}k + k^2 k_\rho l + k_\rho \sum_{i=1}^{\frac{l}{p}}(q+ip-ik)^2(n-i)p \leq k_\rho q^2 l + \frac{l}{p}k + k^2 k_\rho l + k_\rho np \sum_{i=1}^{\frac{l}{p}}(q+ip)^2 = k_\rho q^2 l + \frac{l}{p}k + k^2 k_\rho l + k_\rho np \sum_{i=1}^{\frac{l}{p}} q^2 + k_\rho np \sum_{i=1}^{\frac{l}{p}} i^2 p^2 + k_\rho np \sum_{i=1}^{\frac{l}{p}} 2qip = k_\rho q^2 l + \frac{l}{p}k + k^2 k_\rho l + k_\rho npq^2 \frac{l}{p} + \frac{k_\rho}{6} npp^2 \frac{l}{p}(\frac{l}{p}+1)(2\frac{l}{p}+1) + qpk_\rho np \frac{l}{p}(\frac{l}{p}+1) = k_\rho q^2 l + \frac{l}{p}k + k^2 k_\rho l + k_\rho lq^2 \frac{l}{p} + \frac{k_\rho}{6} lp^2 \frac{l}{p}(\frac{l}{p}+1)(2\frac{l}{p}+1) + qpk_\rho l \frac{l}{p}(\frac{l}{p}+1)$.

If we substitute p with k, we have $addString(q,l) = O(q^2 l^2 + l^4 + ql^3)$.

Upper Bound of the Average Case Complexity when the Inferred Automaton Is Deterministic

We can also compute an upper bound of the average complexity of the kBehavior algorithm to get closer to realistic cases.

In the computation of the worst case complexity, we assume that the length of the prefix of the string that is already accepted by the automaton is empty. However, this prefix is not empty both in the general case and in the average case. In particular, at every iteration, a string to be processed is decomposed into 4 parts: a $prefix$, which is already accepted by the automaton, a substring ρ that is already accepted by a subautomaton, a substring μ that separates the $prefix$ from ρ and is not accepted by any subautomaton and a tail nu of the string. The length of each of these substrings can vary from 0 to l depending from the specific sample. For instance, if the current trace is already accepted by the automaton, the prefix is the entire string, and the other substrings are empty. If the current trace is accepted from a state different from the initial state, the substring ρ is the entire string, and the other substrings are empty. Different combinations of these values are possible according to the cases. We do not make hypotheses on the frequency of each decomposition case, and we consider an average length of $\frac{l}{4}$ for each of these substrings.

Due to the huge number of possible executions of the function `pattern` and the difficulty in identifying an average case, we do not compute the average complexity that would be quite arbitrary, but we over-approximate the cost of the average case by referring to the worst case of the function `pattern`: $T_{\texttt{pattern}}(q,l) = \frac{k_\rho q l}{2}$.

Updating the *addString* complexity under this hypothesis, we obtain $T_{addString}(q,l) = prefix + T_{\texttt{pattern}}(q, l-prefix) + T_{extendFSA}(0,\mu) + T_{addString}(q+q_{new}, \nu) < \frac{l}{4} + \frac{3k_\rho}{4} ql + k + T_{addString}(k, \frac{l}{4}-k) + T_{addString}(q+q_{new}, \frac{l}{4})$

Given that

1. $\frac{l}{4} - k < \frac{l}{4}$, we have $T_{addString}(k, \frac{l}{4}-k) < T_{addString}(k, \frac{l}{4})$
2. q_{new} is between 0 and $\mu = \frac{l}{4}$. Assuming a uniform distribution, the average value of the newly added states q_{new} is $\frac{l}{8}$

We can update the over-approximation of the average case as follow.

$T_{addString}(q,l) < \frac{l}{4} + \frac{3k_\rho}{4} ql + k + T_{addString}(k, \frac{l}{4}) + T_{addString}(q+\frac{l}{8}, \frac{l}{4})$.

We unfold the recursion for a few steps, introducing some over-approximations to compute the double recursion. We first consider the single recursive terms,

then we consider their sum, and finally we replace the result in the original expression.

$T_{addString}(k, \frac{l}{4}) < T_{addString}(q, \frac{l}{4}) \leq \frac{l}{16} + \frac{3k_\rho}{4} q\frac{l}{4} + k + T_{addString}(k, \frac{l}{16}) + T_{addString}(q+\frac{l}{32}, \frac{l}{16}) < \frac{l}{16} + \frac{3k_\rho}{4} q\frac{l}{4} + k + T_{addString}(k, \frac{l}{16}) + T_{addString}(q+\frac{l}{8}+\frac{l}{32}, \frac{l}{16})$

$T_{addString}(q+\frac{l}{8}, \frac{l}{4}) < \frac{l}{16} + \frac{3k_\rho}{4}(q+\frac{l}{8})\frac{l}{4} + k + T_{addString}(k, \frac{l}{16}) + T_{addString}(q+\frac{l}{8}+\frac{l}{32}, \frac{l}{16})$

$T_{addString}(k, \frac{l}{4}) + T_{addString}(q+\frac{l}{8}, \frac{l}{4}) < \frac{l}{8} + \frac{3k_\rho}{4} q\frac{l}{4} + 2k + \frac{3k_\rho}{4}(q+\frac{l}{8})\frac{l}{4} + 2T_{addString}(k, \frac{l}{16}) + 2T_{addString}(q+\frac{l}{8}+\frac{l}{32}, \frac{l}{16})$

Substituting this result in the above formula we have

$T_{addString}(q, l) < \frac{l}{4} + \frac{l}{8} + \frac{3k_\rho}{4} ql + \frac{3k_\rho}{4} q\frac{l}{4} + 3k + \frac{3k_\rho}{4}(q+\frac{l}{8})\frac{l}{4} + 2T_{addString}(k, \frac{l}{16}) + 2T_{addString}(q+\frac{l}{8}+\frac{l}{32}, \frac{l}{16})$

We show another iterative step.

$2T_{addString}(k, \frac{l}{16}) < 2T_{addString}(q, \frac{l}{16}) \leq \frac{l}{32} + \frac{3k_\rho}{4} q\frac{l}{8} + 2k + 2T_{addString}(k, \frac{l}{64}) + 2T_{addString}(q+\frac{l}{128}, \frac{l}{64}) < \frac{l}{32} + \frac{3k_\rho}{4} q\frac{l}{8} + 2k + 2T_{addString}(k, \frac{l}{64}) + 2T_{addString}(q+\frac{l}{8}+\frac{l}{32}+\frac{l}{128}, \frac{l}{64})$

$2T_{addString}(q+\frac{l}{8}+\frac{l}{32}, \frac{l}{16}) < \frac{l}{32} + \frac{3k_\rho}{4}(q+\frac{l}{8}+\frac{l}{32})\frac{l}{8} + 2k + 2T_{addString}(k, \frac{l}{64}) + 2T_{addString}(q+\frac{l}{8}+\frac{l}{32}+\frac{l}{128}, \frac{l}{64})$

We combine the two terms into a sum.

$2T_{addString}(k, \frac{l}{16}) + 2T_{addString}(q+\frac{l}{8}+\frac{l}{32}, \frac{l}{16}) < \frac{l}{16} + \frac{3k_\rho}{4} q\frac{l}{8} + 4k + \frac{3k_\rho}{4}(q+\frac{l}{8}+\frac{l}{32})\frac{l}{8} + 4T_{addString}(k, \frac{l}{64}) + 4T_{addString}(q+\frac{l}{8}+\frac{l}{32}+\frac{l}{128}, \frac{l}{64})$

If we substitute this result in the above formula.

$T_{addString}(q, l) < \frac{l}{4} + \frac{l}{8} + \frac{l}{16} + \frac{3k_\rho}{4} ql + \frac{3k_\rho}{4} q\frac{l}{4} + \frac{3k_\rho}{4} q\frac{l}{8} + 7k + \frac{3k_\rho}{4}(q+\frac{l}{8})\frac{l}{4} + \frac{3k_\rho}{4}(q+\frac{l}{8}+\frac{l}{32})\frac{l}{8} + 4T_{addString}(k, \frac{l}{64}) + 4T_{addString}(q+\frac{l}{8}+\frac{l}{32}+\frac{l}{128}, \frac{l}{64})$

Before transforming the recursion into a sum, we compute the value of the ith term of the sequence $\frac{l}{8}, \frac{l}{8}+\frac{l}{32}, \frac{l}{8}+\frac{l}{32}+\frac{l}{128}, \ldots$. The term a_i is given by $a_i = \sum_{j=1}^{i} \frac{l}{2*4^j} = \frac{l}{2}\sum_{j=1}^{i}(\frac{1}{4})^j = \frac{l}{2}(\frac{1-(\frac{1}{4})^{i+1}}{1-\frac{1}{4}} - 1)$, that can be over-approximated as $a_i = \frac{l}{6}(1-(\frac{1}{4})^i) < \frac{l}{6}$.

We can now compute the number of steps needed by *addString* to terminate. Note that the iterations above incrementally consider strings of lesser length until reaching a length less than k, which prevents the identification of any pattern and results in adding a tail with a number of transitions corresponding to the number of symbols in the remaining portion of the string. Thus, the recursion is interrupted when $\frac{l}{4^n} \leq k$, where n is the number of executed recursive steps, and then the number of recursive steps is $n = log_4 \frac{l}{k}$.

We are now ready to transform the recursion into a sum.

$T_{addString}(q, l) \leq \sum_{i=1}^{n} \frac{l}{4^i} + \sum_{i=1}^{n} \frac{3k_\rho}{8} q\frac{l}{2^i} + \sum_{i=1}^{n} 2^i k + \sum_{i=1}^{n} \frac{3k_\rho}{8}(q+a_i)\frac{l}{2^i} < l(\frac{4}{3} - \frac{1}{3}(\frac{1}{4})^n) + \frac{3k_\rho}{8} q(2-(\frac{1}{2})^n) + k(2^{n+1}-1) + \frac{3k_\rho}{8} lq(2-(\frac{1}{2})^n) + \frac{3k_\rho}{8} l\frac{l}{6}(2-(\frac{1}{2})^n)$

If we replace n with $log_4 \frac{l}{k}$, we have

$T_{addString}(q, l) \leq l(\frac{4}{3} - \frac{1}{3}\frac{k}{l}) + \frac{3k_\rho}{8} q(2-\sqrt{\frac{k}{l}}) + k(2\sqrt{\frac{l}{k}} - 1) + \frac{3k_\rho}{8} lq(2-\sqrt{\frac{k}{l}}) + \frac{3k_\rho}{8} l\frac{l}{6}(2-\sqrt{\frac{k}{l}}) \leq l\frac{4}{3} + \frac{6k_\rho}{8} q + 2k\sqrt{\frac{l}{k}} + \frac{6k_\rho}{8} lq + \frac{6k_\rho}{8} l\frac{l}{6}$.

We have thus identified the following over-approximation of the average case complexity $O(ql + l^2)$.

5 Conclusions

This report is a compendium to [1] and it presents an analysis of the complexity of kBehavior, a FSA inference algorithm.

The complexity of kBehavior depends on the number q of states in the automaton to be extended with a new trace and the length l of the trace to be processed.

The best case complexity corresponds to the case of a trace already accepted by the current automaton. In this case, the trace is processed in linear time ($O(l)$ operations).

The worst case complexity depends on the kind of inferred automaton. When inferring a deterministic automaton, the over-approximated worst case complexity is $O(ql^2 + l^3)$. When inferring a non-deterministic automaton, the over-approximated worst case complexity is $O(q^2l^2 + l^4 + ql^3)$. These results suggest a high impact on the length of the string. However, empirical results obtained when applying kBehavior to infer models of several open source software systems indicate that the algorithm can also process long strings efficiently [1].

We estimate the over-approximated average complexity to be $O(ql+l^2)$. The strong dependency on the length of the string is inherited from the worst case complexity, however it is not confirmed by our experiments that indicate better values, as discussed above.

Acknowledgment

This work has been partially supported by the European Community under the Information Society Technologies (IST) programme of the 6th FP for RTD - project SHADOWS contract IST-035157.

References

[1] L. Mariani, F. Pastore, and M. Pezzè, "Dynamic analysis for diagnosing integration faults," *IEEE Transactions on Software Engineering*, 2010, submitted.

[2] L. Wendehals and A. Orso, "Recognizing behavioral patterns atruntime using finite automata," in *WODA '06: Proceedings of the 2006 international workshop on Dynamic systems analysis.* New York, NY, USA: ACM, 2006, pp. 33–40.

[3] V. Dallmeier, A. Zeller, and B. Meyer, "Generating fixes from object behavior anomalies," in *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering*, Auckland, New Zealand, November 2009.

[4] L. Mariani, F. Pastore, and M. Pezzè, "A toolset for automated failure analysis," in *In ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 563–566.

[5] G. Ammons, R. Bodík, and J. R. Larus, "Mining specifications," in *proceedings of the 29th Symposium on Principles of Programming Languages*. ACM Press, 2002, pp. 4–16.

[6] L. Mariani and M. Pezzè, "Dynamic detection of COTS components incompatibility," *IEEE Software*, vol. 24, no. 5, pp. 76–85, September/October 2007.

[7] A. Wasylkowski, A. Zeller, and C. Lindig, "Detecting object usage anomalies," in *proceedings of the 6th joing meeting of the European Software Engineering Conference and the International Symposium on Foundations of Software Engineering*. ACM Press, 2007, pp. 35–44.

[8] R. Parekh, C. Nichitiu, and V. Honavar, "A polynomial time incremental algorithm for learning DFA," in *proceedings of the 4th International Colloquium on Grammatical Inference*, ser. LNCS, vol. 1433. Springer, 1998.

[9] D. Lorenzoli, L. Mariani, and M. Pezzè, "Automatic generation of software behavioral models," in *30th International Conference on Software Engineering (ICSE)*. IEEE Computer Society, May 2008.

[10] S. Porat and J. Feldman, "Learning automata from ordered examples," *Machine Learning*, vol. 7, pp. 109–138, 1991.

[11] J. Oncina and P. Garcia, "Inferring regular languages in polynomial update time," in *Pattern Recognition and Image Analysis*, N. P. de la Blanca, A. Sanfeliu, and E.Vidal, Eds. World Scientific, 1992, pp. 49–61.

[12] O. Cicchello and S. C. Kremer, "Inducing grammars from sparse data sets: a survey of algorithms and results," *Journal of Machine Learning Research*, vol. 4, pp. 603–632, 2003.

[13] P. Dupont, "Incremental regular inference," in *proceedings of the 3rd International Colloquium on Grammatical Inference*, ser. LNCS, L. Miclet and C. Higuera, Eds., vol. 1147. Springer-Verlag, 1996.

[14] J. Cook and A. Wolf, "Discovering models of software processes from event-based data," *ACM Transactions on Software Engineering and Methodology*, vol. 7, no. 3, pp. 215–249, 1998.

[15] A. Biermann and J. Feldman, "On the synthesis of finite state machines from samples of their behavior," *IEEE Transactions on Computer*, vol. 21, pp. 592–597, June 1972.

[16] S. P. Reiss and M. Renieris, "Encoding program executions," in *proceedings of the 23rd International Conference on Software Engineering*. IEEE Computer Society, 2001, pp. 221–230.

[17] M. Renieris and S. Reiss, "Fault localization with nearest neighbor queries," in *proceedings of the 18th Internation Conference on Automated Software Engineering*. IEEE Computer Society, 2003, pp. 30–39.

[18] L. Mariani and M. Pezzè, "Behavior capture and test: automated analysis of component integration," in *proceedings on the 10th International Conference on Engineering Complex Computer Systems*. IEEE Computer Society, 2005, pp. 292–301.

[19] A. Babenko, L. Mariani, and F. Pastore, "Ava: automated interpretation of dynamically detected anomalies," in *ISSTA '09: Proceedings of the eighteenth international symposium on Software testing and analysis*. New York, NY, USA: ACM, 2009, pp. 237–248.

[20] D. Cotroneo, R. Pietrantuono, L. Mariani, and F. Pastore, "Investigation of failure causes in workload-driven reliability testing," in *Fourth international workshop on Software quality assurance (SOQUA'07)*. ACM, 2007, pp. 78–85.

[21] L. Mariani and F. Pastore, "Automated identification of failure causes in system logs," in *Proceedings of the 19th IEEE International Symposium on Software Reliability Engineering (ISSRE'08)*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 117 – 126.