Progressively Generating Traces for Parallel Processes

Zhang Ke[a], Qiu Zongyan[a]

[a]*LMAM and Department of Informatics, School of Math., Peking University, Beijing, China*

Abstract

We sometimes need to enumerate possible traces of a set of parallel processes which run in interleaving manner, or in other words, run in all possibly different speeds, e.g., in carrying on related testing, or model checking. However, for a large and complex parallel system, the number of its possible execution traces is huge, then how to enumerate these traces in a better way becomes an interesting problem. This paper provides two ways to classify possible traces of parallel processes, with the generating algorithms. Both methods can progressively generate subsets of all the possible traces. These methods can be used in various circumstances to provide stronger and stronger confidence on the behaviors of parallel programs, e.g., model checking, testing, etc.

Keywords: Parallel Processes, Model Checking, Trace Generation

1. Introduction

We sometimes want to enumerate possible traces of a set of parallel processes which run in the interleaving manner, or in other words, run in all possibly different speeds. Many cases of this problem can be found in practice, e.g., in testing parallel programs, in model checking parallel processes, etc.

In practice, the trace sets are often huge, and it is time and resource expensive, even impossible to try all the traces before putting the models or systems into the next working stage or real use. Existing works often use naive techniques. For example, the popular model checker SPIN simply uses Depth First Search to enumerate all the traces. But in most cases, we have only limited recourse comparing to the size of the target models or systems, but want to understand possible behaviors of them deeply, even if not completely. In this case, some enumerating criteria or strategies, to go beyond the naive enumerating, will be interesting and valuable.

In an early work [2], Katayama *et al.* proposed the enumerating criteria OSC_k (Ordered Sequence Criteria) for classifying the traces of parallel pro-

Email addresses: zksms@pku.edu.cn (Zhang Ke), zyqiu@pku.edu.cn (Qiu Zongyan)

cesses, where k is a natural number parameter. For each $k > 1$, OSC_k provides a stronger assurance than OSC_{k-1} on the coverage of the behaviors of the parallel processes. If we have some method following the criteria to enumerate progressively OSC_1, then OSC_2, etc., and use them to explore executions of parallel processes, we might gain more and more confidence about their behaviors. Although the idea for OSC_k is interesting, no algorithm for generating OSC trace sets was proposed in [2], nor in other literature. To put the idea into practice, in this paper, we present an algorithm which generates the trace sets OSC_k for all k from given programs. Furthermore, our algorithm ensures that the generated OSC_k is a superset of OSC_{k-1}, thus OSC_1, $OSC_2 \backslash OSC_1$, $OSC_3 \backslash OSC_2 \ldots$ form a classification of the whole trace set, which is not guaranteed by the original definition of OSC_k. The algorithm allows us to use the incrementally generated traces to organize analysis procedures. More details for OSC_k will be given in the next section.

In addition, we propose another criteria FSC_k (Fair Sequences Criteria) of classifying the trace set. For each $k > 1$, FSC_k is, by the definition, a super set of FSC_{k-1}, thus provides a stronger assurance on the behavior of parallel processes. We can also use the classification FSC_1, $FSC_2 \backslash FSC_1$, $FSC_3 \backslash FSC_2 \ldots$ to organize a progressive analysis for parallel processes. An algorithm for generating FSC_k sets is also developed and analyzed in this paper.

The paper is organized as follows: Section 2 introduces definitions for the criteria OSC_k and FSC_k, and give some intuitions.Section 3 firstly presents an algorithm for generating OSC_1 and proves its correctness and termination, then the algorithm for generating OSC_k in general. Section 4 present an algorithm for generating FSC_k, as well as some analysis for FSC_k and the algorithm. Then we offer some information for the implementations. Section 6 concludes the paper with some discussions.

2. The criteria OSC_k and FSC_k

In this section, we introduce the definitions of the criteria OSC_k and FSC_k, and discuss their features in classifying traces of parallel processes. Here we only discuss with the case of two processes, the extension to the case of multi-processes is natural.

Suppose there are two parallel processes, and they have to execute m and n atomic sequential commands respectively. We denote them as $p_1 = (A_1; A_2; ...; A_m)$ and $p_2 = (B_1; B_2; ...; B_n)$ respectively.

To understand the interleaving traces of the two processes, we consider about their state graph, which can be drawn as an $(m + 1) * (n + 1)$ mesh, in which $(0, 0)$ represents the start point of the execution, and (m, n) represents the state where the two processes have terminated. In general, state (i, j) in the mesh represents a situation where the first process has just executed i commands, and the second has just executed j commands. In each step, the execution can either go downward from (i, j) to $(i, j + 1)$ (when $j < n$), to mean that B_{j+1} is done; or go rightward from (i, j) to $(i + 1, j)$ (when $i < m$) to mean the completion of A_{i+1}. Since every complete execution (complete trace) corresponds to a path in the mesh which starts from state $(0,0)$ and terminates at (m,n), we will take the paths in the mesh instead of the executions of the processes in the following discussion.

2.1. Introduction of OSC_k

Firstly, we give a definition for OSC_k following the idea of [2], although in the original paper, the authors did not give a clear definition as this.

Definition 1 (criterion OSC_k)**.** A set of traces from state $(0,0)$ to (m,n) satisfies criterion OSC_k, where $1 \leq k \leq m+n$, when it covers all the possible i-length paths in the state transition mesh, for all $i \leq k$. That is, each possible sequence of length k or less in the execution mesh of the program would appears in at least one trace of the OSC_k set.

Clearly, for a given program, each OSC_k is in general not a uniquely defined set. For example, a trace set satisfies OSC_k also satisfies OSC_{k-1}, and the set of all possible traces satisfies OSC_k for all k. Thus, this definition is not very useful in practice. Our algorithm given below will try to generate a set satisfying OSC_k which is as small as possible. (However, it might not be the smallest one, as shown below.)

Intuitively, OSC_1 requires that each edge in the mesh appears in at least one trace of the set. And the OSC_2 means, for any $0 < i \leq m$, $0 < j \leq n$, the trace set must include at least one trace which takes the form $(\ldots; A_i; B_j; \ldots)$, $(\ldots; B_j; A_i; \ldots)$, $(\ldots; A_{i-1}, A_i; \ldots)$, or $(\ldots, B_{j-1}; B_j; \ldots)$. In the view of the state mesh, the trace set must include four traces going through state (i,j) which take all the four different possible ways. In general, OSC_k requires that for any x which $0 \leq x \leq k$, and for any i, j satisfy $0 \leq i, (i+x-1) \leq m, 0 \leq j, (j+k-x-1) \leq n$, the path set OSC_k must cover all the possible sequences of the form $\{A_i, A_{i+1}, \ldots, A_{i+x-1}, B_j, B_{j+1}, \ldots, B_{j+k-x-1}\}$.

When two parallel processes have some synchronizing points, we can divide their execution into segments, then take above strategy. After all the paths in OSC_k are checked, we could say all the possible sequences of visiting this shared variable has been checked, and we can be more convinced with the correctness of the parallel processes' behavior. For example, suppose there is a shared variable x, and only commands A_3, A_5, B_6 visit x which should be done exclusively. As the above paragraph said, a criterion OSC_4 covers all the possible sequence of A_3, A_4, A_5, B_6. So, we can say OSC_4 covers all the possible visit sequence of variable x.

2.2. Introduction of FSC_k

Due to some disadvantages (which will be discussed later) of criteria OSC_k, we develop another criteria FSC_k as follows.

Definition 2 (criterion FSC_k)**.** Suppose $m \leq n$, for a trace t from $(0,0)$ to (m,n), $t \in FSC_k$ iff each state (i,j) passed by t satisfies $i < m \Rightarrow |i-j| \leq k$, and $i = m \Rightarrow i - j \leq k$.

Intuitively, FSC_k mean that the difference between the execution speeds of the two processes should not be too large. More precisely, FSC_k requires that the difference between the numbers of executed commands for the two processes cannot be larger than k before one of them ends. Clearly, FSC_1 includes those traces which ensure the strongest fairness, that one process can only be at most one step faster than another process; FSC_2 ensures a weaker fairness and thus it includes more traces, in which one process can be two steps faster than another process; and so on.

Obviously, if process B has more than k more commands than A, then after process A ends, the constraint $|i-j| \leq k$ will not always hold. So we don't require the $|i-j| \leq k$ hold when $i = m$, instead, we require $i - j \leq k$ to only prevent the case that $i = m \wedge j = i - k - 1$.

Compared to the criterion OSC, FSC has a clearer definition, there has and only has one set that satisfies FSC_k for given k, but there may be a lot of sets satisfy OSC_k for any given $k \leq \min(m, n)$. And the algorithm for generating FSC_k is also simpler and faster than that for OSC_k. What's more, according to the definition, FSC_{k-1} is always a subset of FSC_k, that is not required by the definition of OSC. So we can form a progressively generation as the following paragraph describes.

Obviously, FSC_1, $FSC_2 \backslash FSC_1$, $FSC_3 \backslash FSC_2 \ldots$ form a classification of all the possible traces. In practise, if we think that each command costs similar time, then the traces in FSC_1 are most probably to happen, and then the traces in $FSC_2 \backslash FSC_1$ also have a high possibility than the other traces. When k is very large, we may think that $FSC_k \backslash FSC_{k-1}$ is very unfair and the probability of their occurrent is low. So, if we want to check the traces of the parallel program, we may check FSC_1 at first, and then $FSC_2 \backslash FSC_1$, and so on. In this case, in the circumstance with limited resource, we may get more confident even if we must be finish before checking all possible traces, since we have tried the more possible traces at the beginning.

3. Generating Algorithms for OSC_k

Although the idea of OSC has been proposed for many years, no algorithms of generating traces has been proposed yet. In this section, we will present the algorithms for generating traces of OSC_k. In next section we will introduce the algorithm of generating traces of FSC_k, and perform a theoretical analysis about the amount of traces in FSC_k accordingly.

3.1. The Algorithms of OSC_1

In this subsection we will introduce the algorithm for OSC_1, since it is clearer and much easier to understand.

At first, suppose the two processes have m and n atomic commands, and we can draw a $(m+1) * (n+1)$ state mesh. We need to mark each state by a node with some fields. In **Algorithm 1** we give definitions for the data structure and some basic predicates/operations used in our algorithms.

```
struct Node
    int down, right;
    int value;
bool equal(Node $t_1$, Node $t_2$)
    // whether $t_1$ and $t_2$ have the same elements
bool lessequal(Node $t_1$, Node $t_2$)
    // whether each element of $t_1$ is less than or equal to
    // the corresponding element of $t_2$
bool less(Node $t_1$, Node $t_2$)
    // whether each element of $t_1$ is less than the corresponding element of $t_2$
Node add(Node $t_1$, Node $t_2$)
    // returns a Node whose elements are the summation of what of $t_1$ and $t_2$
Node del_leaf(Node $t$)
    // returns the tree that delete all $t$'s leaf node
```

Algorithm 1: Nodes and its operators

In the Nodes, filed "value" is used to count the number of traces go through this node, "down" denotes the number of traces which come to this node and then go downward, and "right" denotes the number of traces which come to this node and then go rightward. When the algorithm ends, values of the mesh should satisfies the following equations:

$$\begin{array}{lcll} t[i][j].\mathsf{value} & = & t[i][j].\mathsf{down} + t[i][j].\mathsf{right} & \\ t[i][j].\mathsf{value} & = & t[i][j-1].\mathsf{down} + t[i-1][j].\mathsf{right} & \text{if } i > 0 \wedge j > 0 \end{array}$$

Here we use array t to represent all the nodes attaching at each state of the mesh. The first equation is obvious, which denotes that the traces passing this state are either going down or going right. The second equation means that the number of traces passing a state should equal the number of the out-going traces from its two preceding states to this state.

Now we think about the initial values of the nodes. To derive these values, we need an array pcount (for "path counter"), where $\mathsf{pcount}[i][j][k]$ denotes the number of traces start from state (i, j) with length k. For paths of length 0, we take the default value 1 for each state, because we can think there is always and only one path of length 0 passing each state. Based on all the $\mathsf{pcount}[i][j][0]$, we can calculate all $\mathsf{pcount}[i][j][1]$ as follows:

$$\mathsf{pcount}[i][j][1] = \mathsf{pcount}[i][j+1][0] + \mathsf{pcount}[i+1][j][0]$$

For the indexes out of the range, we take the value as 0 for default.

We can take $\mathsf{pcount}[i][j][1]$ as the initial value of $t[i][j].\mathsf{value}$, since OSC_1 must cover all the edges in the mesh, so there must be at least $\mathsf{pcount}[i][j][1]$ traces go through (i, j). For the same reason, we take $\mathsf{pcount}[i+1][j][0]$ and $\mathsf{pcount}[i][j+1][0]$ as the initial value for $t[i][j].\mathsf{right}$ and $t[i][j].\mathsf{down}$, respectively. **Algorithm 2** describes the initialization step, where $\mathsf{pcount}[i][j][k]$ is regarded as default 0 if $(i > m)$ or $(j > n)$ or $(i < 0)$ or $(j < 0)$.

```
void path_initialize(int[][][] pcount)
for each state $(i, j)$ do
    pcount$[i][j][0] \leftarrow 1$
end
for each state $(i, j)$ do
    pcount$[i][j][1] \leftarrow$ pcount$[i+1][j][0]$ + pcount$[i][j+1][0]$
end
void node_initialize(Node[][] $t$, int[][][] pcount)
for each state $(i, j)$ do
    $t[i][j]$.value $\leftarrow$ pcount$[i][j][1]$
    $t[i][j]$.down $\leftarrow$ pcount$[i][j+1][0]$
    $t[i][j]$.right $\leftarrow$ pcount$[i+1][j][0]$
end
```

Algorithm 2: Initializations

After the initialization, we begin to loop on the mesh to make sure that every tree satisfies the constraints mentioned above, as **Algorithm 3** describes. When we found a node violating the constraints, we should increase some of the values to make the constraints true. There are two cases:

- When the number of the traces go into a state is larger, we should increase the number of traces go out the state, and decide whether this new trace go downward or rightward, the idea is to let the proportion of $t[i][j]$.down and $t[i][j]$.right to be closer to the proportion of the initial value of them, i.e. $path\_count[i][j+1][0]$ and $path\_count[i+1][j][0]$.
- When the number of the traces go out a state is larger, we should increase the number of traces go into this state, and decide whether it comes from leftward or upward, according to the relationship of $t[i][j-1]$.down and $t[i-1][j]$.right, we'll just increase the smaller one.

After the algorithm terminates, the whole graph must satisfy the two constraints. Then we can use these trees to generate the trace set of OSC_1. We can use the following method to enumerate the traces:

- We start from $(0, 0)$ and check the node. If $t[0][0]$.down $\neq 0$, let the trace go down, and decrease both $t[0][0]$.value and $t[0][0]$.down by 1; else let the trace go right, and decrease both $t[0][0]$.value and $t[0][0]$.right by 1.
- When we are currently in position (i, j), do the same thing as above. When the process reaches (m, n), we find a complete trace.
- Repeat this process until $t[0][0]$.value $= 0$, which shows that we have enumerate all the traces of OSC_1.

3.2. The Algorithms of OSC_k

The algorithm for general OSC_k is similar to what for OSC_1, but is much longer and complex. So we will just informally describe it here instead of giving

```
void main()
int pcount$[m][n][2]$
Tree $t[m][n]$
$path\_initialize$(pcount)
$node\_initialize(t)$
while the graph has been modified in last loop do
    for each state (i, j) do
        int $t\_in \leftarrow t[i][j-1]$.down $+ t[i-1][j]$.right
        int $t\_out \leftarrow t[i][j]$.value
        if $(i = 0 \land j = 0) \lor (t\_in = t\_out)$ then
            continue
        if $t\_out < t\_in$ then
            while $t\_out < t\_in$ do
                $t\_out{+}{+}$
                $t[i][j]$.value$++$
                if $t[i][j]$.down $*$ pcount$[i+1][j][0] > t[i][j]$.right $*$ pcount$[i][j+1][0]$
                then
                    $t[i][j]$.right$++$
                else
                    $t[i][j]$.down$++$
            end
        else
            while $t\_out > t\_in$ do
                $t\_in{+}{+}$
                if $t[i][j-1]$.down $< t[i-1][j]$.right then
                    $t[i][j-1]$.down$++$
                else
                    $t[i-1][j]$.right$++$
            end
    end
end
```

Algorithm 3: update the binary trees

the whole algorithm. The source code of generating the trace set OSC_k can be found in section 5.

In the case of OSC_k, we need a structure full binary tree(depth $= k$) instead of the full binary tree(depth $= 1$) defined in the previous section. For a tree T, $T.down$ and $T.right$ will be the subtrees of T now. The values of the tree have similar meanings with the case of OSC_1, for example $t[i][j].right.down.value$ means how many traces go through node (i, j) and then go rightward and then downward.

In the initialization step, for a state (i, j), we'll set the initial value of $t[i][j].value$ as $path\_count[i][j][k]$, since OSC_k must cover all the k-length traces on the graph, and $path\_count[i][j][k]$ means the amount of k-length traces start from (i, j), so there must be at least $path\_count[i][j][k]$ go through state (i, j). For the same reason, we can take $path\_count[i+1][j][k-1]$ as the initial value of $t[i][j].right.value$, and take $path\_count[i][j+1][k-1]$ as the initial value of

$t[i][j].down.value$, and so on.

And when the algorithm ends, the values should satisfies the following constraints:

$$T.value = T.down.value + T.right.value$$
$$\text{(here } T \text{ may be } t[i][j] \text{ or any subtree of } t[i][j])$$
$$\text{equal}(\text{add}(t[i][j-1].down,\ t[i-1][j].right),\ \text{del\_leaf}(t[i][j]))$$
$$\text{(here } !(i = 0 \wedge j = 0))$$

We loop through the whole graph, when we found a tree that doesn't satisfy these constraints, we should increase some values to hold these constraints. When a value are increased, we also need to decide which of its subtree to increase, the method is the same with precious section.

When the algorithm terminates, we can use the following method to find traces:

At first, the starting point of the trace is $(0, 0)$. We can check the tree $t[0][0]$ and find a leaf node, of which the value is not zero, suppose this leaf is $t[0][0].right.down$. So we can determine that the first two step of this trace is $right \rightarrow down$, and we should let all these values $t[0][0].value$, $t[0][0].right.value$, $t[0][0].right.down.value$ minus 1. Then we can go rightward, and find a non-zero son of $t[1][0].down$, to determine the third step of this trace, and so on.

What's more, using this algorithm, it will ensure that the generated OSC_{k-1} is always a subset of OSC_k. So that we can use the criteria $OSC_1 \sim OSC_k$ to process a progressively analyzing of the parallel processes. Since these path sets satisfy $OSC_1 \subseteq OSC_2 \subseteq ... \subseteq OSC_k$, we could firstly use OSC_1 to analyze the program, which will cover all the edges of the state graph, and then use $OSC_2 \backslash OSC_1$ to analyze, which will ensures a stronger quality together with OSC_1, and so on. Each step will cost more time, and the user can determine the time to stop by himself if he think the current k is large enough to convince him.

At last we briefly analyze the time complexity of the algorithm. For given parallel processes has m and n commands, the amount of traces in OSC_k can be determined by m, n, k, although it's hard to express the formula between these values. Practically, the amount is about $O(m * n * 2^k)$ when k is not very large (compared to m and n).

Suppose that there are N traces altogether in the trace set OSC_k. The time complexity of the algorithm is obviously $O(N * m * n * 2^k)$, since it loops no more than N times, and in each loop it will go through $m * n$ trees, which has $2^{k+1} - 1$ nodes.

In practice, the implementation of this algorithms is able to generate more than 100 long traces in 1 second, so we can believe that the time in generating traces can be ignored, comparing to the time in analyzing those traces.

3.3. Correctness and Termination of the Algorithms

In this section, we will only prove the correctness and termination of this algorithm in criteria OSC_1 since we've formally defined the algorithms in OSC_1, and the proving in the case of OSC_k is similar with this one.

At first we'll prove its correctness. The condition of its termination is the whole graph hasn't been modified in a loop, i.e. the graph satisfies the following constraints C_1 and C_2:

C_1: For each state (i, j), if $!(i = 0 \wedge j = 0)$, then

$$t[i][j].value = t[i][j-1].down + t[i-1][j].right$$

C_2: And for each state (i, j),

$$t[i][j].value = t[i][j].down + t[i][j].right$$

The first constraint just means the amount of traces get into state (i, j) must equals the amount of traces get out of state (i, j), except the initial state $(0, 0)$. So when $t[0][0].value > 0$, according to C_2, at least one of $t[i][j].down$ and $t[i][j].right$ should be bigger than 0. So we can always choose one direction to move and reach another state (i, j). According to C_1, there are at least 1 trace get into state (i, j), so there should be at least 1 trace get out of (i, j), i.e. $t[i][j].value \geq 1$. For the same reason, we can always choose one direction to move and reach another state (i, j), and so on. At last this trace must be able to reach the final state (m, n).

After we find such a trace and reduce some of the values in the graph, we can see C_1 and C_2 still holds. So we can go on and get another trace until the values on the whole graph are all 0.

It's also easy to prove these traces cover all the edges on the graph. Notice that the initial value of each state, if state (i, j) has an edge downward, then the initial value of $t[i][j].down$ is 1. During the algorithm, the values could only increase, but no decrease. So when the algorithm terminate, the value $t[i][j].down \geq 1$, i.e. at least 1 trace will go through the node (i, j) and go downward. Using the same method we can prove that all the edges has been covered by trace set OSC_1.

So we've proved the correctness of the algorithm, then we'll proof its termination. We can define a trace set OSC_{m+n}, which covers all the $(m+n)$-length trace in the graph, i.e. it enumerates all the possible traces. Obviously the values of any OSC_k cannot exceed that of OSC_{m+n}, so it is an upper bound of OSC_k and ensures the termination of OSC_k.

4. The Algorithms of FSC_k

In this section, we will introduce the algorithm of generating the trace set $FSC_k \backslash FSC_{k-1}$. In this section, we always suppose the two processes have m and n commands, and $m \leq n$.

At first, let's introduce some notations: for a given k, we say a state (i, j) is an inner state iif

$$(|i - j| < k) \vee (i = m \wedge j \geq i + k);$$

and state (i, j) is an boundary state iif

$$(|i - j| = k) \wedge (j < i + k);$$

and all the other states, i.e.

$$(i < m \wedge |i - j| > k) \vee (i = m \wedge j < i - k),$$

are called outer state.

```
void main()
for each boundary state $(i,j)$ do
    find_inner_trace($i,j,k$); // find traces from $(0,0)$ to $(i,j)$, which only
    contains inner states besides $(i,j)$
    find_trace($m,n,i,j,k$); // find traces from $(i,j)$ to $(m,n)$, which only
    contains inner states and boundary states
    for trace $a \in$ the first trace set do
        for trace $b \in$ the second trace set do
            output(trace_append($a,b$))
        end
    end
end
```

Algorithm 4: enumerate traces of $FSC_k \backslash FSC_{k-1}$

We can see that, the union set of inner states and boundary states is just the set of states that $t \in FSC_k$ can pass; and the set of inner states is just the set of states that $t \in FSC_{k-1}$ can pass.

So, every trace T in $FSC_k \backslash FSC_{k-1}$ must satisfies:

$$C_1 : \forall (i,j) \in T, (i,j) \text{ is an inner state or boundary state}$$
$$C_2 : \exists (i,j) \in T, (i,j) \text{ is a boundary state}$$

For each boundary state (i,j), we can enumerate the traces which pass (i,j) before passing any other boundary states. This task can be reduced into two tasks: (1) enumerate the sub-traces from $(0,0)$ to (i,j), and all the states (besides (i,j)) are inner states; (2) enumerate the sub-traces from (i,j) to (m,n), and each state is either an inner state or a boundary state.

We can easily use searching to process the two enumerations. So we've enumerated the traces which pass (i,j) before passing any other boundary states. For every boundary state (i,j) we can do so, and then we've enumerated all the traces of $FSC_k \backslash FSC_{k-1}$.

4.1. The Amount of traces in FSC_k

In this section, we will analysis the amount of traces in each set FSC_k.

A trace belongs to the trace set FSC_k iff all the states on it are either an inner state or a boundary state. What's more, a trace belongs to $FSC_k \backslash FSC_{k-1}$ iif it belongs to FSC_k and at least one state on it is a boundary state.

When the processed have m and n commands, we notate the set of traces as $Trace(m,n)$. And then we define three subset of $Trace(m,n)$: $L_k(m,n)$, $R_k(m,n)$ and $LR_k(m,n)$ (we will omit the parameter and use L_k, R_k, LR_k if it will not lead to ambiguity).

We say a trace $t \in L_k$ iif there exists a state (i,j) of t, which satisfies $i - j > k$. Also, we say $t \in R_k$ iif there exists a state (i,j) of t, which satisfies $i < m \wedge j - i > k$. And $t \in LR_k$ iif $t \in L_k \wedge t \in R_k$.

Then, let's calculate the amount of traces in FSC_k. Using the above notations, the amount equals to $|Trace(m,n)| - |L_k| - |R_k| + |LR_k|$, according to the Inclusion-Exclusion Principle. Here $|Trace(m,n)|$ obviously equals C_{m+n}^{m}.

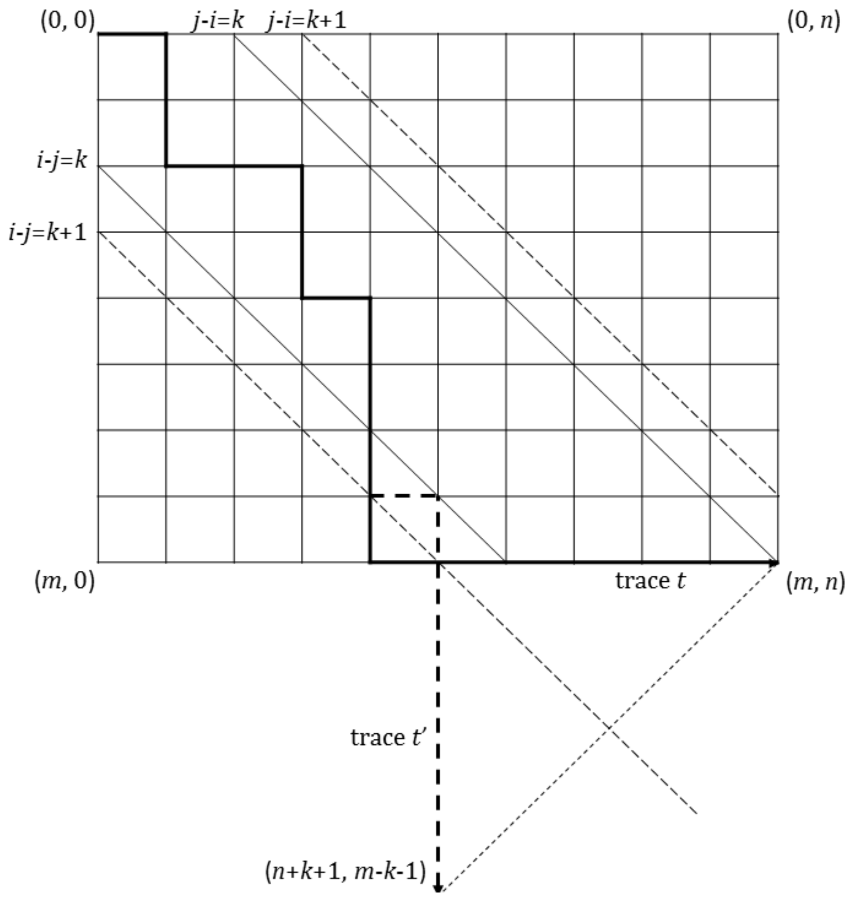

Figure 1: trace t in L_k, and t'

Suppose $m \leq n$, then we calculate the three part $|L_k|, |R_k|$ and $|LR_k|$ separately.

4.1.1. L_k

We show the amount of traces in the set L_k in this section. We'll discuss the problem in two cases, $k \geq m$ or $k < m$.

If $k \geq m$, obviously $i \leq m$ and $i - j > k$ cannot be satisfied at the same time. In this case $L_k = \emptyset$.

Then we'll mainly discuss the case $k < m$. We can notate each trace as a sequence $c_1; c_2; ...; c_{m+n}$, in which every c_i can choose two values a or b, a represents the first process executes a command, and b represents the second process executes a command.

Since each trace t in L_k includes a state (i, j) that $i - j > k$. Suppose the first state in the sequence which satisfies this condition is (i_0, j_0), then obviously $i_0 - j_0 = k + 1$. Consider about the following function:

$$f : L_k \rightarrow Trace(n+k+1, m-k-1)$$

For each command after (i_0, j_0) in t, if it is a then we change it to b, if it is b then we change it to a, so that we get a new trace t' in set $Trace(n+k+1, m-k-1)$.

Let's see the reason that $t' \in Trace(n+k+1, m-k-1)$. Since there are m commands of a and n commands of b in t, and before (i_0, j_0) there are $x+k+1$ commands of a and x commands of b in t, after (i_0, j_0) there are $m-(x+k+1)$ commands of a and $n-x$ commands of b in t, i.e. there are $n-x$ commands of a and $m-(x+k+1)$ commands of b in t'. So, in t' there are $n+k+1$ commands of a and $m-k-1$ commands of b in total, i.e. $t' \in Trace(n+k+1, m-k-1)$.

Using the same method, for each trace $t' \in Trace(n+k+1, m-k-1)$, we could find the first state (i_0, j_0) satisfies $i_0 - j_0 = k+1$, and convert all the commands after (i_0, j_0), then we can get a unique trace $t \in L_k$, and $f(t) = t'$. Thus the function f is a bijection, and we know that $|L_k| = |Trace(n+k+1, m-k-1)|$, i.e. $|L_k| = C_{m+n}^{n+k+1}$.

Figure 1 shows an example, here $k = 2$. And the states between the two lines $i-j=k$ and $j-i=k$ are the inner states, the states on the two lines are boundary states, and the states out of the two lines are outer states. Here t is a trace in L_k, and $t' = f(t)$. In the figure we can see, after trace t reaches the line $i-j=k+1$, t and t' are symmetrical on this line. And from this figure we may see f is a bijection between the two sets L_k and $Trace(n+k+1, m-k-1)$ obviously.

4.1.2. R_k

Similar to the previous part, we show the calculation of R_k in this section. We'll discuss the problem in two cases, $k > n-m$ and $k \leq n-m$.

If $k > n-m$ holds, the calculation is similar to the previous section. Since each trace t in R_k includes a state (i,j) that $i < m \wedge j - i > k$. Suppose the first state in the sequence which satisfies this condition is (i_0, j_0), then obviously $j_0 - i_0 = k+1$. Then we can define such a function:

$$f' : R_k \rightarrow Trace(n-k-1, m+k+1)$$

For each command after (i_0, j_0) in t, if it is a then we change it to b, if it is b then we change it to a, so that we get a new trace t' in set $Trace(n-k-1, m+k+1)$.

f' is also a bijection, we omit the proof since it is similar to the previous section. So in this case, $|R_k| = C_{m+n}^{n-k-1}$.

Then let's consider the case that $k \leq n-m$. Now $|R_k|$ is hard to calculate, instead we'll show the cardinal of $Trace(m,n) \backslash R_k$. Let's define a bijection:

$$extend : (Trace(m, m+k) \backslash R_k(m, m+k)) \rightarrow (Trace(m,n) \backslash R_k(m,n))$$

For a trace $t \in (Trace(m, m+k) \backslash R_k(m, m+k))$, adds $n-m-k$ commands of b at the end of t, so we get a new trace t' in set $(Trace(m,n) \backslash R_k(m,n))$.

Thus $|Trace(m, m+k) \backslash R_k(m, m+k)| = |Trace(m,n) \backslash R_k(m,n)|$. Then we can calculate $|R_k(m, m+k)|$ by defining the function f'':

$$f'' : R_k(m, m+k) \rightarrow Trace(m-1, m+k+1)$$

And f'' is doing the same thing as f and f', it's also a bijection, i.e. $|R_k(m, m+k)| = C_{2m+k}^{m-1}$. Since $|Trace(m, m+k)| - |R_k(m, m+k)| = |Trace(m,n)| -$

$|R_k(m,n)|$, we can derive the cardinal of $R_k(m,n)$: $|R_k| = C^m_{m+n} - (C^m_{2m+k} - C^{m-1}_{2m+k})$.

4.1.3. LR_k

The last step is to calculate the $|LR_k|$, this one is more difficult to derive. Since it's much lesser than $|L_k|$ and $|R_k|$, we'll just give an estimation of it.

We're going to prove that:

$$\frac{|LR_k|}{C^m_{m+n}} \leq \frac{|R_k|}{C^m_{m+n}} \cdot \frac{|L_k|}{C^m_{m+n}}.$$

Since each trace t in L_k satisfies that there exists a state (i,j) of t, which satisfies $i - j > k$. We can divide the set L_k into several disjoint subsets:

$L^0_k \triangleq \{t \in L_k | (k+1, 0) \text{ is the first outer state in } t\}$

$L^1_k \triangleq \{t \in L_k | (k+2, 1) \text{ is the first outer state in } t\}$

$L^i_k \triangleq \{t \in L_k | (k+i+1, i) \text{ is the first outer state in } t\}$

and so on. Also, we can divide the set R_k and LR_k into several disjoint subsets:

$R^0_k \triangleq \{t \in R_k | (0, k+1) \text{ is the first outer state in } t\}$

$R^1_k \triangleq \{t \in R_k | (1, k+2) \text{ is the first outer state in } t\}$

$R^i_k \triangleq \{t \in R_k | (i, k+i+1) \text{ is the first outer state in } t\}$

$LR^0_k \triangleq \{t \in LR_k | \text{ state } (0, k+1) \in t\}$

$LR^1_k \triangleq \{t \in LR_k | \text{ state } (1, k+2) \in t\} \backslash LR^0_k$

$LR^i_k \triangleq \{t \in LR_k | \text{ state } (i, k+i+1) \in t\} \backslash \bigcup_{j=0}^{i-1} LR^j_k$

and so on. We are going to prove that:

Theorem 1: $\forall i, \frac{|LR^i_k|}{|R^i_k|} = \frac{|L_{2k+1}(m-i, n-k-i-1)|}{C^{m-i}_{m+n-k-2i-1}}$

Theorem 2: $\forall i, \frac{|L_{2k+1}(m-i, n-k-i-1)|}{C^{m-i}_{m+n-k-2i-1}} \leq \frac{|L_k(m,n)|}{C^m_{m+n}}$

Here **Theorem 1** is obvious, since a trace $t \in Trace(m,n)$ reaches $(i, k+i+1)$ and also satisfies $t \in R_k(m,n)$, is the same as another trace $t' \in Trace(m-i, n-k-i-1)$ starts from $(0,0)$ and satisfies $t' \in L_{2k+1}(m-i, n-k-i-1)$.

For **Theorem 2**, since we've solved the value of $L_k(m,n)$, just substitute it in **Theorem 2**:

$$\frac{|L_{2k+1}(m-i, n-k-i-1)|}{C^{m-i}_{m+n-k-2i-1}} \leq \frac{|L_k(m,n)|}{C^m_{m+n}}$$

$$\Leftrightarrow \frac{C^{n+k-i+1}_{m+n-k-2i-1}}{C^{m-i}_{m+n-k-2i-1}} \leq \frac{C^{n+k+1}_{m+n}}{C^n_{m+n}}$$

$$\Leftrightarrow \frac{(m-i)!(n-k-i-1)!}{(m-2k-i-2)!(n+k-i+1)!} \leq \frac{m!n!}{(m-k-1)!(n+k+1)!}$$

$$\Leftrightarrow \frac{(m-i)...(m-2k-i-1)}{(n+k-i+1)...(n-k-i)} \leq \frac{m...(m-k)}{(n+k+1)...(n+1)}$$

Since $m < n+k+1$, we know that

$$\frac{(m-i)}{(n+k-i+1)} \leq \frac{m}{(n+k+1)},$$

and so on. Thus

$$\frac{(m-i)...(m-k-i)}{(n+k-i+1)...(n-i+1)} \leq \frac{m...(m-k)}{(n+k+1)...(n+1)}.$$

And the other part of the left side

$$\frac{(m-k-i-1)...(m-2k-i-1)}{(n-i)...(n-k-i)} \leq 1,$$

which is to say

$$\frac{(m-i)\dots(m-k-i)}{(n+k-i+1)\dots(n-i+1)}\frac{(m-k-i-1)\dots(m-2k-i-1)}{(n-i)\dots(n-k-i)} \leq \frac{m\dots(m-k)}{(n+k+1)\dots(n+1)},$$

that's equivalent to **Theorem 2**. Thus we've successfully proved the **Theorem 2**. That's to say,

$$\frac{|LR_k^i|}{|R_k^i|} \leq \frac{|L_k|}{C_{m+n}^m},$$

i.e.

$$|LR_k^i| \leq |R_k^i| \cdot \frac{|L_k|}{C_{m+n}^m}.$$

So we know that

$$\sum_i |LR_k^i| \leq \sum_i |R_k^i| \cdot \frac{|L_k|}{C_{m+n}^m}$$

$$\Leftrightarrow |LR_k| \leq \frac{|R_k|\cdot|L_k|}{C_{m+n}^m}$$

$$\Leftrightarrow \frac{|LR_k|}{C_{m+n}^m} \leq \frac{|R_k|}{C_{m+n}^m} \cdot \frac{|L_k|}{C_{m+n}^m}.$$

After all, we've proved that:

$$|Trace(m,n)| - |L_k| - |R_k| \leq |FS_k| = |Trace(m,n)| - |L_k| - |R_k| + |LR_k|$$

$$\Rightarrow C_{m+n}^m - |L_k| - |R_k| \leq |FS_k| \leq C_{m+n}^m - |L_k| - |R_k| + \frac{|R_k|}{C_{m+n}^m} \cdot \frac{|L_k|}{C_{m+n}^m}$$

$$\Rightarrow C_{m+n}^m - |L_k| - |R_k| \leq |FS_k| \leq C_{m+n}^m \cdot (1 - \frac{|L_k|}{C_{m+n}^m}) \cdot (1 - \frac{|R_k|}{C_{m+n}^m})$$

After we substitute the value of $|L_k|$ and $|R_k|$ into this formula, we can derive the estimation of $|FS_k|$.

5. Implementation

We implement the algorithms mentioned above in Java. For the given value m, n, k, it can generate all the traces of OSC_k and FSC_k, where the two processes has m and n commands.

The form of output is:

$$A_1B_1B_2A_2$$

Which shows that the first process execute one command, and then the second process execute one command, then the second process execute another, at last the first process execute one command.

And the full source code of our trace generating tool is available at: http://sourceforge.net/projects/cdldeveloper/.

6. Conclusion

In the paper we introduce two analyzing criteria OSC_k and FSC_k, and give algorithms that can generate all the traces of them automatically, and then we prove the correctness and termination of the algorithms of OSC.

And we described how to use these criteria to check the behavior of parallel processes, we can either set a big enough k (according to the processes) and use the criterion OSC_k to check it; or use OSC_1, then $OSC_2 \backslash OSC_1$, then $OSC_3 \backslash OSC_2$ and so on to classify the traces, and perform a progressive checking. And it's the same for criteria FSC.

Now the algorithms is only applicable on those parallel processes without control, else the state graph is not just a mesh, and will become more large and complex. In the future, we'll try to apply similar method on checking more complex processes, e.g. including choice and loop.

What's more, if there are 3 processes, we can generate the trace set OSC_k of the first two processes, and use this trace set together with the third process to generate OSC_k of these three processes. But this is not associative, i.e. use OSC_k of the last two processes and the first process will generate a different OSC_k set of the three processes. Thus the classifications in this paper aren't applicable to the case of more than 2 processes. We are also on working to apply this method to more processes.

References

[1] Per Brinch Hansen, "The Origin of Concurrent Programming: From Semaphores to Remote Procedure Calls", 2002.

[2] Eisuke Itoh, Yutaka Kawaguchi, Zengo Furukawa, and Kazuo Ushijima, "Ordered Sequence Testing Criteria for Concurrent Programs and the Support Tool", in *SEC 1994*, pp. 236-245.

[3] Eisuke Itoh, Zengo Furukawa, and Kazuo Ushijima, "A prototype of a concurrent behavior monitoring tool for testing of concurrent programs", in *SEC 1996*, pp. 345-354.

[4] William E. Howden, "Reliability of the Path Analysis Testing Strategy", in *Software Engineering, Sep. 1976*, pp. 208-215.