Coping with Nondeterminism in Network Protocol Testing

R. E. Miller[1], D.-L. Chen[2], D. Lee[3], and R. Hao[4]

[1] Department of Computer Science, University of Maryland
[2] Department of Computer Science, Tsinghua University, China
[3] Department of Computer Science and Engineering, Ohio State University
[4] Bell Labs Research China, Lucent Technologies

Abstract. Given a nondeterministic protocol specification, we want to determine the deterministic implementation under test with a conformance of trace inclusion in the specification. We identify them using both active and passive testing. Four cases are studied with experiments on Internet protocols. In the first two cases, the implementation machine is a derived machine of the specification. In the third case, the implementation machine is a derived machine of the k-way expansion of the specification machine. The fourth case deals with the general case of nondeterministic machines.

1 Introduction

Network protocols are often partially specified, the unspecified inputs may be ignored, or cause an error message [1]. The choices depend on the design of an implementation. Often network protocols contain optional requirements, which are specified by "MAY" statements in many RFCs. These two cases can be regarded as options in network protocol specifications, providing certain flexibility to protocol implementations. Due to the options, a protocol cannot be modelled by a deterministic finite state machine (DFSM). The common approach is to use a non-deterministic finite state machine (NFSM) to model these protocols instead. This situation has complicated the protocol testing operations. There are several studies on testing NFSMs, both active testing [2], [3], [4], [5] and passive testing [6].

When a vendor implements a protocol, it may implement some of the options and discard the others, or implement all the options with configuration parameters and let the user make the decision. Hence, given a protocol, there may be different deterministic implementations that conform to the specification. Often we need to identify the deterministic implementation of the object system. For example, there are several TCP variants deployed on the Internet. [7] provided test scenarios to examine these deployments. In general, it is a machine identification problem, however, it is rather complex.

In this paper, we assume the specification machine A is an NFSM and the implementation machine B which conforms to A is a DFSM. We study the

problem of identifying the DFSM implementation, given its NFSM specification. The specification machine is assumed to have n states, p inputs and q outputs. For such a machine, there are totally $(qn)^{pn}/n!$ candidate machines. The problem of identifying B is difficult, because the distinguishing sequences for a NFSM may not exist, when there is a distinguishing sequence, it can be of exponential length [8]. Due to the difficulty in the general case, some work [3], [5] focus on the special case of *Observable* NFSM (ONFSM).

Our approach is different from the exist works in several aspects. First we provide both passive testing and active testing algorithms for this machine identification problem. Passive testing/monitoring [6], [9] has been a important area in network analysis. Our passive testing algorithm can be used to identify new features of the protocols, like routing protocols, HTTP and TCP. Second we study the cases of both ONFSM and general NFSM. According to the NFSM is observable or not and the implementation is a subautomaton or not, there are four cases. We study the complexity of each case and provide algorithms when it is feasible. We propose the concept *derived machines* to study the relationship between implementation and specification in the first three cases. Third we study the nondeterminism in Internet protocols and propose algorithms to solve the identification problem. Third we use simulation to show the efficiencies of the algorithms.

In Section 2 we provide the basic concepts for NFSM. We propose the definition of *derived machines* when a DFSM is a subautomaton of an NFSM specification. Section 3, 4, 5 and 6 study the four cases, respectively. We discuss both active and passive testing approaches with experiments on Internet protocols. Conclusion is drawn on Section 7.

2 Preliminaries

The following definitions and properties about NFSMs are based on [8].

Definition 1. *A* nondeterministic *finite state machine (NFSM) M is a 4-tuple $M = (I, O, S, h)$ where I is a finite set of input symbols, O is a finite set of output symbols, S is a finite set of states, h is the* transition function*: $S \times I \rightarrow \mathcal{P}(S \times O)$, where $\mathcal{P}$ denotes the power set operator.*

Definition 2. *The transition function f is defined for $s, s' \in S, x \in I, y \in O$ as follows:*

$$f(s, x) \stackrel{\text{def}}{=} \{s' \mid \exists y\ [s', y] \in h(s, x)\}.$$

The conditional transition function h_y is defined for $s, s' \in S, x \in I, y \in O$ as follows:

$$h_y(s, x) \stackrel{\text{def}}{=} \{s' \mid [s', y] \in h(s, x)\}.$$

Definition 3. *A NFSM is observable if for all $s \in S, x \in I, y \in O$, we have $Card(h_y(s, x)) \leq 1$, where $Card(Z)$ denotes the cardinal number of the set Z.*

In this paper, the specification is assumed to be reduced[8]. Our conformance relation is defined by *trace inclusion* [10]. Let s_A, s_B be the initial states of machine A, B respectively. The *trace* of machine M from its initial state s_M is denoted by $T(M, s_M)$.

Definition 4. *An implementation machine (B, s_B) conforms to a specification machine (A, s_A) if and only if $T(B, s_B) \subseteq T(A, s_A)$.*

We use *derived machine* when an implementation machine is a *subautomaton* of the specification machine. A derived machine is required to be connected and deterministic.

Definition 5. *Given a NFSM $A = (I, O, S, h)$, an implementation machine $B = (I', O', S', h')$ is called a* derived machine *of A if B is deterministic, connected, and a subautomaton of A.*

Since a derived machine B is a subautomaton of the specification machine A, its trace is included in the trace of A; hence it conforms to A. Note when B conforms to A, B is not required to be a derived machine of A, which is discussed in Section 5. There relationship is shown in Figure 1.

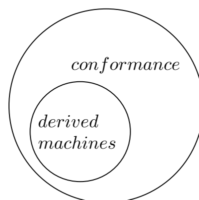


Fig. 1. Derived Machine and Conformance

We now proceed to investigate four different cases of determining the implementations from active and passive testing.

3 Case 1: Derived Machine of Observable NFSM

Network protocol specifications may allow different responses according to an input. This kind of nondeterminism is *observable* and can be judged by the appearance of the output.

3.1 Active Testing Approach

Usually network protocol systems have *reset capability*, an input symbol, denoted by r, leading the machine back to its initial state, and it can be realized by a restart of the device. Obviously, taking an i/o pair as a symbol, ONFSM can

be regarded as a deterministic finite automaton. To identify the implementation machine, we have to traverse the transitions with the same input from a state. The following procedure will construct the implementation machine.

begin
1. $B = (I, O, S = \{s_0\}, h = \{r\})$; /* s_0: initial state, r: reset */
2. **while** ($\exists s_i \in S, a \in I$, $h(s_i, a)$ not specified in B)
3. trap B in s_i; /* usually with a sequence started by r */
4. send input a to B, assume output b;
5. find the unique transition $h(s_i, a) = (s_j, b)$ in the transition table of A;
6. add $h(s_i, a) = (s_j, b)$ to h;
7. **if** ($s_j \notin S$) /* a new state explored in B */
8. add s_j to S;

end

In step 5, there is a unique transition $h(s_i, a) = (s_j, b)$ in the transition table of A because A is observable and B conforms to A. If we use a breadth-first search (BFS) strategy to explore the transitions from the states, the ith explored state is reachable in i steps with prefix r. Assume the ith state has k_i undecided inputs, $k_i \leq p$. The complexity of the construction is $C \leq \sum_{i=1}^{n} i * k_i = O(p \sum_{i=1}^{n} i) = O(pn^2)$.

Theorem 1. *If A is an ONFSM and B is a derived machine of A, the construction procedure takes time $O(pn^2)$ to build the derived machine .*

3.2 Passive Testing Approach

The passive testing approach is divided into a *homing* phase and an *identification* phase, similar to [6]. The procedure traces the current states of the specification NFSM A using the observed I/O sequence from the implementation machine B. A *passive testing map* records the possible states and their related transitions during a passive testing.

Definition 6. *A passive testing map is a directed graph $G = (V, E)$. For an observed sequence $e_1, e_2, \ldots, e_k$, $V = L_0 \cup L_1 \cup \ldots \cup L_k$, where L_j records the possible states after event e_j, a node $v_{ij} = (s_i, j) \in L_j$ if s_i is a possible state after event e_j. $E = tr(0, 1) \cup tr(1, 2) \cup \ldots \cup tr(k-1, k)$, where $tr(j, j+1)$ records the possible transitions from L_j to L_{j+1}. If $(s_i, j) \in L_j$, and there is a transition $s_i \xrightarrow{e_j} s_v$ in the specification, then $(s_v, j+1) \in L_{j+1}$ and $\{(s_i, j) \xrightarrow{e_j} (s_v, j+1)\} \in tr(j, j+1)$.*

Figure 2 gives an example of passive testing map. We use the following notations in our passive testing algorithms:

L^-, L^+	- the level before and after level L in state tracing
$e_L = x_L/y_L$	- the input/output pair at level L
C	- the current state set (C^- is the level before the current level)

If a state in the specification can trigger multiple transitions upon an input, it is a *branching state* for this input. A derived machine selects from the candidate transitions.

ON-LINE CHECKING - ALGORITHM I
input: ONFSM A and observed sequence
output: fault_detected or machine B

begin
1. $B := A$; /* B is derived from A by deleting transitions from the branching states */
2. $C := \{s_0, s_1, ..., s_{n-1}\}$; /* the initial possible state set */
3. **while** ($|C| > 1$ & $next(x/y)$) /* the homing phase */
4. $C := h_y(C, x)$; /* state tracing */
5. record the transitions from C^- to C in $tr(C^-, C)$; /* for backtracking */
6. $L := C^-$; /* the current level C is a singleton */
7. **do** /* the backtracking phase */
8. remove states from L with no outgoing transition in $tr(L, L^+)$;
9. remove transitions leading to the removed states from $tr(L^-, L)$;
10. **if** ($|L| \equiv 1$)
11. remove transitions $\{(s, s', x_L/y')|s \in C, y' \neq y_L\}$ from B;
12. **while** ($|L| \equiv 1$); /* backtrack to decide the past before C */
13. **while** ($|C| \equiv 1$ & $next(x/y)$) /* the forward phase */
14. **if** ($f(C, x) > 1$) /* branching state for x */
15. remove transitions $\{(s, s', x/y')|s \in C, y' \neq y\}$ from B;
16. **if** (B becomes deterministic) /* all the branchings are decided */
17. return B;
18. $C := h_y(C, x)$; /* state tracing */
19. **if** ($|C| \equiv 0$) /* fault detected */
20. **return** fault_detected;
21. **if** ($next(x/y) \equiv null$) /* passive testing ends */
22. **return** B; /* B is not decided yet */

end

The algorithm is mainly composed of three phases, a homing phase (line 2-5), a backward tracking phase (line 6-12), and a forward tracking phase (line 13-18).

We use an example in Figure 2 to illustrate these phases. Assume the current state set is $\{s_1, s_2\}$ when a sequence a/x b/y c/z is observed. *Homing* is reached at the observation of b/y and the current state set is $\{s_6\}$. *Backtracking* removes $\{s_5, s_2\}$ from the levels. The transition $s_1 \overset{a/y}{\to} s_4$ is removed from B. The forward checking of c/z removes $s_6 \overset{c/y}{\to} s_9$ from B.

In Algorithm I, since back-tracking is triggered by homing, a node in the passive testing map can be back-tracked at most once. Hence the time complexity of this algorithm is $O(L)$, where L is the length of the test sequence. Because the passive testing map is recorded for backtracking, it has at most $n * L$ nodes .

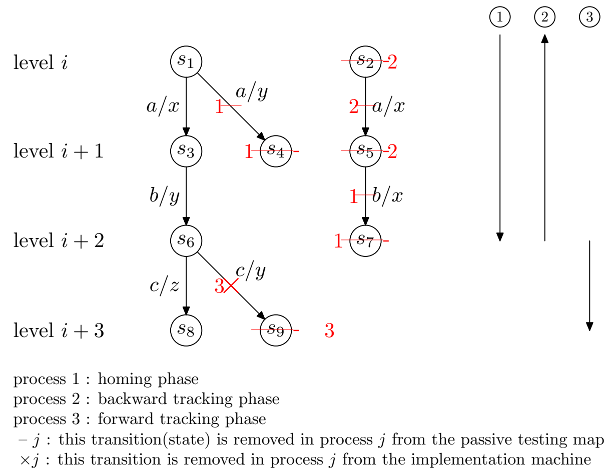


Fig. 2. The passive testing map

Theorem 2. *If A is an ONFSM and B is a derived machine of A, Algorithm I is correct. Its time complexity is $O(L)$ and its space complexity is $O(n * L)$, where L is the length of the test sequence.*

3.3 An Example from RIP

In the Routing Information Protocol (RIP) [11], "split horizon" is a scheme for avoiding problems caused by including routes in updates sent to the gateway from which they were learned. There are two types of split horizon. The "simple split horizon" scheme omits routes learned from one neighbor in updates sent to that neighbor. While "split horizon with poisoned reverse" includes such routes in updates, but sets their metrics to *infinity*. In Fig.3, the transition from $S1$ to $S2$ represents "simple split horizon" and the transition from $S1$ to $S3$ represents "split horizon with poisoned reverse". The two transitions are exclusive.

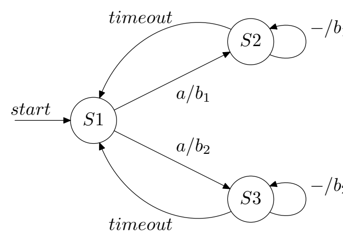


Fig. 3. split horizon in RIP

To identify the implementation machine, in active testing, the test sequence is $r.a$. In passive testing, the derived machine can be decided at the observed output b_1 or b_2. We have tried both active testing and passive testing to identify the RIP implementation. Most vendors set "simple split horizon" as the default configuration; but Cisco (with IOS version 12) only supports "split horizon with poisoned reverse".

4 Case 2: Derived Machine of NFSM

In this section we study the case when the specification A is a general NFSM (including non-observable transitions) and the implementation B is a derived machine of A. In active testing, two approaches, machine enumeration and machine construction, are proposed to solve the machine identification problem. In passive testing, *backtracking* is used to eliminate the unselected transitions.

4.1 Active Testing Approaches

In this case, it is *NP-hard* to determine the implementation machines:

Theorem 3. *Given NFSM A, B is a derived machine of A, the problem of deciding B is NP-hard.*

Deciding Hamiltonian Path can be reduced to deciding B out of A, thus the algorithm is NP-hard. The proof is given in [12] . □

Two methods are proposed here to identify the derived machine under test. The first method is on-line machine enumeration. Candidate machines are enumerated on-line and then use cross verification to separate the candidate machines. The second method is on-line machine construction using distinguishing sequences. We want to reduce the number of candidate machines by eliminating the inconsistent transitions with distinguishing sequences. The first method is "generate then distinguish", while the second is "distinguish when generating". We will compare their efficiencies by simulation experiment.

4.1.1 On-line Machine Enumeration

Commonly the number of derived machines is quite large with nondeterministic transitions. On-line machine enumeration dose not generate all the derived machines. Instead it constructs machines on-the-fly. It removes inconsistent transitions according to the output of machine B.

On-line Machine Enumeration

queue Q contains machines to explore
begin
1. Insert A into Q; /*initialization */
2. **while** $Q \not\equiv \phi$
3. take a machine M from Q;

```
4.        for each unexplored state s in M;
5.            for each input i
6.                get the last y from machine B with input prefix(s).i;
7.                remove transitions s × i/y' → s' (y' ≠ y) from M;
8.                if there is no transition with output y from s × i
9.                    goto 2;   /*for another candidate machine*/
10.               if there exists multiple transitions with output y from s × i
11.                   for each transition t = s × i/y → s'
12.                       M' clones M, M selects t and removes others transitions with s × i;
13                        Insert M' to Q;
14.                   goto 3;   /*for another candidate machine*/
15.           mark state s as explored;
16.       print M;   /* M is a derived machine */
17. apply cross verification to identify the derived machine;
end
```

On-line enumeration generates a set of candidate deterministic machines. *Cross verification* [13] is applied to rule out the wrong ones(line 17).

If the specification has l nonobservable branching points of degree $r_1, r_2, ..., r_l$, there is at most $K = \prod_{i=1}^{l} r_i$ candidate machines generated from the enumeration. At each branching point, a sequence with length less than n is applied to machine B to get the corresponding output. It is known that the minimization of an DFSM is $O(pn \log n)$ [14], where p is the number of input symbols. The standardization takes time $O(pn)$ [15]. It takes $O(pn)$ to determine if two machines are isomorphic or not. If the two machines are not isomorphic, it takes $O(pn)$ to find a distinguishing sequence with length not greater than $2n - 1$ which can separate the two machines. The confirmation experiment takes time $O(pn^3)$ [13].

Theorem 4. *The on-line enumeration algorithm takes $O(Kpn \log n + pn^3)$ steps to identify the derived machine B, where K is the product of all the branching degrees in the specification machine.*

4.1.2 On-line Machine Construction

The second method constructs the machine using *pairwise distinguishing sequences*. The on-line enumeration method generates many candidate machines when the number of nonobservable transitions increases. If pairwise distinguishing sequences [16] exist, the end states of nonobservable transitions can be decided, reducing the number of candidate machines.

On-line Machine Construction

```
queue Q contains machines to explore
begin
1. Insert A into Q;   /*initialization */
2. while Q ≢ φ
```

```
3.      take a machine M from Q;
4.      while M is not decided
5.          explore the visited states in M;
6.          for each state with nonobservable transitions
7.              generate pairwise distinguishing sequences;
8.              if pairwise distinguishing sequence ds exists
9.                  use ds to remove inconsistent transitions;
10.             if only one candidate state s left
11.                 marked s as visited;
12.             else   /*the end states are not distinguishable now*/
13.                 generate multiple copies according to the nonobervable transitions;
14.                 add the copies to Q;
15.     if M is decided
16.         print M;
17. if there are multiple candidate machines
18.     apply cross verification to identify the derived machine;
end
```

This is an *adaptive strategy.* The construction procedure approximates the object derived machine by iterations. When more states are decided, the possibilities of pairwise distinguishing sequences increase and their lengths decrease.

For an NFSM, a distinguishing sequence of two state s_i and s_j may not exist. If it does, in the worst case, its length is up to $(2^n - 2)$[12]. But in practice they do exist and are not long. [5]

4.1.3 Experiments

Table 1 gives the experimental results of our simulations. In a simulation, a NFSM specification is generated according to four parameters: *number of states, input/output alphabet, branching rate*, where *branching rate* indicates the possibility of having multiple transitions for an input. Then on-line enumeration and on-line construction are applied to the NFSM. Each simulation was done 4 times with the same parameters and calculates the average cost[6]. From Table 1 we can tell that in most cases, on-line construction takes less time to identify the object machine than on-line enumeration.

When multiple copies are generated in on-line enumeration, further splitting has to be done for each copy. Also a state distinguishing may have to be repeated on these copies. That is the reason that on-line enumeration takes more time.

For NFSMs with {8 states, 3 inputs, 3 output, branching rate=0.2}, we use on-line construction to identify the derived machines and study the distribution of distinguishing sequences, as shown in Table 2. In most cases, pairwise

[5] Note that even pairwise distinguishing sequence does not exist, the algorithm still works by generating multiple copies and then using cross verification.

[6] The simulations are carried out on a Pentium 1.2GHz PC. Note that the absolute time is not important here. Different simulations should not be compared because the complexity of a NFSM is not merely decided by its parameters.

Table 1. Experiments

# of states	# of inputs	# of outputs	branching rate	average time(msec)	
				on-line enumeration	on-line construction
4	2	2	0.2	115	70
4	2	2	0.4	121	42
4	3	3	0.2	335	27
4	3	3	0.4	50	18
8	2	2	0.2	55	16
8	2	2	0.4	66	14
8	3	3	0.2	38	20
8	3	3	0.4	387	34
10	2	2	0.2	63	25
20	2	2	0.2	104	22

distinguishing sequences exist and their length is less than 3. This explains the advantage of the on-line construction method.

Table 2. Experiment Results on Active Testing

		1	2	3	4	5
# of requests for DS		19296	240	64	16	532
DS not exist		0	0	0	2	200
DS exist	len = 1	17376	162	32	12	266
	len = 2	1920	78	32	2	57
	len = 3	0	0	0	0	9

4.2 Passive Testing Approach

4.2.1 Algorithm

The passive testing procedure for identifying derived machines from NFSM with non-observable transitions is different from ALGORITHM I. The current state set may not converge even after it reaches a singleton state. To identify B, *backtracking* is used to rule out the unselected transitions from A.

ON-LINE CHECKING - ALGORITHM II

input: ONFSM A and observed sequence
output: fault_detected or machine B

begin
1. $B := A$; /* B is derived from A by deleting transitions from the branching states */
2. $C := \{s_0, s_1, ..., s_{n-1}\}$; /* the initial possible state set */
3. **while** ($|C| > 0$ & $next(x/y)$)

```
4.      if ( |C| ≡ 1 )   /* singleton */
5.          remove the transitions {(s, s', x/y')|s ∈ C, y' ≠ y } from B;
6.      C := h_y(C, x);   /* state tracing */
7.      record the transitions from C⁻ to C in tr(C⁻, C);
8.      if (∃s ∈ C⁻, h_y(s, x) ≡ 0)   /* discrepancies from the current state set */
9.          L := C⁻, level_changed := true;   /* the backtracking start level */
10.         while ( level_changed )   /* backtrack until the current level unchanged */
11.             if (∃s ∈ L with no outgoing transitions)
12.                 foreach (s ∈ L with no outgoing transitions)
13.                     delete transitions leading to s in tr(L⁻, L);
14.                     delete s from L;   /* s is not in the current path */
15.                 L := L⁻;   /* set L to a upper level */
16.             else   /* the current level unchanged */
17.                 level_changed := false;   /* backtracking ends */
18.         while ( L < C )   /* go from the unchanged level to the end of the explored levels */
19.             if ( |L| ≡ 1 )   /* the current level is a singleton */
20.                 remove { t|s ∈ L, t = (s, s', x_L/?), t ∉ tr(L, L⁺) } from B;   /* ? means don't care */
21.                 if ( B becomes deterministic )   /* all the branchings are decided */
22.                     return B;
23.             L := L⁺;   /* set L to the next level */
24. if ( |C| ≡ 0 )   /* fault detected */
25.     return fault_detected;
26. if ( next(x/y) ≡ null )   /* passive testing ends */
27.     return B;   /* B is not decided yet */
end
```

Algorithm II is different from Algorithm I. Since the current state set is not monotonously decreased, the *backward checking* and *forward checking* are combined together. Whenever a discrepancy in outputs is observed in the current state set, backward checking is triggered to remove the invalid paths in the past. Then forward checking is used to select transitions from singletons.

We use an example specified in Figure 4 to illustrate the procedure. Assume the current level is $\{s_1\}$ and a sequence a/x b/y c/z is observed. Backward checking starts at the third level when s_5, s_7 cannot fire any transitions with c/z. Backtracking removes s_3, s_5 from the levels. Hence the upper levels become singletons. Then forward checking removes the following three transitions from B: $s_1 \overset{a/x}{\rightarrow} s_3$, $s_2 \overset{b/y}{\rightarrow} s_5$, $s_2 \overset{b/z}{\rightarrow} s_6$.

Backtracking stops when the level L stays unchanged in this tracking. We call a level L visited in backtracking if one or more states are removed in this backtracking. A level is at most visited n times in backtracking during the whole procedure, where n is the number of states in A. The forward checking is similar.

In Algorithm II, the state tracing levels are required to be stored for backtracking. Note that when a level becomes a singleton, backtracking will not overcome it and the level information before it can be discarded.

— : this transition(state) is removed from the passive testing map
× : this transition is removed from the implementation machine

Fig. 4. The passive test map

Theorem 5. *If A is a NFSM and B is a derived machine of A, Algorithm II is correct and takes time $O(n * L)$ and space $O(nl)$ for checking where L the test sequence length, and $l < L$ is the maximum length of a backtracking.*

4.2.2 Experiments

In a passive testing simulation, a NFSM and its derived machines are generated at first; then for each derived machine, multiple observed sequences are generated; after that the passive testing algorithm is applied to check if the derived machine can be identified. In Table 3 , the 4 columns in the left are parameters of the NFSMs; the length of a random generated sequence is $n * p$. A set of observed sequences are generated from the derived machine. If there exists 22 derived machines, 14 are identified by the current set of sequences, it is denoted as 14/22. The other 8 derived machines are not identified by the observed sequences. It is clear that the observed behavior increases the possibility of machine identification.

Table 3. The effect of multiple observed sequences

# of states	# of inputs	# of outputs	branching rate	length of a sequence	# of observed sequences			
					0.5*n	n	4n	8n
4	2	2	0.1	n*p	5/5	5/5	5/5	5/5
4	2	2	0.2	n*p	10/22	11/22	14/22	14/22
8	2	2	0.2	n*p	11/22	17/22	22/22	22/22

Also some observed sequences should be long enough to reach the all the states in the machine. Table 4 shows long observed sequences increase the possibility of machine identification by passive testing.

Table 4. The effect of lengths of observed sequences

# of states	# of inputs	# of outputs	branching rate	# of sequences	length of observed i/o sequence			
					0.5 n*p	n*p	2n*p	10n*p
4	2	2	0.1	n	5/5	5/5	5/5	5/5
4	2	2	0.2	n	4/22	11/22	13/22	14/22
8	2	2	0.2	n	6/22	14/22	22/22	22/22

4.3 TCP Congestion Control

Congestion control is required in TCP implementations. Several algorithms have been proposed and standardized [17] in the network community. In RFC2581 [17], *Slow Start* and *Congestion Avoidance* are mandatory , while *Fast Retransmit* and *Fast Recovery* (FR/FR) are recommended. Figure 5 shows the difference between them. The FR/FR algorithm has one more state, *FR/FR*, than the basic requirement. When duplicate ACKs (*dupAck*) are observed, the FR/FR algorithm counts its number and fires the transition to state *FR/FR* when there are 3 *dupAck*s. The transitions about *dupAck* are non-observable.

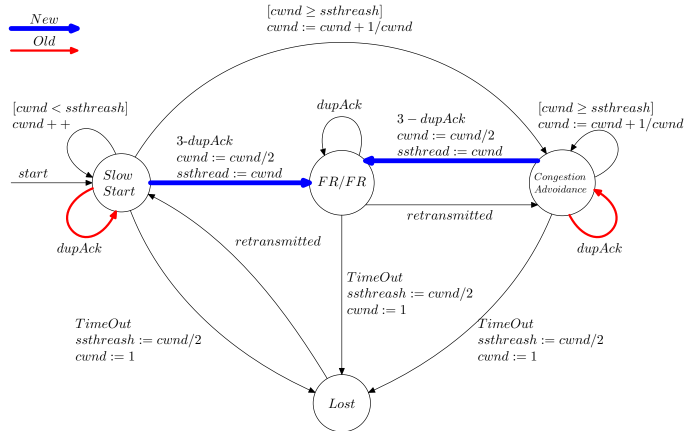
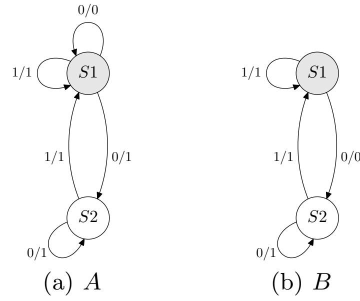


Fig. 5. TCP Congestion Control Algorithms

Different congestion control algorithms are deployed in the Internet. It is a job to study their deployment and their influence on TCP performance. In [7], the authors designed test scenarios to identify what algorithm is used in the remote web server. It is an active testing approach.

5 Case 3: Conformance Relation in Observable NFSM

The specification machine A is observable but the implementation machine B is not restricted to a derived machine of A. Fig.6 gives an example that B conforms to A but B is not equivalent to any derived machine of A.



(a) A (b) B

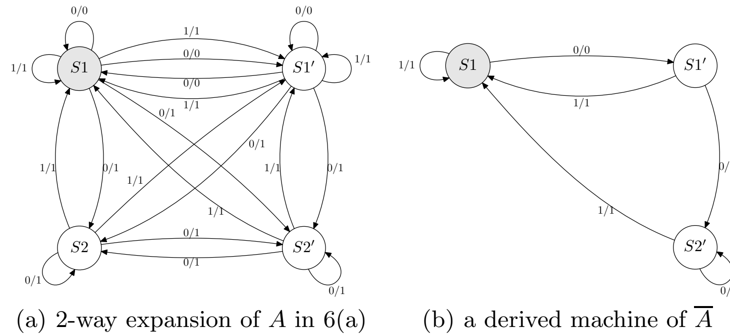
Fig. 6. Machine A, B, $T(B, S1) \subseteq T(A, S1)$

We will prove that B is a derived machine of the *k-way expansion* of A, where k is the upper bound of the state number of B.

Definition 7. *Given a NFSM* $A = (I, O, S, h)$*, a k-way expansion of* A *is a machine* $\overline{A} = (I, O, S', h')$ *that*

$\forall s_i \in S,\ s_i^m \in S', 1 \le m \le k;$

$\forall [s_j, y] \in h(s_i, x),\ [s_j^m, y] \in h'(s_i^l, x), 1 \le m \le k, 1 \le l \le k$ □



(a) 2-way expansion of A in 6(a) (b) a derived machine of $\overline{A}$

Fig. 7. 2-way expansion and its derived machine

We can see that $\overline{A}$ has k times the number of states in A and k^2 transitions for each transition in A. Our idea is that each state in B may be constructed by

a set of states in $\overline{A}$. Intuitively each state in B may be constructed from a copy of A, by selecting the features it needed. If each state in B can be simulated by one copy of A, then B may be constructed in the *k-way expansion* of A.

Theorem 6. *Suppose A is a minimal ONFSM specification and B is a minimal DFSM that conforms to A that has k states. Then B is equivalent to some derived machine of the k-way expansion of A.*

The proof is given in [12]. □

Fig.7(b) is a derived machine of the 2-way expansion of machine A in Fig.6. It is equivalent to machine B in Fig.6.

We show in [12] that B may not be a derived machine of $(k-1)$-way expansion of A. $k - way$ expansion is the upper bound and the bound is tight.

There are totally $k^{knp}/(k!)^n$ derived machines from the k-way expansion of A. See [12] for an explanation of the calculation.

For this case we only consider the active testing approach. Enumerating the derived machines can only be applied when k, n, p are very small numbers. We examine the topological structure of the graph of the machine, take into consideration of the strongly connected components(SCC) and the branching to them, and apply heuristics to reduce the candidate derived machines. We use the following PPP Authentication to illustrate.

The Password Authentication Protocol (PAP) is used as a PPP authentication protocol [18]. The dial-in system sends its PAP authentication information (*username, password*) in an Authenticate-Request to the server. The server sends an Authenticate-Ack to indicate the success of authentication. If authentication fails, the server sends an Authenticate-Nak. It should also attempt to terminate the link to frustrate a would-be system cracker, although a small number of attempts are often permitted. Most dial-up systems permit users to retry several times. We take state *Ack-Rcvd* and its neighbor states from the state machine. {*RAR+, RAR-* } are valid inputs for *Ack-Rcvd*. Another input *RAA* is for the PAP server and is ignored in *Ack-Rcvd*, since *Ack-Rcvd* is a state for the PAP client. The SCC containing *Ack-Rcvd* and its outgoing transitions are shown in Fig 8.

Assume that the retries number is not greater than 3, we can generate the derived machines from *3-way expansion* of state *Ack-Rcvd*. The derived machines can be easily classified into 4 equivalent classes [19]. Since *RAR+/SAA* is a distinguishing sequence for state *Ack-Rcvd* and *Closed*. We can use it to judge which derived machine is the implementation machine.

6 Case 4: General NFSM

The specification machine A is a general NFSM and the implementation machine B is not restricted - may not be a derived machine of A. It can be shown [19] that B can not be constructed from *k-way expansion* of A, no matter how large k is [12]. Obviously, we can disregard the specification A and apply a test to identify B [16] with an exponential cost. Can we take advantage of the information in the

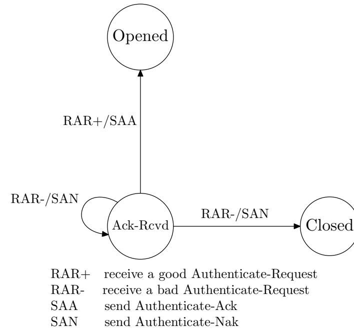


Fig. 8. PAP Authentication with retries

specification machine and identify the implementation machine more efficiently? As a pathological case, the given NDFS A may not contain any information, eg, upon each input/output, there is a transition from each state to all other states. It remains to be investigated that how to explore the structure and available information of A to derive B efficiently or that how to characterize A such that B can be constructed in polynomial time.

7 Conclusion

We have studied the problem of identifying the deterministic implementation from nondeterministic specification using active or passive testing. From experiences in real network protocol system implementations, we introduce the concept of derived machines, developed efficient algorithms for determining the implementation machines, and analyzed the complexity of various cases with different assumptions on the specifications and implementations. The results are summarized in Table.5.

Table 5. Derived Machine and Conformance

Case	complexity	active testing	passive testing
derived machine of ONFSM	$O(pn^2)$	$\surd$	$\surd$
derived machine of NFSM	NP-hard	on-line exploration	back-tracking
conformance of ONFSM	k-way expansion	expand SCC	?
conformance of NFSM	exponential	?	

We only explored limited structures of the nondeterministic specifications, i.e., their observability and k-way expansions. In practice, the nondeterminism is more restricted, as seen from the case studies of PPP, RIP and TCP. It remains to be studied how to further explore and classify the nondeterministic structure so that the implementations can be determined more efficiently.

References

1. Yannakakis, M., Lee, D.: Testing finite state machines: Fault detection. J. Computer Science and Systems **50** (1995) 209–227
2. Luo, G., v. Bochmann, G., Petrenko, A.F.: Test selection based on communicating nondeterministic finite state machines using a generalized wp-method. IEEE Transactions on Software Engineering **20** (1994) 149–162
3. Petrenko, A., Yevtushenko, N., Bochmann, G.: Testing deterministic implementations from nondeterministic fsm specifications. In: Proc. of the 9th Intern Workshop on Protocol Test Systems. (1996)
4. Alur, R., Courcoubetis, C., Yannakakis, M.: Distinguishing tests for nondeterministic and probabilistic machines. In: Symposium on Theory of Computer Science, 1995, ACM. (1995) 363–372
5. Hierons, R.: Generating candidates when testing a deterministic implementation against a non-deterministic finite-state machine. The Computer Journal **46** (2003) 307–318
6. Lee, D., N.Netravali, A., Sabnani, K.K., Sugla, B., John, A.: Passive testing and applications to network management. In: Proceedings of IEEE International Conference on Network Protocols. (1997)
7. Padhye, J., Floyd, S.: On inferring TCP behavior. In: ACM SIGCOMM. (2001) 287–298
8. Starke, P.H.: Abstract Automata. American Elsevier Publishing Company, Inc (1972)
9. Lee, D., Chen, D., Hao, R., Miller, R.E., Wu, J., Yin, X.: A formal approach for passive testing of protocol data portions. In: International Conference on Network Protocols. (2002) 122–131
10. Milner, R.: Communication and Concurrency. Prentice Hall International (1989)
11. Malkin, G.: RIP version 2 - carrying additional information. RFC 1723 (1994)
12. Miller, R.E., Chen, D., Lee, D., Hao, R.: Coping with nondeterminism in network protocol testin, full version. (2005)
13. Lee, D., Sabnani, K.: Reverse-engineering of communication protocols. In: International Conference on Network Protocols. (1993) 208–216
14. Hopcroft, J.: An nlogn algorithm for minimizing states in a finite automaton. The Theory of Machines and Computations (1971) 189–196
15. Kohavi, Z.: Switching and Finite Automata Theory. second edition edn. McGraw-Hill, Inc., New York (1978)
16. Lee, D., Yannakakis, M.: Principles and methods of testing finite state machines - a survey. Proceedings of The IEEE **84** (1996) 1090–1123
17. Allman, M., Paxson, V., Stevens, W.: TCP Congestion Control. RFC 2581 (1999)
18. Carlson, J.D. In: PPP Design, Implementation, and Debugging. 2nd edition edn. Addison Wesley Professional (2000) 99
19. Lee, D., Miller, R.: Passive testing of network protocols specified by nondeterministic finite state machines, draft document. (1999)