

# Network Protocol System Monitoring - a Formal Approach with Passive Testing

David Lee, Dongluo Chen, Ruibing Hao, Raymond E. Miller, Jianping Wu and Xia Yin

**Abstract**— We study network protocol system monitoring for fault detection using a formal technique of passive testing that is a process of detecting system faults by passively observing its input/output behaviors without interrupting its normal operations. After describing a formal model of Event-driven Extended Finite State Machines we present two algorithms for passive testing of protocol system control and data portions. Experimental results on OSPF and TCP are reported.

## 1 Introduction

Protocol system fault detection is often conducted by active testing. A test sequence is designed with a desired fault coverage and then applied to a system implementation under test to reveal faults from its output responses [1]. However, for network management often we cannot interrupt normal system operations arbitrarily by applying test sequences; we can only monitor the system input/output behaviors to infer possible faults. Consequently, passive testing has been proposed for network fault management [2]; it is a process of detecting faults in a network protocol system by passively observing its input/output behaviors without interrupting the normal network operations.

A passive fault detection approach was first proposed in [3] in 1980s, in which passive observers are used for network fault detection. Later on multiple observers have been used to reduce the complexity of each observer [4].

The protocol specification contains a control portion and a data portion. The control portion determines how messages are sent and received and can be considered as a Finite State Machine (FSM), which contains states and transitions. The data portion specifies functions that involve the parameter values associated with the messages. Informally, the data portion is described in words which are then formulated as a set of rules among parameter values. Formally, the data portion is specified by an Extended Finite State Machine (EFSM), which is an extension from the FSM achieved by introducing variables and parameters [1].

A general formal model for passive conformance testing has been first presented in [2] where the finite state machine is used to model the protocol control portion, and fault detection algorithms for both deterministic and non-

deterministic systems are designed, implemented, and applied to detect faults at run-time for the Signal System 7 (SS7) protocol system. Similar work based on signature is proposed in [5]. In [6] and [7] the authors specify network systems as communicating finite state machines (CFSM) and study fault detection and location. Backtracking is an effective technique for fault detection [8]. The formal technique finds more new applications such as [9] that investigates passive monitoring and inferring TCP connection characteristics. The published works are mostly on testing of the control portion rather than data portion. In [10] a simple algorithm of passive testing on EFSM has been developed and applied to the GSM-MAP protocol. The algorithm records the values of variables, and discards them whenever inconsistency occurs. Yet no convincing arguments are given on how the faults can be detected. A more systematic study of passive testing of the data portion has been reported in [11] and this article is an expanded version of that paper.

Variables contain important information of protocol systems; they determine the system states, data portions and external behaviors of protocol systems. On the other hand, it is known to be difficult to test variable values due to the complexity of tracing them [1]. In this paper, we use the range to denote the value of protocol variables, propose an algorithm to trace the variable values as well as the system states, and present two efficient implementations of the algorithm. In the first implementation we *narrow* down the range of each variable as much as possible whenever additional information can be derived from a transition. A set of range operations is introduced and we use examples to illustrate their usage. In the second implementation we record all the constraints derived from a transition path and verify the executability of the path by solving the constraints as a system of linear equations/inequalities. Our algorithms can deal with commonly encountered operations on variable values associated with state transitions and also provide efficient variable value determination for the protocol data portion fault detection.

Network monitoring has been an active networking research topic. Many of the existing research works in this aspect monitor the network situation through network management systems [12] [13]; some of them use diagnostic messages [14]. There have been many monitoring tools: *TCPdump*, *Sniffer* contain protocol decoders for general purpose; the AT&T PacketScope [15] is designed for backbone IP tuning; the Cambridge Nprobe [16] integrates multi-

D. Lee is with the Dept. of Computer Science and Engineering, The Ohio State University. D. Chen, J. Wu and X. Yin are with the Dept. of Computer Science, Tsinghua University, China. R. Hao is with Bell Labs Research China. R. Miller is with the Dept. of Computer Science, University of Maryland.

layer information for WWW and other applications' behavior analysis. Sometimes network devices provide *port mirroring* to duplicate network traffic for analysis purposes.

Our approach is different from those network monitoring works. Firstly, most network monitoring works only consider typical network faults such as up/down of a link/router and the performance of the network, while our work emphasizes protocol implementation faults. Flaws in network devices may cause poor performance or even network crashes [17], e.g., the MCI WorldCom outage in 1999 [18]. Thus our approach is more appropriate to verify new protocol and feature deployments in the Internet[19]. Secondly, our approach is based on an event driven model, it can carry out fault detection at a more abstract event level to identify protocol implementation faults, compared to the packet level analysis in most existing passive monitoring tools.

The paper is organized as follows. In Section 2, an Event-driven EFSM model is described. In Section 3, a simple algorithm is first introduced and discussed, and then a more powerful algorithm is proposed with an analysis. The efficient implementations of the algorithm are also studied. In Section 4, experimental results on the OSPF neighbor state machine and the TCP state machine are reported.

## 2 A Model: Event-driven Extended Finite State Machine

Finite State Machine (FSM) has been widely used to model the network protocol control portion; it contains a finite number of states and produces an output on state transition after receiving an input. However, protocol data portions often contain variables, parameters and operations based on their values, and to model them in a succinct way, the Extended Finite State Machine (EFSM) was introduced [1].

Passive testing is based on the observable behaviors of network system devices, i.e., packets/messages exchanged among peers, which are called *events*. Specifically, an event is either an input message to the device under test or an output message from it. The fields in a message are called parameters of the event. As in an EFSM, a predicate is a Boolean function on the current variable and event parameter values, and an action is an assignment of the variable values. We are ready to define Event-driven Extended Finite State Machine (EEFSM) to model protocol systems for passive testing:

**Definition 1** *An Event-driven Extended Finite State Machine (EEFSM) is a 6-tuple  $M = \langle S, s_0, \Sigma, \vec{x}, \vec{y}, T \rangle$  where  $S = \{s_0, s_1, \dots, s_{n-1}\}$  is a finite set of states with  $s_0$  in  $S$  as the initial state,  $\Sigma = \{e_1(\vec{y}), \dots, e_k(\vec{y})\}$  is a finite set of events,  $\vec{x} = (x_1, \dots, x_p)$  is a finite set of variables,  $\vec{y} = (y_1, \dots, y_q)$  is a finite set of parameters, and  $T$  is a finite set of transitions. For a transition  $t = \langle s, s', e(\vec{y}), P(\vec{x}, \vec{y}), A(\vec{x}, \vec{y}) \rangle$ ,  $s$  and  $s'$  are the start and end state of the transition,  $e(\vec{y})$  is the triggering event,  $P(\vec{x}, \vec{y})$  is a predicate and  $A(\vec{x}, \vec{y})$  is an action, which is a sequence of assignments*

$\vec{x} := A(\vec{x}, \vec{y})$ , where  $\vec{x}$  and  $\vec{y}$  in the arguments are the current variable and event parameter values, respectively.

The combination of system state and current variable values  $\langle s, \vec{x} \rangle$  is called the configuration of the system. At start state  $s$  and upon input or output event  $e(\vec{y})$ , for a transition  $t = \langle s, s', e(\vec{y}), P(\vec{x}, \vec{y}), A(\vec{x}, \vec{y}) \rangle$ , if for the current configuration and the event parameter values  $\vec{y}$ , the predicate  $P(\vec{x}, \vec{y})$  returns TRUE, then the machine moves to the end state  $s'$  and updates its variable values by the assignment  $\vec{x} := A(\vec{x}, \vec{y})$  where  $\vec{x}$  on the left side contains the updated variable values.

Note that machine variables  $\vec{x}$  and event parameters  $\vec{y}$  are different. Machine variables are local; they are associated with an EFSM protocol entity. Event parameters are contained in an input/output event to be passed among different EFSM protocol entities. Machine variables can be used in a predicate or assignment, and can be updated. However, parameters can only be used in a predicate or assignment, but cannot be updated.

Some protocol specifications provide transition tables and each transition is an input/output pair. We need to convert the transition table into the EEFSM model. If a protocol specification has a transition in the form of  $s_1 \xrightarrow{i/o} s_2$ , we rewrite it as  $s_1 \xrightarrow{i} s'_1 \xrightarrow{o} s_2$ . For example, a transition  $t : s_1 \xrightarrow{?a(y_1)/!b(x_2), [x_1 > 3], x_2 := y_1 + 8} s_2$  (which means when the machine is in state  $s_1$ , upon receiving of input  $a(y_1)$ , if  $[x_1 > 3]$ , it will assign variable  $x_2$  with a new value  $y_1 + 8$  and produce an output  $b(x_2)$ ) is mapped to two EEFSM transitions and an additional state:  $t' : s_1 \xrightarrow{?a(y_1), [x_1 > 3], x_2 := y_1 + 8} s'_1$  and  $t'' : s'_1 \xrightarrow{!b(y_2), [y_2 \equiv x_2], null} s_2$ . Note that EEFSM is a verification model. Only event parameters are listed inside an event, thus we use  $!b(y_2), [y_2 \equiv x_2]$  instead of  $!b(x_2)$ . It means, if the machine outputs an event  $b$ , the value of its parameter  $y_2$  should equal to the value of machine variable  $x_2$ .

This model is rather flexible and yet powerful enough for a study of passive testing of network protocols, such as OSPF and TCP.

## 3 Passive Testing Algorithms

Passive testing of a protocol control portion usually contains two phases. It first identifies the current state, which is called *passive homing*. Then it attempts to detect faults, which are revealed by the discrepancies between the implementation and the specification from passively tracing the observed events. However, when the protocol data portions are involved in an EEFSM model the homing phase has to identify the current state and the variable values. Since the current state and variable values depend on each other, it is hard to identify them separately. Taking advantage of their mutual dependency, we determine them simultaneously. We study two passive testing algorithms with the EEFSM model. Algorithm 1 is rather straightforward, however, it shows the basic feature of passive testing. After discussing its deficiencies, we present Algorithm

2, which is much more powerful for detecting faults.

### 3.1 Algorithm 1

We want to identify the current state and each variable value in  $\vec{x} = (x_1, \dots, x_p)$ . If a variable  $x_i$  has a known value, we denote it by  $x_i^*$ , otherwise, it is unknown without a superscript “\*”. The current status of a machine can be characterized by: the current possible state set  $S_c$  and the current variable value vector  $\vec{x}$  with the known variables marked by \*. Initially a machine can be in any one of the states with  $S_c = \{s_0, s_1, \dots, s_{n-1}\}$ , and all the variable values are unknown.

To check whether a transition is executable and can be traced along in a passive testing process, its predicate has to be evaluated. However, it is possible that some variables are unknown in the predicate and, consequently, an evaluation may yield one of the following results: (1) TRUE: The Boolean expression is evaluated to be TRUE; (2) FALSE: The Boolean expression is evaluated to be FALSE; or (3) POSSIBLE: There is no definite answer because some of the variable values are unknown. It is a tri-valued logic expression whose truth table is defined in Appendix B. If a predicate is evaluated to be FALSE, definitely its corresponding transition cannot be executed. Otherwise, the predicate is TRUE or POSSIBLE and the transition is or might be executable. For the correctness of the testing, i.e., we do not want to rule out any executable transitions, we take into consideration all the possibly executable transitions. Note that when an event occurs at a start state  $s$ , there can be one or more possibly executable transitions, so long as its triggering event is the same as the one observed, the state  $s$  is in the current possible state set  $S_c$ , and its predicate is evaluated to be TRUE or POSSIBLE. Each such transition creates a next state  $s'$  and updated variable values  $\vec{x}'$  from the corresponding action by:

**Variable Assignment Rule 1** For a variable  $x_i$ , if all the possibly executable transitions either assign it a same value  $x_i^*$  or do not assign any value to it but it has a current value  $x_i^*$ , then its new value is assigned as  $x_i^*$ . Otherwise, it is assigned as *unknown*.

On the other hand, if a transition is executable or possibly executable, the end state of this transition is added to the next possible state set  $S_c$ . Therefore, we obtain the next possible system status after processing an event. This passive testing process is continued upon each observed event until the next possible state set  $S_c$  becomes empty, and, in this case a fault must have occurred.

#### Algorithm 1

Input : IUT and the observed events /\* implementation under test: an EEFM \*/  
Output : FAULT detected

**begin**

```

1.  $S_c := \{s_0, s_1, \dots, s_{n-1}\};$ 
   /* initialize possible current state set */
2.  $\vec{x} = \{x_1, \dots, x_p\}, x_i := unknown, i = 1, \dots, p;$ 
   /* variable values are unknown initially */
3. while (Event( $e(\vec{y})$ ))
4.    $S'_c := \phi;$  /* possible next state set */
5.   for each transition  $t$  such that  $t.start\_state \in S_c$ 
6.     if ( $(t.event \equiv e) \wedge (t.predicate(\vec{x}, \vec{y}) \neq FALSE)$ )
       /* check current event and predicate */
7.        $S'_c := S'_c \cup \{t.end\_state\};$  /* update state */
8.       determine updated  $\vec{x}$  by Variable Assignment
           Rule 1; /* update variable values */
9.     if  $S'_c \equiv \phi$  /* a contradiction; faults detected */
10.      return FAULT;
11.    $S_c := S'_c$  /* done with current event */
end

```

We now discuss the correctness of Algorithm 1. The rationale behind passive testing is similar to “proof by contradiction”. We assume that the IUT is correct and then trace it upon input/output events. When Algorithm 1 terminates with FAULT, i.e., the possible current state set  $S_c$  is an empty set, it is obvious that it is impossible for the IUT to produce the last event if it is correct; otherwise, there must be a non-empty  $S_c$ , since during the tracing we have only ruled out the next states, which are definitely impossible. Therefore, if Algorithm 1 returns FAULT, the IUT must be faulty. On the other hand, during the tracing, we are “lenient” in a sense that we only rule out the impossible next states and keep all the possible next states, which are either executable or possibly executable. Therefore, if the IUT conforms to the specification, the current state set  $S_c$  is never empty, Algorithm 1 keeps on tracing, and FAULT will never be returned. Hence it is correct; it will never rule out any correct implementations; when it returns FAULT the IUT is definitely faulty.

**Proposition 1** *Algorithm 1 is correct. If an IUT conforms to the specification, Algorithm 1 never returns FAULT.*

Algorithm 1 attempts to keep track of the variable values while maintaining a record of all the possible current states. The essence of passive testing is “narrowing” down the current possible states and variable values until they become an empty set and that reveals faults. Algorithm 1 is simple yet not efficient in detecting faults. Specifically, it does not take advantage of information of the predicates nor fully utilize the information in the assignments of the executed transitions. Furthermore, Algorithm 1 keeps only one possible variable value set for all possible current states. If we keep track of the variable values for each possible current state, we can more effectively rule out impossible states and variable values. We use the following examples to explain.

1. Since predicates contain constraints on the variable values, they are helpful in identifying the variable values. But in Algorithm 1 predicates are only used to guard the execution of the transitions. Therefore, it may fail to confine the ranges of the variables and, as a result, follow transitions, which are not executable.

See the example in Fig.1. With the initial state S1 and variable  $u$  unknown, after input event  $?a(3)$  occurs, Algorithm 1 takes that the transition from S2 to S3 is also possible. Actually after  $?a(3)$  occurs,  $u > 3$  must be satisfied. If we keep this information, then the transition with event  $!c$  from S2 to S3 cannot be executed because the associated predicate evaluates to be FALSE. So there should not be an input event  $?a(3)$  followed immediately by an output event  $!c$ . If an implementation contains such a fault, it will go undetected by Algorithm 1.

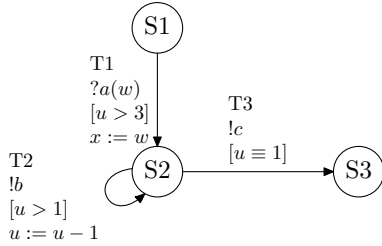


Figure 1: Predicates in Protocol Specification

- Algorithm 1 determines variable values in a simple way without taking advantage of the relations among the variables. See the example in Fig.2. When the transition from S1 to S2 fires,  $x_2$  gets its value from  $x_1 + 1$ . Then  $x_2$  is output in the transition  $!c(y = 5)$  through  $y$ . Even if we have used the implicit information, the variable values we obtain after these transitions are  $\langle x_1 = ?, x_2 = 5 \rangle$ . But if we know the relation between  $x_1$  and  $x_2$ , we can tell that  $x_1 = 4$ . Note that there are many assignments in protocol specifications between variables. Discarding the information of the relations between variables reduces the capability to discover variable values and hence to detect faults.

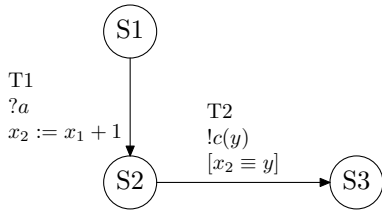


Figure 2: Relations among Variables

- Algorithm 1 only keeps one possible variable value set; a variable is *unknown* unless it has a definite and consistent assignment. This approach is not effective in “narrowing” down the possible variable values. If we associate a variable value set with each possible state, more information can be obtained. See the example in Fig.3. With Algorithm 1, no variable value can be determined. But if we associate a variable value set with each possible state, more information can be extracted. In this example, when even  $?a(4, 7)$  occurs, we trace the two transitions. The current possible configurations are  $\langle S2, x_1 = 0, x_2 = 8 \rangle$  and

$\langle S3, x_1 = 4, x_2 = 10 \rangle$ . Afterwards if event  $!d$  happens, we can delete the first one and know the current configuration as  $\langle S3, x_1 = 4, x_2 = 10 \rangle$ . Here the event  $!c$  and  $!d$  are characterizing sequences that can distinguish S2 from S3.

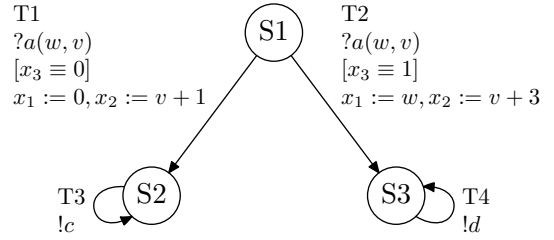


Figure 3: Characterizing Sequences

## 3.2 Algorithm 2

The previous observations lead to Algorithm 2 that takes advantage of information in the predicates and the relations among the variables, and processes each possible current state separately along with its possible variable values. We use assertion to formalize the constraints from the predicates and assignments:

**Definition 2 (Assertion)** *An assertion,  $assert(\vec{z})$ , is a function that returns a Boolean value TRUE with the variable values  $\vec{z}$ .*

Obviously, an assertion is a constraint on the variable values  $\vec{z}$ , which consist of both the variable values  $\vec{x}$  and parameter values  $\vec{y}$  in the events. When a transition is executed its predicate is assumed to be TRUE and it is taken as an assertion:  $P(\vec{x}, \vec{y}) \equiv TRUE$ . An assignment  $\vec{x}' := A(\vec{x}, \vec{y})$  also provides an assertion:  $\vec{x}' \equiv A(\vec{x}, \vec{y})$  where  $\vec{x}'$ ,  $\vec{x}$  and  $\vec{y}$  are the updated variable values after the transition, current variable values, and the parameter values associated with the event  $e(\vec{y})$ .

For example, in Fig.2 upon event  $?a$  the assignment ( $x_2 := x_1 + 1$ ) gives assertion  $x'_2 \equiv x_1 + 1$ . When the output event  $!c(5)$  occurs, we have assertion  $x'_2 \equiv 5$ . From the two assertions, we obtain  $x_1 = 4$ . Naturally, the current protocol system status is represented by its current state and the associated assertions:

**Definition 3 (Candidate Configuration)** *A Candidate Configuration (CC) is a pair  $c := \langle s, Assert_s(\vec{x}, \vec{y}) \rangle$  where  $s$  is a state and  $Assert_s(\vec{x}, \vec{y})$  is a set of assertions on the variable and parameter values.*

Initially, there are  $n$  CCs:  $\langle s_i, \phi \rangle, i = 0, 1, \dots, n - 1$ . Upon input/output events, the set of CCs is updated. Each transition from every CC is checked against the current event to find if it can lead to a new CC. When the set of CCs becomes empty faults are detected. This procedure is summarized in:

### Algorithm 2

Input: IUT and the observed events /\* implementation

under test: an EEFSM \*/  
Output: FAULT detected  
 $C$ : current CC set  
 $C'$ : next CC set

**begin**

```

1.  $C := \{ \langle s_i, \phi \rangle : i = 0, 1, \dots, n-1 \};$ 
   /* initialize possible current CC set */
2. while Event( $e(\vec{y})$ )
3.    $C' := \phi$ ; /* possible next CC set */
4.   for each  $\langle s, Assert_s \rangle$  in  $C$ 
   /*  $Assert_s$  is assertion set with state  $s$  */
5.     for each transition  $t$  in  $T$ 
6.       if ( $t.event \equiv e$ )  $\wedge$  ( $t.start\_state \equiv s$ )
   /* check event and start state */
7.         if  $Eval(t.predicate, Assert_s) \neq \text{FALSE}$ 
   /* TRUE or POSSIBLE */
8.            $Assert_{t.end\_state} := Merge(Assert_s,$ 
    $t.predicate, t.action)$ ;
9.            $C' := C' \cup \{ \langle t.end\_state,$ 
9.              $Assert_{t.end\_state} \rangle \}$ ; /*add a next CC*/
10.        if  $C' \equiv \phi$  /* a contradiction; faults detected */
11.          return FAULT;
12.    $C := C'$  /* done with the current event */
end

```

Initially, all states are possible yet without any assertions, as initialized in Line 1. Upon an event  $e(\vec{y})$  in Line 2, it updates the CC set in Line 3-12. For each CC,  $\langle s, Assert_s \rangle$ , in  $C$ , we check it against every transition  $t$ ; if it coincides with the current event as checked in Line 6, we process it as follows. Subroutine  $Eval(t.predicate, Assert_s)$  checks the executability of the transition. If it returns FALSE, then it is definitely not executable and we discard  $t$ . Otherwise, it is possibly executable and we proceed with it. Subroutine  $Merge(Assert_s, t.predicate, t.action)$  in Line 8 takes all the assertions for state  $s$  and also the assertions from  $t.predicate$  and  $t.action$ , and constructs an updated CC for the next state:  $\langle t.end\_state, Assert_{t.end\_state} \rangle$ , which is then added into the updated CC set  $C'$  in Line 9. Note that after processing all the transitions in Line 9 if a CC contains no valid assertions, we discard it, since we would have followed transitions, which are not executable. After processing an event  $e$ , if the current CC set  $C'$  is empty at Line 10, we have arrived at a contradiction: we have followed transitions which are not executable, and, consequently, there are FAULTS in the protocol system implementation; we terminate the procedure in Line 11. Otherwise, we continue the passive monitoring upon new events from Line 2.

The subroutine  $Eval(t.predicate, Assert_s)$  evaluates the predicate  $t.predicate$  along with the assertions of state  $s$ . From all the constraints on the variables and the parameters  $\vec{y}$  in the event  $e(\vec{y})$ , if  $t.predicate(\vec{x}, \vec{y})$  is TRUE or FALSE, it returns the respective value. Otherwise, it returns POSSIBLE. Basically, it is to check the satisfiability of a system of equations and inequalities. The subrou-

tine  $Merge(Assert_s, t.predicate, t.action)$  is to update the value of  $\vec{x}$  and renew the assertion. It constructs a new assertion set for the next state  $t.end\_state$  from the current state assertions,  $Assert_s$ , and the additional assertions from  $t.predicate$  and  $t.action$ . We defer the implementation details in Subsection 3.3.

By a same argument as for Algorithm 1 if Algorithm 2 returns FAULT then the IUT is definitely faulty, and it never returns FAULT if the IUT conforms to the specification, since it only rules out the definitely impossible CCs and keeps all the possible CCs.

**Proposition 2** *Algorithm 2 is correct. If an IUT conforms to the specification, Algorithm 2 never returns FAULT.*

### 3.3 Efficient Implementations of Algorithm 2

In communication protocols, variables can be of Boolean, integer and other types, and most variables are Boolean or integer-valued, which are easier to trace in passive testing, compared with other types. In this section, we discuss the implementation of subroutines  $Eval$  and  $Merge$  in Algorithm 2 for integer-valued variables only and that satisfies the needs for our study of the practical protocol system. Variables of other types in general are more difficult to handle. In practice, we usually explore the practical constraints for an efficient treatment of these variables, and the approach used in this work for handling the integer-valued variables may also be useful. We shall not further digress here.

#### 3.3.1 Evaluating Predicates by Integer Linear Programming

Subroutine  $Eval(t.predicate, Assert_s)$  checks if the predicate of transition  $t$  is executable from state  $s$  under the constraints of  $Assert_s$ . Obviously, it is to evaluate  $t.predicate \wedge Assert_s$ , which is a tri-valued logic expression, since some variables are unknown. We first transform it into *Disjunctive Normal Form*(DNF). Generally transforming a boolean expression into DNF will blowup exponentially. But in our experiments, this transformation dose not cause a computability problem for two reasons. First in network protocols, the expressions are not complicated. The number of simple predicates in a predicate is not large and there are much more “AND” than “OR” operators. Second, to reduce the complexity in the implementation, we replace all the known variables and parameters with their values before the transformation. Then evaluate each simple predicate before blowup. For example, a predicate ( $FLAG \equiv 1 \wedge x < 7$ )  $\vee$  ( $FLAG \equiv 0 \wedge (x > 3 \vee y < 5)$ ), where  $FLAG$  is a boolean parameter of the current event, can be decided easily when  $FLAG$  is observed from the event. If  $FLAG$  equals 1, the predict turns out to be  $x < 7$ ; if  $FLAG$  equals 0, the predicate turns out to be  $(x > 3 \vee y < 5)$ , which is quite simple compared with the original predicate. In our experiments the transformation is not time consuming.

After the transformation, we then check each conjunctive term that consists of one or more simple predicates, which can be normalized as follows.

**Definition 4 (Normalized Simple Predicate)** *A normalized simple predicate is in the form of  $a_1x_1 + a_2x_2 + \dots + a_kx_k \sim Z$ , where*

- $\sim$  is one of the symbols in  $\{<, >, \leq, \geq, \equiv, \neq\}$
- $x_1, x_2, \dots, x_k$  are integer-valued variables
- $a_1, a_2, \dots, a_k, Z$  are integer constants

The normalized simple predicates of a conjunctive term consist of a system of linear equations/inequalities. The problem of checking the feasibility of a system of linear equations/inequalities is a *Linear Programming*(LP) problem, which has polynomial time algorithms [20]. If some of the variables are integer-valued, it is *Mixed Integer Linear Programming*(MILP) and is NP-hard [21]. Integer-valued variables are pretty common in network protocols, and checking the feasibility of a conjunctive term can be reduced to an MILP problem. Heuristic methods, such as Branch-and-Bound and Cutting-plane algorithms, are applicable for solving the problem. In our implementation of the *Eval* subroutine, we use an MILP software package *lp\_solve*(which uses the Branch-and-Bound algorithm) to solve the MILP problem.

**Subroutine *Eval\_ILP***(*t.predicate*, *Assert<sub>s</sub>*)

Input : *t.predicate* /\* current predicate \*/  
*Assert<sub>s</sub>* /\* assertion set of current state \*/  
Output : FALSE or POSSIBLE

**begin**

1. **if** *t.predicate* evaluates FALSE
2.     return FALSE; /\* transition not executable \*/
3. let *Cons* be the DNF representation of the conjunction of *t.predicate* and all assertions in *Assert<sub>s</sub>*;
4. *flag* := FALSE; /\* value to be returned \*/
5. **for** each conjunctive term *D* in *Cons*
6.     normalize *D* to a set of linear constraints;
7.     call *lp\_solve* with the set of linear constraints;
8.     **if** *lp\_solve* gives a feasible solution
9.         *flag* := POSSIBLE;
10.    **else** delete *D* from *Cons*; /\* *D* not feasible \*/
11. **if** *flag*  $\neq$  FALSE
12.     *Assert<sub>s</sub>* := *Cons*; /\* the updated assertion set \*/
13. **return** *flag*

**end**

In the *Eval\_ILP* subroutine, *t.predicate* is first evaluated in Line 1. If it returns FALSE then the subroutine terminates with a value FALSE. Otherwise, it depends on the feasibility checking of *Assert<sub>s</sub>* along with *t.predicate*. From Line 3 to Line 10 the algorithm checks the feasibility of *t.predicate*  $\wedge$  *Assert<sub>s</sub>* by first transforming it into DNF and then checking the feasibility of each conjunctive term. For each conjunctive term *D*, the algorithm formulates an MILP problem with the set of linear constraints converted from all the simple predicates in the conjunctive term (Line

6). Each conjunctive term is POSSIBLE if and only if the tool *lp\_solve* gives a feasible solution to the MILP problem in Line 7. If none of the conjunctive terms is POSSIBLE, this transition is not executable. Otherwise, this transition is possibly executable and a new configuration is generated with the updated *Cons* as its assertion set.

The software tool *lp\_solve* uses Branch-and-Bound. One of its shortcomings is that it may search the whole feasible solution space in the worst case. The complexity of (Integer) Linear Programming is to cope with the correlative dependencies among variables. However, in communication protocols, the correlative dependencies among system variables are quite simple, and we can take advantage of this observation and design efficient heuristic algorithms for the feasibility checking.

### 3.3.2 Evaluating Predicates by Interval Refinement

In network protocols, the correlative dependencies among the variables are usually rather simple. That is, in a simple predicate  $a_1x_1 + a_2x_2 + \dots + a_kx_k \sim Z$  most coefficients equal 0, and hence the variables are relatively independent. Instead of using the Branch-and-Bound approach, we propose another heuristic method: *Interval Refinement*(IR).

We use *Interval* [22] to represent the current range of a variable. The range of *v* is denoted as  $R(v) = [\underline{v}, \bar{v}]$  where  $\underline{v}, \bar{v}$  are the lower and upper bound of *v*, respectively. As a special case, a known value is represented as  $R(v) = [a, a]$  with  $\underline{v} = \bar{v} = a$ .  $R(\sim Z)$  represents the interval of  $\sim Z$ , e.g.,  $R(> 3) = [4, +\infty)$ . Each integer-valued variable has its pre-defined lower and upper bounds.

Note that a range of a variable is a special case of an assertion, which takes a form  $\underline{x}_i \leq x_i \leq \bar{x}_i$ . Thus we can divide the assertion set into two types: the first type specifies the range of each variable in an interval, denoted as  $R(\vec{x})$ , and the second type specifies the relations among variables, which are boolean functions with value TRUE on  $\vec{x}$ , denoted by  $Dep(\vec{x})$ .

Appendix B gives definitions of Interval arithmetic and evaluation of integer and logical expression with variables whose values are intervals. The ranges of all the integer-valued variables of a protocol system are denoted as  $R(\vec{x})$ . The evaluation of an expression  $f(\vec{x})$  using intervals is denoted as  $f(\vec{x})|_{R(\vec{x})}$ . If  $f(\vec{x})$  is an integer expression as defined in Appendix A, the result is also an interval; if  $f(\vec{x})$  is a logical expression, then the result is one of the three: TRUE, FALSE and POSSIBLE.

Linear Programming deals with a system of linear equations/inequalities, while Interval Refinement is conducted for a simple predicate.

**Definition 5 (Interval Refinement)** *Given a normalized simple predicate  $\sum_i a_i x_i \sim Z$  and  $R(\vec{x})$ , define the refinement of each variable  $R(x_j)$ ,  $j=1, \dots, k$ , as:*

$$R(x_j)' := \begin{cases} R(x_j) & : \text{if } \sum_i a_i x_i |_{R(\vec{x})} \subseteq R(\sim Z) \quad (1) \\ \phi & : \text{if } \sum_i a_i x_i |_{R(\vec{x})} \cap R(\sim Z) \equiv \phi \quad (2) \\ ((R(\sim Z) - \sum_{i \neq j} a_i x_i |_{R(\vec{x})}) / a_j) \cap R(x_j) & : \text{Otherwise} \quad (3) \end{cases}$$

Case (1): All the integer values in  $R(x_j)$  along with other variable values in their respective ranges satisfy the simple

predicate, and  $R(x_j)$  remains the same. Case (2): The integer values in  $R(x_j)$  are disjoint from that specified by the simple predicate, and the new range  $R(x_j)' = \phi$ . Case (3): Neither of the above two cases; there is a non-empty intersection, and we take their intersection as the new range  $R(x_j)'$ .

With interval refinement subroutine  $Eval(t.predicate, Assert_s)$  is to evaluate  $(t.predicate \wedge Dep(\vec{x}))|_{R(\vec{x})}$ . Let  $Dep'(\vec{x}) := t.predicate \wedge Dep(\vec{x})$ . If  $Dep'(\vec{x})|_{R(\vec{x})}$  returns FALSE, transition  $t$  is not executable. Otherwise it is possibly executable and a new candidate configuration is generated from  $t.end\_state, Dep'(\vec{x})$ , and  $t.action$ .

We are to discuss refinement of current variable ranges  $R(\vec{x})$  with regard to a general function  $Dep'(\vec{x})$ . We first convert  $Dep'(\vec{x})$  into DNF, then use each conjunctive term to “narrow” down  $R(\vec{x})$  and obtain a refined  $R(\vec{x})'$ . We then combine the ranges from each conjunctive term by:

**Variable Assignment Rule 2** For each variable range if we have a set of non-empty intervals from the processing of the conjunctive terms then the new variable range consists of an interval whose lower (upper) bound is the minimum (maximum) of all the interval lower (upper) bounds.

**Subroutine**  $Eval\_IR(t.predicate, Assert_s)$

Input :  $t.predicate$  /\* predicate of transition  $t$  \*/  
 $Assert_s$  /\* assertion set of current state  $s$  \*/  
Output : FALSE or POSSIBLE

**begin**

1. **if**  $(t.predicate)|_{R(\vec{x})} \equiv \text{FALSE}$
2.     **return** FALSE;
3. obtain  $R(\vec{x})$  and  $Dep(\vec{x})$  from  $Assert_s$ ;
4.  $Dep'(\vec{x}) := t.predicate \wedge Dep(\vec{x})$  ;
5. transform  $Dep'(\vec{x})$  to DNF;
6.  $S(\vec{x}) := \phi$ ; /\*collect variable ranges during iterations\*/
7. **for** each conjunctive term  $D$  in  $Dep'(\vec{x})$
8.      $R_l(\vec{x}) := R(\vec{x}); R_l(\vec{x})' := \phi$ ;
9.     **for** each simple predicate  $p$  in  $D$
10.         normalize  $p$  into form  $\sum_i a_i x_i \sim Z$  ;
11.         **if**  $\sum_i a_i x_i|_{R_l(\vec{x})} \subseteq R(\sim Z)$  /\* $p$  is TRUE\*/
12.             delete  $p$  from  $D$ ,
13.             **goto** 9 for the next simple predicate;
14.             /\* $p$  does not have new constraints; ignore it\*/
15.         **if**  $\sum_i a_i x_i|_{R_l(\vec{x})} \cap R(\sim Z) \equiv \phi$  /\* $p$  is FALSE\*/
16.             **goto** 7 for the next conjunctive term;
17.             /\*  $p$  is FALSE and hence  $D$  is FALSE \*/
18.         **for** each  $x_j, j = 1, \dots, k$  /\* range refinement \*/
19.              $R_l(x_j)' :=$   
                   $((R(\sim Z) - \sum_{i \neq j} a_i x_i|_{R_l(\vec{x})})/a_j) \cap R_l(x_j)$ ;
20.         **if**  $\exists x_i$  with  $R_l(x_i)' \subset R_l(x_i)$   
                  /\*at least one variable interval is strictly reduced\*/
21.              $R_l(x) := R_l(x)'$ , **goto** 9 for another iteration;
22.          $S(\vec{x}) := S(\vec{x}) \cup R_l(\vec{x})$ ; /\* exit loop for  $D$ \*/
23.         /\* collect refined variable ranges \*/
24. **if**  $S(\vec{x}) \neq \phi$  /\* possibly executable \*/

21.     obtain variable ranges  $R(\vec{x})$  from  $S(\vec{x})$  according  
                  to Variable Assignment Rule 2;
22.     **return** POSSIBLE;
23. **else return** FALSE
- end**

For each conjunctive term in  $Dep'(\vec{x})$ , it either generates a new variable ranges  $R_l(\vec{x})$  or is deleted. The new  $R_l(\vec{x})$  is derived by refining the interval of every variable iteratively. Whenever an interval is strictly reduced in Line 17, it goes back to line 9 for another iteration. The stopping criterion is: no variable has a reduced range during an iteration. An example of Interval Refinement can be found in Section 3.4.

When it returns POSSIBLE at Line 22, we can only claim that the value of current conjunctive term is POSSIBLE instead of claiming it is TRUE. The reason is that we have not solved the feasibility problem, but only applied a heuristic approach to check the necessary conditions for detecting faulty conjunctive terms by iteratively narrowing down the variable intervals. If transition  $t$  is possibly executable, its ending configuration is generated with the new  $R(\vec{x})$  and the updated  $Dep'(\vec{x})$  as its  $Assert$ .

The computation time of this subroutine depends on the number of the conjunctive terms and the intervals of the variables. Since each iteration reduces the interval strictly monotonically and the interval boundaries are finite integers, the iteration stops in finitely many steps. In the worst case, the number of iterations is proportional to the total length of the intervals. However, often in practical protocol systems the predicates are rather simple and the interval ranges are small. We have analyzed some protocol specifications, such as OSPF and TCP, and the number of iterations for a conjunctive term is no more than 3.

We have implemented subroutine  $Eval$  using both MILP and Interval Refinement, respectively, and applied them to OSPF and TCP. Both experiments give the same results. The MILP approach takes more time to converge, yet it has more powerful fault detection capability than the Interval Refinement approach that indeed converges quickly since it only checks necessary conditions for reducing the variable ranges yet has less capability of fault detection. Therefore, we can use Interval Refinement for on-line fault detection and use MILP to examine the data more thoroughly offline.

### 3.3.3 Executing Actions

If subroutine  $Eval$  returns POSSIBLE, transition  $t$  is considered possibly executable, and thus generates a new CC, in which the variables may be further updated by the assignments in the action  $t.action$  in the subroutine  $Merge$  ( $Assert_s, t.predicate, t.action$ ) in Algorithm 2 as follows. First it merges  $t.predicate$  with  $Assert_s$ , which has been illustrated in subroutine  $Eval$ . Then it combines the assertions of  $Assert_s$  with that from the assignments in  $t.action$  as in the following subroutine:

**Subroutine**  $Merge\_action(Assert_s, t.action)$

Input :  $Assert_s$  /\* current assertion set \*/  
 $t.action$  /\* action of current transition \*/  
Output : updated  $Assert$

**begin**

1. obtain  $R(\vec{x})$  and  $Dep(\vec{x})$  from  $Assert_s$ ;
  2. **for** each assignment  $w := f(\vec{x})$  in  $t.action$
  3.  $R(w) := f(\vec{x})|_{R(\vec{x})}$ ; /\* update range \*/
  4. **if**  $w$  is not in the arguments of  $f$   
/\*  $f$  is not a function of  $w$  \*/
  5. delete all the simple predicates in  
 $Dep(\vec{x})$  containing  $w$ ; /\*  $w$  is redefined \*/  
/\* and is independent of its current value \*/
  6. add  $w - f(\vec{x}) = 0$  to  $Dep(\vec{x})$ ;
  7. **else** /\*  $w$  redefined yet depends on its current value \*/
  8. replace every  $w$  in  $Dep(\vec{x})$  with  $f^{-1}(w)$ ;  
/\* redefined  $w$  must satisfy updated constraint \*/
  9. **return**  $Assert := Dep(\vec{x}) \wedge R(\vec{x})$
- end**

Each assignment redefines a variable range whose interval is updated in Line 3. If a variable  $w$  is redefined in  $t.action$  and it is independent of its current value, then the relations associated with  $w$  in  $Dep(\vec{x})$  are no longer valid and should be removed in Line 5. On the other hand, a new assertion  $w - f(\vec{x}) = 0$  is added into the assertion set in Line 6. However, if it also depends on the current value, i.e.,  $w := f(w, u, v, \dots)$ , where  $w$  on the left side is the redefined value and that on the right side is the current value, we use the inverse function  $f^{-1}(w)$  to replace the current  $w$  in  $Dep(\vec{x})$  by the redefined  $w$  value in Line 8. This inverse function is the connection between the variable values before and after the assignment. Assume an assignment has the form of  $w := b * w + \sum_i a_i * x_i + c, b \neq 0$ . Then,

$$f^{-1}(w) = (w - c - \sum_i a_i * x_i) / b$$

For example,  $u = v + 2$  is in  $Assert$ , then  $v$  is redefined by an assignment  $v := v - 1$ . We use  $f^{-1}(v) = v + 1$  to replace  $v$  in  $Assert$ , and get  $u = v + 3$ , which represents the new relationship of  $u$  and  $v$ .

### 3.4 An Example

We use an example EEFMSM in Fig. 4 to illustrate the execution of Algorithm 2 using Interval Refinement. Here each state ignores any unspecified input and stays unchanged. Table 1 illustrates the execution of Algorithm 2 step by step upon the observed event sequence  $?c(8) !a(11) ?a(6) !d(11)$ . Note that a CC is represented in a form  $\langle s, R(\vec{x}), Dep(\vec{x}) \rangle$ .

Initially, there are 7 CCs; each CC corresponds to a state in the EEFMSM along with an empty assertion set. The initial interval of each variable is set according to its declaration and denoted by ‘-’ in the table.

According to Algorithm 2, from the initial set of CCs, upon input event  $?c(8)$ , CC  $\langle S1, -, \phi \rangle$ ,  $\langle S4, -, \phi \rangle$ ,  $\langle S5, -, \phi \rangle$ ,  $\langle S6, -, \phi \rangle$ ,  $\langle S7, -, \phi \rangle$  generate the same successor  $\langle S1, -, \phi \rangle$ , while CC  $\langle S2, -, \phi \rangle$  remains unchanged. CC  $\langle S3, -, \phi \rangle$

Table 1: Execution Steps of Algorithm 2

#	Event	set of CC
0		$\langle S1, -, \phi \rangle$ $\langle S2, -, \phi \rangle$ $\langle S3, -, \phi \rangle$ $\langle S4, -, \phi \rangle$ $\langle S5, -, \phi \rangle$ $\langle S6, -, \phi \rangle$ $\langle S7, -, \phi \rangle$
1	$?c(8)$	$\langle S1, -, \phi \rangle$ $\langle S2, -, \phi \rangle$ $\langle S4, \{R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]\}, \{u > v \wedge seq \equiv u + v \wedge up \equiv seq + 8\} \rangle$ $\langle S4, \{R(u)=[0,7], R(v)=[0,7], R(seq)=[5,19], R(up)=[13,27]\}, \{u \leq v \wedge seq \equiv 19 - u - v \wedge up \equiv seq + 8\} \rangle$
2	$!a(11)$	$\langle S3, \{R(v)=11, R(ctr)=0\}, \phi \rangle$ , $\langle S5, \{R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19\}, \{u > v \wedge 11 \equiv u + v\} \rangle$ , $\langle S5, \{R(u)=[1,7], R(v)=[1,7], R(seq)=11, R(up)=19\}, \{u \leq v \wedge 8 \equiv u + v\} \rangle$
3	$?a(6)$	$\langle S3, \{R(v)=11, R(ctr)=0\}, \phi \rangle$ , $\langle S2, \{R(seq)=11, R(up)=19, R(u)=6\}, \{v \neq 6\} \rangle$ , $\langle S6, \{R(u)=2, R(v)=6, R(seq)=11, R(up)=19\}, \phi \rangle$
4	$!d(11)$	$\langle S7, \{R(u)=2, R(v)=6, R(seq)=11, R(up)=19, R(ctr)=0\}, \phi \rangle$

generates the following two successor CCs,  $\langle S4, \{R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]\}, \{u > v \wedge seq \equiv u + v \wedge up \equiv seq + 8\} \rangle$  and  $\langle S4, \{R(u)=[0,7], R(v)=[0,7], R(seq)=[5,19], R(up)=[13,27]\}, \{u \leq v \wedge seq \equiv 19 - u - v \wedge up \equiv seq + 8\} \rangle$ . The CC set in step 2, 3 and 4 can be calculated following the same procedure. Clearly, after 4 steps only one CC is left and all the variable values have been decided.

Now we show in detail how the successor CC  $\langle S5, \{R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19\}, \{u > v \wedge 11 \equiv u + v\} \rangle$  in step 2 is generated from the CC  $\langle S4, \{R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]\}, \{u > v \wedge seq \equiv u + v \wedge up \equiv seq + 8\} \rangle$  in step 1. First the current state in the CC  $\langle S4, \{R(u)=[1,7], R(v)=[0,6], R(seq)=[1,13], R(up)=[9,21]\}, \{u > v \wedge seq \equiv u + v \wedge up \equiv seq + 8\} \rangle$  is  $S4$ . Upon input event  $!a(11)$ , the transition T8 can be executed only when the predicate  $[x \equiv seq]$  is evaluated not FALSE. Suppose that is the case. Then we add  $[11 \equiv seq]$  to the assertion of the current CC (the current value of parameter  $x$  is 11) and then check if it is executable by subroutine *Eval*. The  $Dep'(\vec{x})$  under check is  $\{u > v \wedge seq \equiv u + v \wedge up \equiv seq + 8 \wedge seq \equiv 11\}$ . We can easily infer that the value of variable  $seq$  is 11 and that of  $up$  is 19. Consequently,  $Dep'(\vec{x})$  is simplified to be  $\{u > v \wedge u + v \equiv 11\}$ .

Table 2: Feasibility Check and Interval Refinement

#	$u - v > 0$	$u + v \equiv 11$	Intervals
0	N/A	N/A	$R(u)=[1,7]$ $R(v)=[0,6]$
1	$R(u)=[(1, +\infty) - [-6,0]]$ $\cap [1,7]=[1,7]$ $R(v)=[(1, +\infty) - [1,7]) / -1] \cap [0,6]=[0,6]$	$R(u)=[(11,11) - [0,6]]$ $\cap [1,7]=[5,7]$ $R(v)=[(11,11) - [5,7]]$ $\cap [0,6]=[4,6]$	$R(u)=[5,7]$ $R(v)=[4,6]$
2	$R(u)=[(1, +\infty) - [-6,-4]]$ $\cap [5,7]=[5,7]$ $R(v)=[(1, +\infty) - [5,7]) / -1] \cap [4,6]=[4,6]$	$R(u)=[(11,11) - [4,6]]$ $\cap [5,7]=[5,7]$ $R(v)=[(11,11) - [5,7]]$ $\cap [4,6]=[4,6]$	$R(u)=[5,7]$ $R(v)=[4,6]$

Table 2 shows how *Eval* checks the feasibility of the assertion and refines the intervals of variables  $u$  and  $v$ . Af-

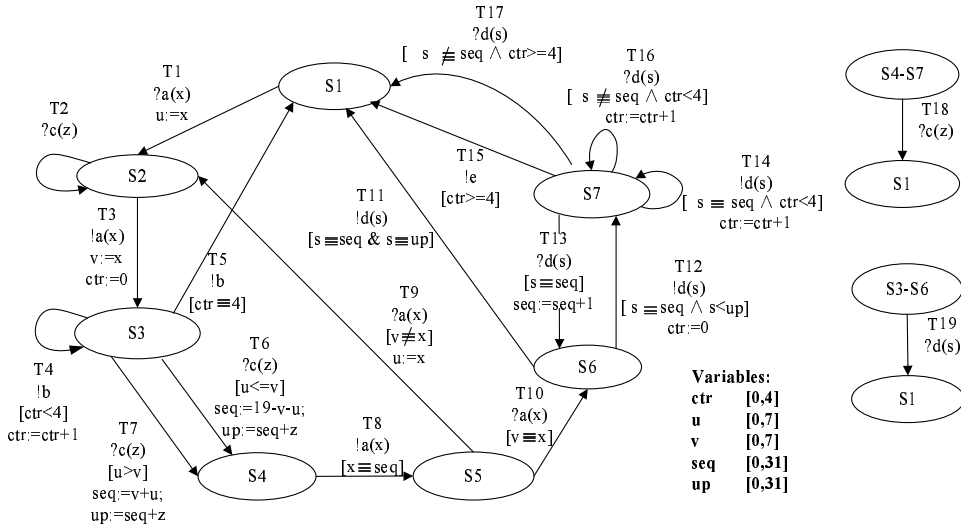


Figure 4: An Example EEFSM

ter 2 iterations, the intervals of variables  $u$  and  $v$  stay unchanged and the refinement has been completed with  $\{R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19\}$ , and no contradiction is detected. The end state of the transition T8 is S5, which results in the successor CC  $\langle S5, \{R(u)=[5,7], R(v)=[4,6], R(seq)=11, R(up)=19\}, \{u > v \wedge 11 \equiv u + v\} \rangle$  in step 2.

## 4 Experimental Results

We now report the experimental results by applying our passive testing algorithms on the OSPF [23] neighbor state machine and TCP [24]. OSPF and TCP represent different types of network protocols. OSPF is a distributed routing protocol for the intra-domain Internet routing while TCP is an end-to-end transport protocol. OSPF uses *hello* messages to keep the adjacency relation between two neighboring routers. The messages exchanged between OSPF neighbor state machines are very much alike, thus it is not easy to get homing in the experiment. The messages exchanged between two TCP entities are quite different from each other, such as *SYN* and *FIN* packets, and it is rather easy for TCP to reach homing in the experiment.

Typically TCP is embedded inside a protocol stack, providing services to the upper application layer, and, consequently, an upper layer tester is needed in the active testing of TCP, which means a higher cost compared to passive testing. We show that a good transition coverage can be obtained using our passive testing algorithm, and faults can be detected effectively yet without invoking an upper layer active tester.

### 4.1 Experiments on OSPF Neighbor State Machine

OSPF is a widely used intra-domain routing protocol based on the Open Shortest Path First algorithm. An OSPF

neighbor state machine is used to maintain connections between two neighboring OSPF routers and to exchange Link State Advertisements (LSA). Variables, such as sequence numbers, are used to record the current status of the connections.

We implemented three passive testing algorithms to compare their fault detection capabilities. They are Algorithm 1 and Algorithm 2 presented in this paper and the algorithm for nondeterministic FSM (NDFSM) published in [2]. We applied these algorithms in the OSPF neighbor state machine testing. Note that the algorithm in [2] is on NDFSM only. For a comparison, we transform the EEFSM into an NDFSM by extracting the variables from the specification, i.e., we ignore the data portion and only focus on the control portion of the protocol. The transformed NDFSM is in Appendix C.

The experiments are conducted in an experimental environment as in Fig. 5. In the experimental network, we used *Socrates* [25] to generate link state information. *Socrates* is a software tool developed at Bell Labs. It can simulate an OSPF network and exchange OSPF link state information with routers. OSPF packets are captured from a conversation between a Cisco router and the Router Under Test (RUT). Then the OSPF packets are decoded and sent to the Tester. This test architecture can work in a production network and detect faults on-line without disconnecting the RUT.

We examine the efficiency of various passive testing algorithms with regard to *state homing* and *variable homing*. With the NDFSM model, state homing is not obtained in any of the 25 cases. There are 8 cases out of the 25 experiments that Algorithm 1 completed the homing phase, and 14 cases out of 25 that Algorithm 2 completed the homing phase. The experimental results are shown in Table 3.

From the statistics, we can see that the NDFSM model is inappropriate for the passive testing of the behavior of OSPF. Passive testing algorithms using EEFSM model are

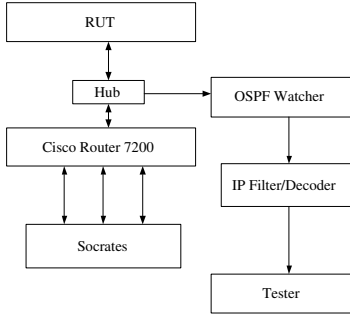


Figure 5: Experiment Environment

Table 3: Experimental Results of OSPF

Algorithm	State Homing	Variable homing
NDFSM algorithm	0/25	—
Algorithm 1	8/25	8/25
Algorithm 2	14/25	14/25

much more efficient than the NDFSM model, since the former monitors both control and data portion. Algorithm 2 uses Intervals to record the possible variable value ranges and uses assertion to record the relations among variables. Therefore, it has a better knowledge of the state machine and is more powerful in fault detection, comparing with Algorithm 1.

In the experiments, among the homing cases, Algorithm 2 takes on average 4.4 steps to achieve state homing, and takes on average 11 steps to obtain variable homing. The number of possible CCs in Q1 never exceeds 10. Thus this algorithm can detect faults in the OSPF neighbor state machine rather quickly.

## 4.2 Experiments on TCP State Machine

TCP is the dominant transport layer protocol in Internet. It is typically embedded inside a protocol stack where it supplies service primitives to the upper layers. Consequently, it is hard to conduct active testing, since we have rather limited controllability of TCP itself and have to get upper layers involved for test execution. We now present our passive testing experiments on TCP.

TCP contains a set of variables to record its state information. On the other hand, it allows nondeterministic behaviors, such as piggyback and delay acknowledgement. We use the following integer variables in the experiment according to [24] and [26].

$snd\_una$  : send unacknowledged  
 $snd\_nxt$  : send next  
 $snd\_wnd$  : send window  
 $snd\_wl1$  : segment sequence number used for last window update  
 $snd\_wl2$  : segment acknowledgment used for last window update  
 $iss$  : initial send sequence number  
 $rcv\_nxt$  : receive next  
 $rcv\_wnd$  : receive window  
 $irs$  : initial receive sequence number  
 $seg\_seq$  : sequence number of the current segment  
 $seg\_ack$  : acknowledgment number of the current segment  
 $seg\_len$  : length of the current segment  
 $last\_acked$  : acknowledgement sent out in the last segment

The sequence number  $seg\_seq$  is a *Serial Number* as de-

defined in RFC1982[27]. For sequence number, addition and comparison use modulo operations. In our implementation, we use a trick to remove the modulo operations. In our experiments, the TCP payload in a connection is not larger than  $2^{32}$ . When the sequence number in a TCP connection wraps, that is, it reaches the upper bound  $2^{32} - 1$ , we shift all the sequence numbers across the boundary. With this shift, all the sequence numbers can be treated as unsigned integer values without modulo.

The TCP state diagram is in Fig. 6. The EEFSM transition table derived from this state diagram contains 33 states and 102 transitions. The example shows that the state machine in state *Established* sends out a packet with *FIN*, updates its variables and then goes to state *Fin\_Wait\_1*. Note that the parameters with prefix *pkt* are the parameters of the TCP segment, e.g.  $pktFIN$  is the *FIN* bit of the segment.

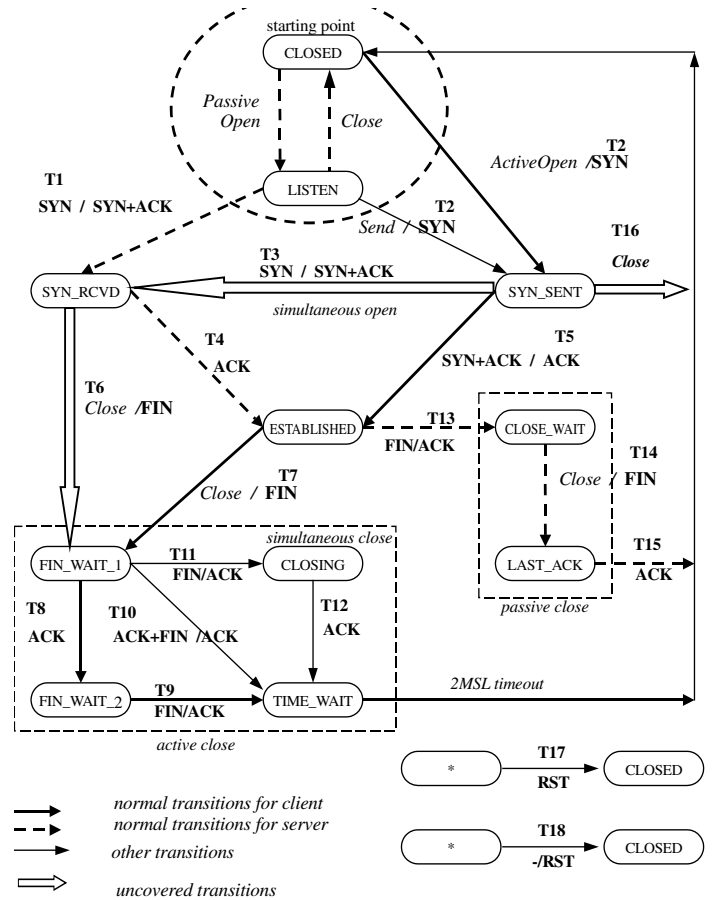


Figure 6: TCP State Diagram and Transition Coverage

start state : *Established*  
 event : TCPOUT  
 predicate :  $pktSeq \equiv snd\_nxt \wedge pktACK \equiv 1 \wedge pktFIN \equiv 1 \wedge pktAckNum \geq last\_acked \wedge pktAckNum \leq rcv\_nxt$   
 action :  $snd\_nxt := snd\_nxt + pktLen;$   $rcv\_wnd := pktWND;$   
 $last\_acked := pktAckNum;$   
 end state : *Fin\_Wait\_1*

*TCPdump* is used to capture the TCP packets to and from a Linux server, which is the implementation under test (IUT) in the experiment, providing sendmail, POP3, proxy, telnet and ftp services. It works as a TCP server when someone telnets it, and it works as a TCP client when it forwards HTTP requests. In the experiment, we collect all the TCP packets to and from the server and monitor every connection to detect possible faults.

TCP uses special messages, such as *SYN* and *FIN*, to trigger its transitions. Thus it is easy to acquire homing in passive testing experiments. In almost all of our experiments, the state machine of TCP obtains state homing and variable homing quickly.

Fault identification is critical as well as fault detection in network fault management. To repeat the fault detected, we need to know the configuration, from which the fault occurs. Also it is desirable to have information to take the system to this configuration. Our passive testing procedure monitors and records the observed input/output messages of the system to provide such information for fault location and identification.

Passive testing algorithms can be separated into two phases, the homing phase and the fault detection phase. Before homing, there is a set of candidate configurations and there may be several transitions matching the current event. When more precise information is acquired, some candidate transitions in the previous stages can be removed. See the example in fig 7. The machine gets homing in the  $i+2$  level. Backtracking from configuration C32, the previous configurations must be C22 and C12, since transitions T2 and T3 can be executed in the scenario while T1 can never be.

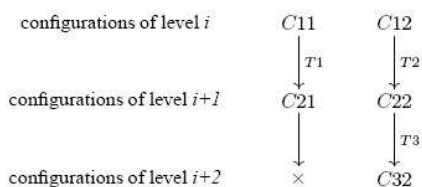


Figure 7: backtracking from level  $i+2$  to  $i$

In addition to fault detection and identification, we want to find out which transitions are covered by passive monitoring. This will be helpful in verifying the operation of the system. The transition coverage in our passive testing of TCP is as follows.

1. When IUT acts as a server, the following transitions are covered: T1, T4, T13, T14, T15, T17, T18
2. When IUT acts as a client, the following transitions are covered: T2, T5, T7, T8, T9, T11, T12, T16, T17, T18
3. There are some transitions uncovered : T3, T6, T16  
T3 implies that the two peers open simultaneously. It would never occur in client-server services. T6 occurs when the setup procedure is terminated by the user application before it enters state *Established*. T16 just sends an *SYN* and does not wait for a reply; it acts

like a denial of service attack and does not occur in this experiment.

From the results we can see that the transition coverage of passive testing is quite satisfactory.

We have studied several TCP/IP protocols including TCP, PPP, BGP, OSPF, RIP, HTTP, FTP and Neighbor Discovery (IPv6). Among these protocols, TCP has more variables and complicated processes, although the variable dependencies are not that complex. This is because TCP/IP protocols are designed according to the KISS rule (Keep It Simple, Stupid).

Although a protocol may be simple and cannot display the effectiveness of MILP algorithm, new or other applications may need this algorithm. A possible usage is multi-protocol analysis for network performance analysis. Multi-protocol analysis, like some works using Nprobe[16], considers different layer protocols to analyze network performance or to locate network faults. The analysis model has more variables and more involved dependencies. For example, HTTP response latency is related to TCP congestion and/or QOS definitions. We believe that our algorithm might be helpful for such applications. This is a future research topic.

## 5 Conclusion

In a unit testing, a protocol system can be isolated and tested by actively applying inputs to reveal faults from the output responses. However, for a system in operation in a networked or integrated environment, often we could only passively observe and monitor the system behaviors to detect faults, and that naturally leads to the passive testing research and development activities.

In this paper, several passive testing algorithms on the EEFSM model and their applications to OSPF and TCP state machine are presented. The deficiencies of existing algorithms are studied, and our algorithms prove to be much more effective in tracing the state and the variables values for detecting faults. We use symbolic logic methods to deal with the predicates and use assertions to record the relations among variables. The idea behind our approach is to refine the valid variable value ranges using as much information from the system specifications as possible, and that efficiently detects system faults yet without interrupting its normal operations. On the other hand, passive testing can also deal with embedded protocol system testing with a low cost and reasonable fault coverage.

After faults are detected by either passive or active testing, system fault diagnosis is required to locate the faults for fault correction and system maintenance. Yet little is known on efficient fault location methods, which are applicable to practical protocol systems.

## Acknowledgment

A preliminary version of this work was presented at the IEEE International Conference on Network Protocols un-

der the title, "A Formal Approach for Passive Testing of Protocol Data Portions", in Conference Proceedings, pp 122-131. We thank the anonymous reviewers for their insightful and constructive comments and suggestions.

We thank Xiaoliang Wei from Tsinghua University for helping us with the experiments on the OSPF.

## References

[1] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. *Proc. of the IEEE*, 84:1090–1123, Aug 1996.

[2] David. Lee, A. N. Netravali, K. K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. *Proceedings of IEEE International Conference on Network Protocols*, pages 113–122, October 1997.

[3] A. Bouloutas, G. Hart, and M. Schwartz. On the design of observers for failure detection of discrete event systems. In *IEEE Workshop on Network Management*, Tarrytown, NY, Sep 1989.

[4] C. Wang and M. Schwartz. Fault detection with multiple observers. *IEEE/ACM Transactions on Networking*, 1:48–55, 1993.

[5] K. Vijayanada Noubir and H. J. Nussbaumer. Signature-based method for run-time fault detection in communication protocols. *Computer Communications Journal*, 21(5), 1998.

[6] R.E. Miller. Passive testing of networks using a cfsm specification. *Proceedings of the IEEE International Performance, Computing and Communications Conference*, pages 111–116, February 1998.

[7] R.E. Miller and K. A. Arisha. On fault location in networks by passive testing. *Proceedings of the IEEE International Performance, Computing and Communications Conference*, pages 281–287, February 2000.

[8] B. Alcalde, A. Cavalli, D. Chen, D. Hhuu, and D. Lee. Passive testing on fsm by variable determination and backward checking. *Proceedings of FORTE*, 2004.

[9] S. Jaiswal, G. Iannaccone, C. Diot, J. F. Kurose, and D. Towsley. Inferring tcp connection characteristics through passive measurements. *Proceedings of INFOCOM*, 2004.

[10] M. Tabourier and A. Cavalli. Passive testing and application to the gsm-map protocol. *Information and Software Technology*, 41:813–821, September 1999.

[11] D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. *Proceedings of ICNP*, pages 122–131, 2002.

[12] W. Wang A. A. Lazar and R. H. Deng. Models and algorithms for network fault detection and identification: A review. In *ICCS/ISITA*, pages 999–1003, 1992.

[13] Cynthia S. Hood and Chuanyi Ji. Proactive network fault detection. In *INFOCOM (3)*, pages 1147–1155, 1997.

[14] R. W. Buskens and R. P. Bianchini, Jr. Distributed on-line diagnosis in the presence of arbitrary faults. In *Proc. 23rd Int. Conf. on Fault-Tolerant Computing (FTCS-23)*, pages 470–479, Toulouse, France, 1993. IEEE Computer Society Press.

[15] N. Anerousis, R. Caceres, N. Duffield, A. Feldmann, A. Greenberg, C. Kalmanek, P. Mishra, K. Ramakrishnan, and J. Rexford. Using the at&t labs packet scope for internet measurements, design, and performance analysis, 1997.

[16] James Hall, Ian Pratt, and Ian Leslie. Observing web browser behaviour using the nprobe passive monitoring architecture. In *Cabernet*, 2001.

[17] Othmar Kyas. *Network Troubleshooting*. Agilent Technologies, Sep 2000.

[18] <http://telephonyonline.com/ar> /telecom\_singing\_network\_blues/.

[19] R.E. Miller, D.L. Chen, D. Lee, and R. Hao. Technical report - coping with nondeterminism in network protocol testing, 2004.

[20] Paul R. Thie. *An Introduction to Linear Programming and Game Theory*. John Wiley & Sons, 2nd edition, 1988.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Cliff Stein. *Introduction to Algorithms*. MIT Press, McGraw-Hill, New York, NY, second edition, 2001.

[22] Ramon E. Moore. *Methods and Applications of Interval Analysis*. Society for Industrial & Applied Mathematics, Philadelphia, 1979.

[23] J. Moy. *RFC 2328 OSPF Version 2*. 1998.

[24] Information Sciences Institute, University of Southern California. *RFC793, TRANSMISSION CONTROL PROTOCOL*, 1981.

[25] R. Hao, D. Lee, R. K. Sinha, and D. Vlah. Testing ip routing protocols - from probabilistic algorithms to a software tool. *Proceeding of FORTE-2000*, pages 249–264, October 2000.

[26] D. Comer and D. Stevens. *Internetworking With TCP/IP Volume II: Design, Implementation, and Internals (Second Edition)*. Prentice Hall, 1994.

[27] R. Elz and R. Bush. *RFC 1982 Serial Number Arithmetic*. 1996.

[28] <http://blrc.china.bell-labs.com/groups/ngi/projects/passive/>.

## Appendix

### A BNF of Predicate and Action

- Integer expression is the atomic part of a predicate and an action.

```

INTEGER_EXPRESSION ::=
    INTEGER
    | INTEGER_VARIABLE
    | EVENT_PARAMETER
    | (INTEGER * INTEGER_VARIABLE)
    | (INTEGER_EXPRESSION + INTEGER_EXPRESSION)
    | (INTEGER_EXPRESSION - INTEGER_EXPRESSION)

```

- The predicates are logic expressions. A Simple predicate is defined as follow.

```

p ::=
  TRUE
  | FALSE
  | (INTEGER_EXPRESSION ≡ INTEGER_EXPRESSION)
  | (INTEGER_EXPRESSION > INTEGER_EXPRESSION)
  | (INTEGER_EXPRESSION < INTEGER_EXPRESSION)
  | (INTEGER_EXPRESSION > INTEGER_EXPRESSION)
  | (INTEGER_EXPRESSION < INTEGER_EXPRESSION)
  | (INTEGER_EXPRESSION ≠ INTEGER_EXPRESSION)

```

A predicate is defined recursively.

$P ::= p \mid (P \wedge P) \mid (P \vee P)$

- Each action is a sequence of assignments.  
 ACTION ::= ASSIGNMENT; | ASSIGNMENT; ACTION | NULL  
 ASSIGNMENT ::= VARIABLE:=INTEGER\_EXPRESSION

## B Operations on Integer Variable Values Denoted by Intervals

This section gives the definitions of evaluations of integer and logical expression using Intervals based on [22].  $X$  and  $Y$  are abbreviations for interval  $[\underline{X}, \overline{X}]$  and  $[\underline{Y}, \overline{Y}]$  respectively.

**Definition B.1 (Interval Addition)**  $[\underline{X}, \overline{X}] + [\underline{Y}, \overline{Y}] = [\underline{X} + \underline{Y}, \overline{X} + \overline{Y}]$

**Definition B.2 (Interval Subtraction)**  $[\underline{X}, \overline{X}] - [\underline{Y}, \overline{Y}] = [\underline{X} - \overline{Y}, \overline{X} - \underline{Y}]$

**Definition B.3 (Interval Intersection)**

$$X \cap Y = \begin{cases} \phi & : \underline{X} > \overline{Y} \text{ or } \underline{Y} > \overline{X} \\ [\max(\underline{X}, \underline{Y}), \min(\overline{X}, \overline{Y})] & : \text{Otherwise} \end{cases}$$

**Definition B.4 (Evaluation of Integer Expression  $f(\vec{x})$ )**

The evaluation of integer expression  $f(\vec{x})$  using interval  $R(\vec{x})$  is denoted by  $f(\vec{x})|_{R(\vec{x})}$ , the evaluation is conducted using the Interval addition and Interval subtraction operations defined above, and the result is also an Interval.

**Definition B.5 (Interval Inclusion)**  $X \subseteq Y$  if and only if  $\underline{Y} \leq \underline{X}$  and  $\overline{X} \leq \overline{Y}$ .

**Definition B.6 (Interval Logical Operations)**

Table 4 defines the interval logical operations, which are all tri-valued functions.

**Definition B.7(Truth Table of Tri-valued Logic)**

The truth table of tri-valued logic is defined by adding the evaluation of POSSIBLE to classical boolean logical evaluation. The added evaluations are defined in Table 5.

**Definition B.8 (Evaluation of Logical Expression  $f(\vec{x})$ )**

The evaluation of logical expression  $f(\vec{x})$  using interval  $R(\vec{x})$  is denoted by  $f(\vec{x})|_{R(\vec{x})}$ , the evaluation is conducted using the Interval logical operations and Truth table for tri-valued logic defined above.

Table 4: Interval Logical Operations

Logic expression	result	Condition
$X \equiv Y$	TRUE	$\overline{X} \equiv \underline{X} \equiv \overline{Y} \equiv \underline{Y}$
	FALSE	$\underline{X} > \overline{Y}$ or $\underline{Y} > \overline{X}$
	POSSIBLE	$\underline{X} < \underline{Y}$ and $\underline{Y} < \overline{X}$ and not $X \equiv \underline{X} \equiv Y \equiv \underline{Y}$
$X \geq Y$	TRUE	$\underline{X} > \underline{Y}$
	FALSE	$\overline{X} < \underline{Y}$
	POSSIBLE	$\underline{X} < \underline{Y}$ and $\underline{Y} < \overline{X}$
$X > Y$	TRUE	$\underline{X} > \underline{Y}$
	FALSE	$\overline{X} < \underline{Y}$
	POSSIBLE	$\underline{X} < \underline{Y}$ and $\underline{X} > \underline{Y}$
$X \leq Y$	TRUE	$\overline{X} < \underline{Y}$
	FALSE	$\underline{X} > \overline{Y}$
	POSSIBLE	$X > \underline{Y}$ and $\underline{X} < \overline{Y}$
$X < Y$	TRUE	$\overline{X} < \underline{Y}$
	FALSE	$\underline{X} > \overline{Y}$
	POSSIBLE	$\underline{X} > \underline{Y}$ and $\underline{X} < \overline{Y}$
$X \neq Y$	TRUE	$\underline{X} > \overline{Y}$ or $\underline{Y} > \overline{X}$
	FALSE	$X \equiv \underline{X} \equiv Y \equiv \underline{Y}$
	POSSIBLE	$\underline{X} < \underline{Y}$ and $\underline{Y} < \overline{X}$ and not $X \equiv \underline{X} \equiv Y \equiv \underline{Y}$

Table 5: Truth Table of Tri-valued Logic

POSSIBLE	$\wedge$	TRUE	=	POSSIBLE
POSSIBLE	$\wedge$	FALSE	=	FALSE
POSSIBLE	$\wedge$	POSSIBLE	=	POSSIBLE
POSSIBLE	$\vee$	TRUE	=	TRUE
POSSIBLE	$\vee$	FALSE	=	POSSIBLE
POSSIBLE	$\vee$	POSSIBLE	=	POSSIBLE

## C The Specification of the OSPF Neighbor State Machine

- The EEFSM model

There are a total of 84 transitions in the EEFSM model of the OSPF neighbor state machine in our experiment. The full version of the EEFSM specification can be found from [28]. One of the transitions is shown below. This transition is from state *Exchange* to state *Loading*. The event that triggered this transition is a Data Description packet.

```

start state : Exchange
event : DD_rcv(RouterID, FlagI, FlagM, DDSeqNum)
predicate : FlagI ≡ 0 ∧ my_MS ≡ 1 ∧ my_M+FlagM ≡ 0 ∧
            DDSeqNum ≡ my_seq ∧ my_DDsent ≡ 1
action : my_NRID:=RouterID; my_seq:=DDSeqNum+1;
         ne_M:=FlagM; my_DDsent:=0;
end state : Loading

```

- The NDFSM model

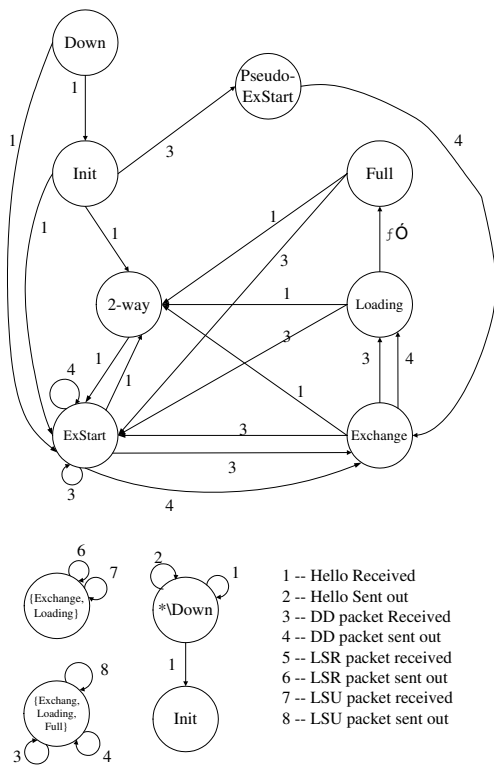


Figure 8: NDFSM Model