Formal Modeling and Verification of Management on a Group of Network Security Appliances

Moonzoo KIM [1]
Eun-Hye CHOI [2]

1: POSTECH, Korea 2: AIST CVS, Japan

Formal Modeling and Verification of Management on a Group of Network Security Appliances

Moonzoo Kim[1] and Eun-Hye Choi[2]

[1] CSE Dept. Pohang University of Science and Technology
Pohang, South Korea
moonzoo@postech.ac.kr
[2] Research Center for Verification and Semantics, AIST
Osaka, Japan
e.choi@aist.go.jp

Abstract. One of the prerequisites for information society is *secure* and *reliable* communication among computing systems. Accordingly, network security appliances become key components of infrastructure, not only as security guardians, but also as reliable network components. Thus, for both *fault tolerance* and *high network throughput*, multiple security appliances are often deployed together in a group. In this paper, we present our experience of formally modeling and verifying a group management protocol for network security appliances using the Spin model checker. To analyze the reliability of the protocol, we classified and modeled various types of faults and analyzed the protocol in the presence of combination of these faults. We could detect several design flaws of the protocol through this project.

1 Introduction

Information society of our age utilizes large number of information applications such as mobile banking, tele-conferencing, and online stock trading systems, for which communication takes a central role. In addition to basic communication services, it becomes essential to provide quality services such as *security* and *reliability*. For these purposes, more security appliances such as firewall, VPN, and IDS/IPS are deployed in various networks. Accordingly, their reliability, not only as security guardians, but also as network components, becomes a critical issue. Therefore, for both *fault tolerance* and *high network throughput*, multiple security appliances are often deployed together in a group and managed via *High-Availability (HA) protocol* among the appliances.

Distributed protocols [12, 3, 14] such as HA protocol take important roles in networked information applications and they are notorious for their difficulty to design correctly. It is a highly challenging task to consider all possible communication scenarios among distributed information security applications. If system failures should also be considered, complexity of protocol designs increases further. Thus, it is often observed that a group of network security appliances exhibits abnormal behaviors such as decreased throughput or dropped normal packets while a single security appliance works well without problems. This abnormal behavior of network security appliances must be prevented at all cost because such appliances often work as gateways between internal and external networks and malfunction of the appliances can crash an entire internal network. In addition, testing a group of network security appliances requires great efforts due to complex network/machine configurations and large number of test scenarios. Furthermore, it is difficult to determine whether misbehaviors are due to errors of HA protocol or due to other factors such as OS/HW failure and/or misconfiguration of networks, etc. Therefore, as formal method techniques have been actively applied to networked applications [5, 8, 7], *formally modeling and verifying* HA protocol with support of an automated analysis tool (e.g. model checker) can enhance the reliability of network security appliances, thus improve the overall security of the network effectively.

This paper presents our experience of formally modeling and verifying the HA protocol used in a commercial network security appliance. By modeling various fault classes and applying model checking, we can check *all* possible behaviors of the security appliances in a group and detect design flaws of the HA protocol that would be hard to find otherwise and thus cause serious security break. First, we classify various possible fault classes considered in this case study and describe how to model these faulty behaviors in a formal modeling language. Second, we formulate requirement properties for the HA protocol. Then, we build an abstract model of the HA protocol in Promela [6] and verify the model using the Spin model checker [2] to detect flaws of the HA protocol. With this formal modeling and verification, we could detect several flaws in the HA protocol and suggest a solution for enhancing reliability of security appliances.

Section 2 overviews the HA protocol. Section 3 overviews Spin and its modeling language Promela. Section 4 describes classification of fault classes and corresponding modeling approaches. Section 5 explains requirement properties and a HA model as well as abstraction techniques we applied. Section 6 shows the results of verifying the HA model. Finally, section 7 summarizes this paper.

2 Overview of the HA Protocol

In order to increase total network throughput, each network security appliance in a group does not passively stay as a standby backup machine, but actively handles network traffic distributed to it [1]. Thus, for multiple network security appliances to handle communication sessions correctly (especially for asymmetric packet routings), they should share session information. Whenever a new session is created via machine 1, machine 1 broadcasts this session information (e.g., source/destination IP's, port number, and protocol) to other machines in the group through a dedicated HA network that is separated from regular traffic network for high priority and reliability. Also, each machine broadcasts updated session information to other machines constantly.

Furthermore, in order to manage a group of network security appliances, one security appliance in the group is designated as a *master* to manage the other *slaves*. Initially, a master is statically designated by a network administrator. The main tasks of a master for the HA protocol are as follows. Besides these tasks, a master also takes charge of configuring network, writing event logs to a log server, etc.

– *Addition of slaves* (see Fig. 1.a))
  When a slave becomes operational, the slave broadcasts `join_request` messages until it receives `join_permit` message from a master. Once the master allows the slave to join the group by sending `join_permit` message to the slave, the master broadcasts new information on network security appliances of the group (`members_info`). Then, the master sends all session informations and packet classification rules to the slave via `update_session_rule<n>` for the slave to work as a member of the group.
– *Assignment of a backup master*
  To prepare a case that a master crashes, the master assigns a slave as a backup master that becomes a master when the master crashes. For backup master assignment, a master sends an assignment message `bkup_m_assign` to a slave that is elected as a backup master.
– *Deletion of slaves* (see Fig. 1.b))
  A master constantly checks status of slaves by broadcasting `m_alive` every second and receiving `s_alive` from every slave. If a master does not receive `s_alive` from a slave for three seconds, the master erases information on the slave and broadcast newly updated member information (`members_info`) to the group. If the erased slave is a backup master, the master elects other slave as a backup master and sends `bkup_m_assign` to the slave.

As depicted in Fig. 1.c), a backup master constantly checks whether a master is operational or not by receiving `m_alive`. If a backup master does not receive `m_alive` for three seconds, the backup master sends `query_m_alive` three times to the master (one per each second). If the backup master does not receive a response from the master, the backup master becomes a master and broadcasts new status in `members_info` message. Then, the backup master assigns another slave as a new backup master by sending `bkup_m_assign` and starts broadcasting `m_alive` messages. A master or a backup master works as a slave when it recovers from failure. An exception is that machine 0, which is statically designated as a master by a network administrator, may work as a master if there exists no master when it recovers from failure.

3 Overview of the Spin Model Checker

Model checking is a formal technique to build a system model and systematically explore whole state space generated from the model in order to check whether or not given requirement properties hold. Spin [2] is a model checker that generates a finite state space from a model in Promela [6] and verifies whether or not the model satisfies given linear temporal logic (LTL) properties.

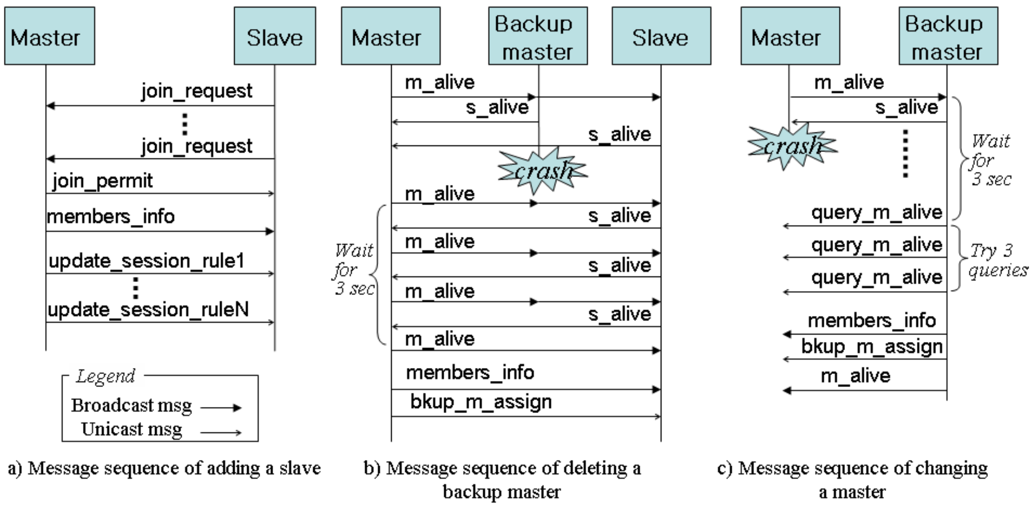


Fig. 1. Message sequences regarding HA activities

3.1 Overview of Spin

Spin is a general purpose verification tool to check properties of concurrent/distributed systems. If a target system model does not satisfy requirement properties, then Spin delivers a *counter example* that can be analyzed by simulation feature of Spin. A model can be executed/simulated in a random or guided or interactive way by Spin. Simulations help system designers to analyze behaviors of a system model with respect to a particular input stimuli and process scheduling. In addition, Spin can generate a message sequence chart to help human engineers to understand communication behaviors of the system visually.

3.2 Promela

Syntax of Promela are similar to those of conventional programming languages such as C or Java. Promela has, however, a formal semantics to generate full state space of a model rigorously. A Promela model consists of *processes*, *message channels* and *variables*. Processes are global objects and similar to processes/threads of programming languages. These processes communicate with each other using message channels or global variables. For example, lines 5-29 of Fig 6 define N `machine` processes. These N processes communicate with each other through channels declared at line 3 - `ch2slv[N]` whose buffer size is one and `ch2mst` whose buffer size is N. If a message channel becomes full, a sender blocks until the message channel has a room. Also, global variables `alive[N]` and `working[N]` at line 1 serve as means to identify status of other processes.

Processes of Promela consist of statements, which can be either declaration, assignment, expression, selection or loop. The selection statement contains execution sequences, each preceded by a double colon. Only one sequence from

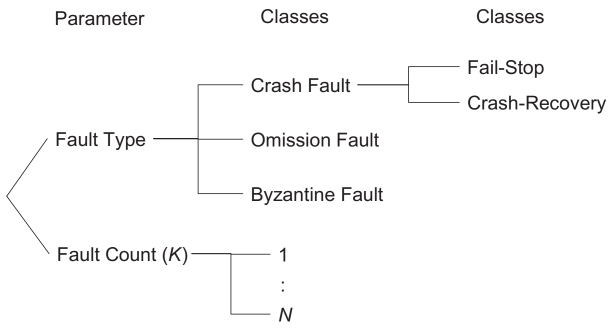


Fig. 2. Fault classification

the list can be executed only if its first statement (guard) is executable. If more than one guard is enabled, one of them is selected in a non-deterministic way. Loop is similar to the selection statement (see lines 10-28), except that the statement is executed repeatedly, until control is explicitly transferred to outside the statement by `goto` or `break` statements.

4 Classification and Modeling of Faults

In this section, we classify faults occurring in a distributed system and propose an environmental and modular modeling method for each of the classified fault classes. By combining the proposed fault modeling modules with a target system model, we can model and analyze various fault scenarios of the target system. There have been research to analyze fault-tolerant systems for specific faults using model checker [13, 4, 11]. To the best of our knowledge, however, this is the first time that a general and reusable modeling framework that covers general fault classes classified by the *fault type* and the *fault count* is proposed.

4.1 Classification of Faults

In order to analyze the HA protocol with the presence of various types of faults, we consider the fault classes shown in Fig. 2. Faults in a distributed system can be classified by various parameters. Among the parameters, we chose two parameters, *fault type* and *fault count*, for the classification of faults occurring in a group of network security appliances.

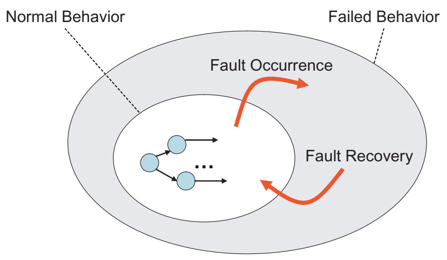
First, the *fault type* is the most traditional parameter for classifying faults in distributed systems and indicates the behavior of a process when faults are occurred in the process. According to the fault type, faults are classified into *crash fault*, *omission fault*, and *Byzantine fault*.

– Crash fault: causes a process to halt.
– Omission fault: causes a process to not respond to some requests, that is, fail to send some messages.

```
active [N] proctype machine() {

  do
        :: $<guard_1> \rightarrow <command_1>$ ;
        :: $<guard_2> \rightarrow <command_2>$ ;
                        ⋮
        :: $<guard_m> \rightarrow <command_m>$ ;
  od
}
```

(a) Target system model



(b) State space of a process

Fig. 3. Target system and state space of a process assuming faults

– Byzantine fault: causes a process to behave in a totally arbitrary manner in a broad sense. The arbitrary behavior of a process is often modeled as sending arbitrary messages to arbitrary processes.

The crash faults are furthermore classified into *fail-stop* and *crash-recovery*. The *fail-stop* fault causes a process to halt and not restart, while the *crash-recovery* fault causes a process to be recovered and restart after it is failed.

Next, the *fault count* indicates the number of fault occurrence, i.e. failed processes. In many cases, fault-tolerant distributed systems are analyzed assuming the maximum number of processes that can be in the failed states at the same time. Hereafter, $K(1 \leq K \leq N)$ denotes the maximum number of fault count where N denotes the number of processes in the network of security appliances.

4.2 Environmental Modeling of Faults

In this section, we give how to model each of the fault classes in Fig. 2 over a given target system which is modeled in Promela as shown in Fig. 3(a). Since distributed systems, including the network of security appliances, are sets of processes (machines) communicating each other by sending and receiving messages, a distributed system is naturally modeled in Promela as asynchronous processes communicating via message passing. Hence, target distributed protocols, such as the HA protocol, can basically be modeled in Promela as shown in Fig. 3(a).

Fig. 3(b) illustrates a local state space of a process assuming faults. Fault modeling, which means behavior modeling of the target system in the presence of faults, thus consists of modeling of fault occurrence, modeling of failed behavior, and modeling of fault-recovery.

Fig. 4 shows the proposed modeling for each fault class. For fault modeling, we introduce boolean variable `failed[_pid]` which is true if Process `_pid` is failed; otherwise false.

First, fault occurrence and fault-recovery is described as the following guarded commands using variable `failed[_pid]`:

```
bool failed[N];

active [N] proctype machine() {

  failed[_pid] = false;
  do
  /* normal behavior */
  :: !failed[_pid] && <guard1 > → <command1 > ;
  :: !failed[_pid] && <guard2 > → <command2 > ;
                 .
                 .
                 .
  :: !failed[_pid] && <guardm > → <commandm > ;

  /* fault occurrence */
  :: !failed[_pid] → failed[_pid] = true;

  /* failed behavior */
  :: failed[_pid] → ; /* do nothing */
  od
}
```

(a) Fail-stop

```
bool failed[N];

active [N] proctype machine() {

  failed[_pid] = false;
  do
  /* normal behavior */
  :: !failed[_pid] && <guard1 > → <command1 > ;
  :: !failed[_pid] && <guard2 > → <command2 > ;
                 .
                 .
                 .
  :: !failed[_pid] && <guardm > → <commandm > ;

  /* fault occurrence */
  :: !failed[_pid] → failed[_pid] = true;

  /* fault recovery */
  :: failed[_pid] → failed[_pid] = false;

  /* failed behavior */
  :: failed[_pid] → ; /* do nothing */
  od
}
```

(b) Crash-recovery

```
bool failed[N];

active [N] proctype machine() {

  failed[_pid] = false;
  do
  :: <guard1 > → <command1 > ;
  :: <guard2 > → <command2 > ;
                 .
                 .
                 .
  /* normal behavior */
  :: <guardm > → ch_i!msg_x ;

  /* fault occurrence */
  :: !failed[_pid] → failed[_pid] = true;

  /* failed behavior */
  :: failed[_pid] && <guardm > → ;
  od
}
```

(c) Omission fault

```
bool failed[N];

active [N] proctype machine() {

  failed[_pid] = false;
  do
  :: <guard1 > → <command1 > ;
  :: <guard2 > → <command2 > ;
                 .
                 .
                 .
  /* normal behavior */
  :: !failed[_pid] && <guardm > → ch_i!msg_x ;

  /* fault occurrence */
  :: !failed[_pid] → failed[_pid] = true;

  /* failed behavior */
  :: failed[_pid] && <guardm > → ch_j!msg_y ;
  od
}
```

(d) Byzantine fault

```
bool failed[N]; int h = 0;

active [N] proctype machine() {

  failed[_pid] = false;
  do
  /* normal behavior */

  /* failed behavior */

  /* fault occurrence */
  :: !failed[_pid] && h<K → failed[_pid] = true; h = h + 1;

  /* fault recovery */
  :: failed[_pid] → failed[_pid] = false; h = h - 1;
  od
}
```

(e) Fault count (K)

Fig. 4. Fault modeling

```
:: !failed[_pid] → failed[_pid] = true; /* fault occurrence */
:: failed[_pid] → failed[_pid] = false; /* fault-recovery */
```

As shown in Fig. 4, the former guarded command representing fault occurrence is commonly used for modeling the crash faults, the omission faults, and the Byzantine faults. The latter guarded command representing fault-recovery is used for modeling the crash-recovery faults. For modeling the fault count (K), we introduce integer variable `h` which denotes the number of current failed processes and describe changing the value of `h` with fault occurrence and fault-recovery as the following guarded commands:

```
:: !failed[_pid] && h<K → failed[_pid]=true; h=h+1; /* fault occurrence */
:: failed[_pid] → failed[_pid]=false; h=h-1;     /* fault-recovery */
```

Next, failed behaviors are modeled depending on the fault types. For the crash faults, failed behavior is modeled by adding the following guarded command which represents that the process does nothing when it is failed.

```
:: failed[_pid] → ; /*failed behavior of crash faults*/
```

For the omission faults and the Byzantine faults, failed behaviors are respectively modeled by adding the following guarded commands

```
:: failed[_pid] && < guard_i > → ;         /*failed behavior of omission faults*/
:: failed[_pid]&&< guard_i >→ch_j!msg_y;/*failed behavior of Byzantine faults*/
```

for every guarded command, $<$ $guard_i$ $>\rightarrow<$ $command_i$(=`ch_i!msg_x`) $>$, in Fig. 3(a) whose command part is a message sending statement. In Promela, `ch!msg` describes sending of message `msg` using communication channel `ch`. The former guarded command represents that the process sends nothing when it is failed, and thus it models the failed behavior of the omission faults. The latter guarded command represents that the process sends incorrect message `msg_y`($\neq$`msg_x`) to incorrect channel `ch_j`($\neq$`ch_i`) when it is failed, and thus it models the failed behavior of the Byzantine faults.

5 Modeling the HA protocol

In this section, we describe properties required for the HA protocol and a model of the HA protocol.

5.1 Requirement Properties

The first requirement property we are interested in is *single master* property, i.e., there must exist at most one master at any time. Without satisfying this property, a group of network security appliances may have inconsistent state information. Also, *deadlock-free* property is a basic requirement for most protocols. More importantly, we would like to check that the HA protocol achieves its goals - fault tolerance and high network throughput. First try of formulating the goals of the HA protocol may be to specify continuous network security services in the following way: [3]

$$P_{req'} = \Box(\exists i \in Group.(working(i)))$$

where *Group* indicates a group of machines connected via the HA protocol and *working*(i) means that the ith machine in the group is properly handling network sessions and HA communication. Satisfiability of this property, however, does *not* depend on the HA protocol only, but also on the *environment model* that describes failure behaviors of a machine. For example, if we allow random process/machine crashes in the environment model, $P_{req'}$ might not be satisfied regardless of correctness of the HA protocol.

We refine $P_{req'}$ into P_{req} so that P_{req} is *not* affected by the environment model that cannot be controlled by the HA protocol.

$$P_{req} = \Box(\forall i \in Group.(alive(i) \rightarrow \Diamond(working(i) \vee \neg alive(i))))$$

where *alive*(i) indicates that the ith machine is alive, but may not process the HA protocol correctly yet. In other words, *alive*(i) indicates that the ith machine may not join the group, but works as a single network security machine. P_{req} specifies that if any machine is alive, the machine eventually join the group unless it crashes. Thus, P_{req} incorporates both requirements on fault-tolerance and high network throughput by recovering crashed machines.

5.2 A Model of the HA protocol

Fig. 5 shows overview of the HA model in a finite state machine and Fig. 6 describes the HA model in Promela. Each machine, regardless of whether it is a master or a slave, is modeled as a process. Each machine starts from `machine_init` state (line 7 and 12 of Fig. 6). Initially, machine 0 (whose process id is 0) is statically designated as a master and the machine moves to `mst_init` state (line 13) to become a master. Then, the machine is working at `mst_acting` state (line 14) that is the core of the master procedure. A master performs the following tasks at `mst_acting`.

- To add a slave to the group (`add_slave` state (line 15))
- To assign a slave as a backup master (`bkupmst_assign` state (line 16))
- To delete a dead slave from the group (`del_slave` state (line 17))

Once a machine is determined as a slave, the machine moves to `slv_init` state (line 18) to initialize settings to become a slave. Then, the slave moves to `join_group` state (line 19) where the slave requests a permission to join the group from a master. Once the slave receives the permission from the master, the slave moves to `slv_acting` state (line 20). The slave becomes a master (`become_mst` at line 21) if it is a backup master and there exists no master.

[3] Intuitive meanings of $\Box$ and $\Diamond$ operators in LTL are "always" and "eventually", respectively.

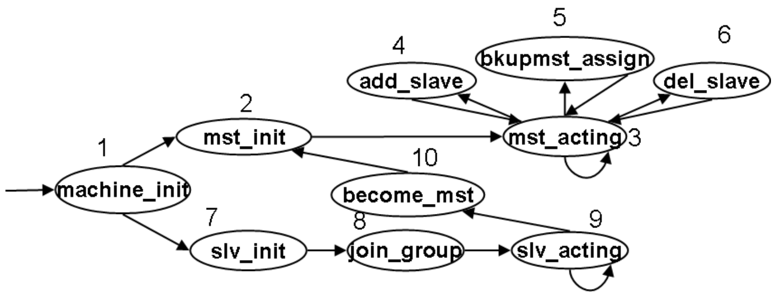


Fig. 5. Overview of the HA protocol model

```
bool alive[N], working[N]; /* omit_failed[N]; byzantine_failed[N]; */
byte mst = NULL, bkupmst= NULL;
chan ch2slv[N]=[1] of {mtype}, ch2mst=[N] of {mtype,byte};

active [N] proctype machine() {
  bool amIbkupmst ;
  byte current=1, next=1;
  alive[_pid]=true;
  ...
end:do
      /* normal behavior */
      :: atomic{ next==1 -> current=1; machine_init();}
      :: atomic{ next==2 -> current=2; mst_init();}
      :: atomic{ next==3 -> current=3; mst_acting();}
      :: atomic{ next==4 -> current=4; add_slave();}
      :: atomic{ next==5 -> current=5; bkupmst_assign();}
      :: atomic{ next==6 -> current=6; del_slave();}
      :: atomic{ next==7 -> current=7; slv_init();}
      :: atomic{ next==8 -> current=8; join_group();}
      :: atomic{ next==9 -> current=9; slv_acting();}
      :: atomic{ next==10-> current=10;become_mst();}

      /* fault occurence at a transition between two states with fault count K */
      :: atomic{next!=NULL && alive[_pid] && (h<K)
           -> next=NULL; current=NULL; h=h+1; failed(); };
      /* recovery */
      :: atomic{ next == NULL -> h=h-1; next=1;};
  od
}
```

Fig. 6. Skeleton of the HA protocol in Promela

More details about the HA model are as follows. The status of machines are defined by two global boolean arrays `alive[N]` and `working[N]` where `N` is a number of machines in the group (the meanings of `alive[i]` and `working[i]` are described in Sect.5.1). `omit_failed[N]` and `byzantine_failed[N]` indicate corresponding faults respectively. A current master and current backup master are indicated by `mst` and `bkupmst` respectively.Each slave has its own communication channel `ch2slv[_pid]` from which it receives messages where `_pid` is a process id of the slave. Also, we have a special channel `ch2mst` that is a dedicated channel to a *current* master. We further abstract networking behaviors of the HA protocol as follows.

- *Abstraction of join process for slaves*
  In the original HA specification, a slave repeatedly broadcasts `join_request` until the slave receives `join_permit` from a master. We simplify this activity by modeling a slave to send a unicast message to `ch2mst` only one time because messages buffered in `ch2mst` are *not* flushed out even when a master crashes. When a master crashes, a new master processes messages in `ch2mst` that were originally sent to the previous master. Thus, slaves do not need to resubmit `join_permit` to a master in case of master's failure.
- *Abstraction of heartbeat health check*
  We have not explicitly modeled heartbeat messages such as `m_alive` and `s_alive`. Neither, we have modeled health checking message such as `query_m_alive`. Instead, a machine polls status of other machines by accessing global variables `alive[i]`, `mst`, and `bkupmst`.
- *Assumption of no packet loss*
  We assume that the HA network does not lose messages. Considering that the HA network is a dedicated network among the machines of the group, this assumption is not unrealistic.

6 Verification of the HA protocol

In this section, we describe the results of verifying the HA protocol with various fault configurations. [4]

6.1 Experimental Result

The HA model consists of around 300 lines of Promela code. We measure the size of state space of the HA model with crash-recovery faults by generating full state space. We start with a model containing a minimal number of machines in a group, which is two. Then, we increase the number of machines until available memory is exhausted. Statistics on the models are illustrated in Table 1. N/A indicates a failure of generating state space due to lack of memory.

We used a computer equipped with PIV 3.0Ghz/2Gb memory/80Gb hdd running spin 4.2.6 on Fedora Linux 4. We set maximum search depth as 5×10^6 and estimated state space as 10^8 where hash table and DFS stack took 697Mb. The results of experiments in the following subsections are for a model of three machines ($N = 3$).

[4] Note that we do not include fail-safe faults in our experiments because these faults are not meaningful for the HA protocol that aims recovery from failures.

Number of machines	2	3	4
States	5835	1.15×10^6	N/A
Transitions	16361	4.57×10^6	N/A
Memory usage(in Mb)	698	771	N/A

Table 1. Statistics on the HA protocol models

6.2 Verification of the Single Master Property P_{SM}

We verified the single master property on the HA model by adding a global counter variable `num_mst`. We modified the HA model to increase `num_mst` by one at `mst_init` state and decrease `num_mst` by one when a master crashes. The verification results are shown in Table 2 where O indicates that P_{SM} is satisfied in a given configuration; X otherwise. The first column of Table 2 shows a number of crash-recovery fault count. The second column indicates verification results with different crash fault counts. Third and fourth columns represent combined fault configurations of crash faults with omission or Byzantine faults if fault count is greater than 0.

The results in the second column show that P_{SM} is satisfied in spite of machine crashes. Also, the third and fourth column indicate that message omission and Byzantine faults do not violate P_{SM}. Note that our model abstracts out several messages for the sake of an minimal model using global variables (see Sect. 5.2) and there exist only three types of messages - `join_request`, `join_permit`, and `bkup_m_assign`. In this situation, message omission may obstruct progress of the HA protocol, but does not disturb its operations; similarly Byzantine faults do not violate P_{SM}. As shown in Table 2, the HA protocol guarantees that there exists at most one master in the presence of all fault combinations.

Fault count	Crash fault	Omission fault	Byzantine fault
0	O	O	O
1	O	O	O
2	O	O	O
3	O	O	O

Table 2. Verification results for P_{SM}

6.3 Verification of the Deadlock-free Property P_{DL}

The verification results of P_{DL} are described in Table 3. The second column shows that if a machine may crash, deadlock can happen. Also, omission fault or Byzantine fault can cause deadlock as shown in the second row of Table 3. Thus, all combinations of crash faults greater than 0 with omission or Byzantine faults can cause deadlock consequently. [5] It is clear that omission or Byzantine fault can cause deadlock because omission/corruption of `join_req` prevents a master from adding a slave to the group. For process crashes, however, a source of deadlock is not clear because the HA protocol has a recovery mechanism from machine crashes. Thus, we tried to identify the causes of deadlock in the presence of crashes.

Fault count	Crash fault	Omission fault	Byzantine fault
0	O	O	O
1	X	X	X
2	X	X	X
3	X	X	X

Table 3. Verification results for P_{DL}

Immediate Cause of the Deadlock. We analyzed all counter examples generated due to deadlock by checking at which states a machine stuck by looking at `machine[`i`]:current` for machine i. The result showed that all machines stuck at `join_group` state. This means that all machines became slaves - no master existed to admit slaves to join the group. In this situation, no progress could be made unless machine 0 should crash and revive as a master, which is clearly beyond the control of the HA protocol.

Therefore, we could conclude that the immediate cause of the deadlock was problem in master assignment process. A remedy for this problem can be to adopt a distributed master election process [10] so that a new master can be elected dynamically even when all machines become slaves.

Identification of Protocol Flaws. By analyzing all counter examples using the McBugger Framework[9], we found the following scenario that caused all machines to become slaves.

> **Flaw A:** *A master died immediately after a backup master (machine 0) had died and revived as a slave. Then, a master revived as a slave and all machines became slaves.*

Note that this flaw was caused by a specific sequence of machine crashes, which was hard to notice because it would not cause deadlock if a backup master

[5] Note that deadlock in our model represents a situation that a group of machines does not make progress in group management activities.

was machine 1 or machine 2. Furthermore, this sequence of machine failures is beyond control of the HA protocol. Thus, to fix this problem, we need major redesign of the HA protocol. Similarly, we could detect Flaw B as follows.

> **Flaw B:** *A backup master died immediately after a master had died and revived as a slave. Then, the backup master revived as a slave and all machines became slaves.*

Similarly to Flaw A, Flaw B is not under the control of the HA protocol. In addition, we detected another flaw as follows.

> **Flaw C:** *A master elected a machine that was dead, as a backup master without knowing that the machine was dead. Then, the master died and it happened that there existed no master.*

Flaw C occurs because a master assigns a slave as a backup master by just sending `bkup_m_assign` to the slave without requiring acknowledgment from the slave. Flaw C can be fixed by adding `ack_bkup_m_assign` message to the HA protocol.

6.4 Verification of the Property P_{req}

We verified P_{req} on the HA model and found that this property was violated in the presence of a crash fault as described in Table 4. Also, omission fault and Byzantine fault caused violation of P_{req}. We found, however, that the cause of violating P_{req} was deadlock because lengths of acceptance cycles in all counter examples were zero. Thus, the analysis result of counter examples due to deadlock (see Sect. 6.3) can be applied for P_{req}.

Fault count	Crash fault	Omission fault	Byzantine fault
0	O	O	O
1	X	X	X
2	X	X	X
3	X	X	X

Table 4. Verification results for P_{req}

7 Conclusion

We have shown our experience of formally modeling and verifying the HA protocol to improve reliability of a group of network security appliances. We devised several abstraction techniques to make the HA model of modest size and applied automated exhaustive state space exploration by model checking to find several flaws of the HA protocol.

Through the project, we realized difficulty of designing a distributed protocol correctly, even a seemingly simple protocol such as the HA protocol. We could detect bugs in the HA protocol through model checking. These bugs had not been noticed before because the original designers of the HA protocol had not been able to consider all possible communication scenarios. The next version of the security appliance adopted a distributed master election process as we suggested in Sect. 6.3.

In addition, we found that a fault model is equally important as a target system model. Considering that environments take important roles in ubiquitous computing systems, a framework of modeling environments with proper fault classes is an important research topic. We will extend our fault modeling framework to allow independent modeling of a target system and its environment.

Acknowledgement

This work was supported by Grant-in-Aid from the Ministry of Education, Culture, Sports, Science and Technology of Japan (No. 18049055).

References

1. Netscreen 5000 series. http://www.juniper.net/products/integrated/ns_5000.html.
2. The Spin Model Checker Home Page. http://www.spinroot.com.
3. Prosenjit Bose, Pat Morin, Ivan Stojmenovic, and Jorge Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. *Wireless Networks*, 7(6):609–616, 2001.
4. C.Bernardeschi, A.Fantechi, and S.Gnesi. Model checking fault tolerant systems. *Software Testing, Verification and Reliability*, 12(4):251–275, 2002.
5. Hubert Garavel and Laurent Mounier. Specification and verification of various distributed leader election algorithms for unidirectional ring networks. *Science of Computer Programming*, 29(1-2):171–197, 1997.
6. G. J. Holzmann. *The Spin Model Checker*. Wiley, New York, 2003.
7. I.Zakiuddin, M.Goldsmith, O. Whittaker, and P.Gardiner. A methodology for model-checking ad-hoc networks. In *SPIN Workshop*, Portland, Oregon, USA, May 2003.
8. K.Bhargavan, C.A.Gunter, M. Kim, I.Lee, D.Obradovic, O.Sokolsky, and M.Viswanathan. Verisim: Formal Analysis of Network Simulations. *IEEE Transaction on Software Engineering*, 8(2), 2002.
9. M. Kim. McBugger - a Monitoring and Checking(MaC) based debugger for model checkers. Submited to Formal Aspects of Testing and Runtime Verification 2006.
10. N.A.Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1997.
11. T.Yokogawa, T. Tsuchiya, and T.Kikuno. Automatic verification of fault tolerance using model checking. In *Pacific Rim International Symposium on Dependable Computing*, 2001.
12. Nitin H. Vaidya, Paramvir Bahl, and Seema Gupta. Distributed fair scheduling in a wireless LAN. In *Mobile Computing and Networking*, pages 167–178, 2000.

13. W.Steiner, J.Rushby, M.Sorea, and H.Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *Proceedings of Intl. Conference on Dependable Systems and Networks*, 2004.
14. Yuan Xue and Baochun Li. A location-aided power-aware routing protocol in mobile ad hoc networks. In *IEEE Global Telecommunications Conference*, pages 2837–2841, 2001.

ネットワークセキュリティ機器プロトコルのモデル化と形式検証
(in English)
(算譜科学研究速報)
発行日：2006 年 7 月 31 日
編集・発行：独立行政法人産業技術総合研究所関西センター尼崎事業所
　　　　　　システム検証研究センター
同連絡先：〒661-0974 兵庫県尼崎市若王寺 3-11-46
e-mail：informatics-inquiry@m.aist.go.jp
本掲載記事の無断転載を禁じます

Formal Modeling and Verification of Management
on a Group of Network Security Appliances
(Programming Science Technical Report)
July 31, 2006
Research Center for Verification and Semantics (CVS)
AIST Kansai, Amagasaki Site
National Institute of Advanced Industrial Science and Technology (AIST)
3-11-46 Nakouji, Amagasaki, Hyogo, 661-0974, Japan
e-mail: informatics- inquiry@m.aist.go.jp
・Reproduction in whole or in part without written permission is prohibited.