# Formal Modeling and Verification of High-Availability Protocol for Network Security Appliances

Moonzoo Kim

CS Dept. Korea Advanced Institute of Science and Technology
Daejeon, South Korea
moonzoo@cs.kaist.ac.kr

**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 and managed via *High-Availability (HA)* protocol.

In this paper, we present our experience of formally modeling and verifying the HA protocol used for commercial network security appliances through model checking. In addition, we applied a new debugging technique to detect *multiple* bugs *without* modifying/fixing the HA model by analyzing all counter examples. Throughout these formal analysis, we could effectively detect several design flaws.

## 1 Introduction

As more computing systems are deployed in wide functions of our society such as mobile banking, tele-conferencing, and online stock trading systems, communication between remote systems becomes essential for ubiquitous computing society. Internet provides such communication services for a large number of applications, but at the cost of *security* and *reliability*. Therefore, more and more security appliances such as firewall, VPN, and IDS/IPS are deployed in small to giant size networks. As security appliances become key components of network, their reliability, not only as security guardians, but also as network components, becomes a critical issue. For high-traffic networks, it is convention to deploy multiple security appliances grouped together for both fault tolerance and high network throughput. Most high-end security appliances achieve these two goals via *High-Availability (HA) protocol* among the appliances.

Despite the importance of HA protocol, HA protocol often causes failures to network security appliances for the following reasons. First, HA protocol, which is a fault-tolerant distributed network protocol, is notorious for its high complexity. It is a challenging task to consider all possible communication/coordination scenarios among the network security appliances in a group. Furthermore, failure and recovery of each machine in the group should also be considered, which

K.S. Namjoshi et al. (Eds.): ATVA 2007, LNCS 4762, pp. 489–500, 2007.

adds complexity further. Second, testing high-end network security appliances requires great efforts due to complex network/machine configurations and a large number of test scenarios. Also, it is hard to determine whether misbehavior is due to errors of the HA protocol or due to other factors such as OS/HW failure and/or misconfiguration of networks, etc. Finally, manufacturers often concentrate on developing functions of each security appliance without considering how these machines should communicate each other through HA protocol. Thus, HA protocol is often designed and implemented in an ad-hoc way at the last stage of development, which often creates unexpected behaviors. As a result, it is often observed that a group of network security appliances exhibits abnormal behaviors such as decreased network throughput or dropped normal packets while a single security appliance works well without a problem. Therefore, it is highly desirable to *formally* model and verify HA protocol design as formal method techniques have been actively applied to enhance reliability of network applications [7,10,11,9].

This paper presents our experience of formally modeling and verifying the HA protocol implemented in a commercial network security appliance NXG2000 [1]. We built a HA protocol model of a moderate size and verified the model using the Spin model checker [2] to check the absence of deadlock in the HA protocol. In this project, we could overcome the limitation of traditional debugging by detecting multiple bugs from the all counter examples without modifying/fixing the HA model as described in Sect. 4.

## 2 Overview of the HA Protocol of NXG2000

NXG2000 [1] is an integrated network security appliance consisting of firewall, VPN, and IDS targeted for gigabit networks. NXG2000 provides upto 1 million concurrent sessions (maximum 2 Gbps throughput) via six gigabit ports. In addition, NXG2000 has a 100 Mbps HA port dedicated to the HA protocol.

Network equipments located at the gateway must achieve high reliability as well as fast recovery lest the whole internal network cannot be operational. Thus, multiple network security appliances are deployed in a group for both fault tolerance and increased network throughput. For this purpose, machines of a group cooperate with each other to perform several tasks such as session synchronization and group management through the HA protocol.

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. Although a master performs various jobs such as synchronizing sessions, configuring network, and creating event logs, we focus on the core management tasks of a master as follows.

1. *Addition of slaves* (see Fig. 1.a))
   When a slave becomes operational, the slave broadcasts `join_request` messages every second until it receives a `join_permit` message from a master.

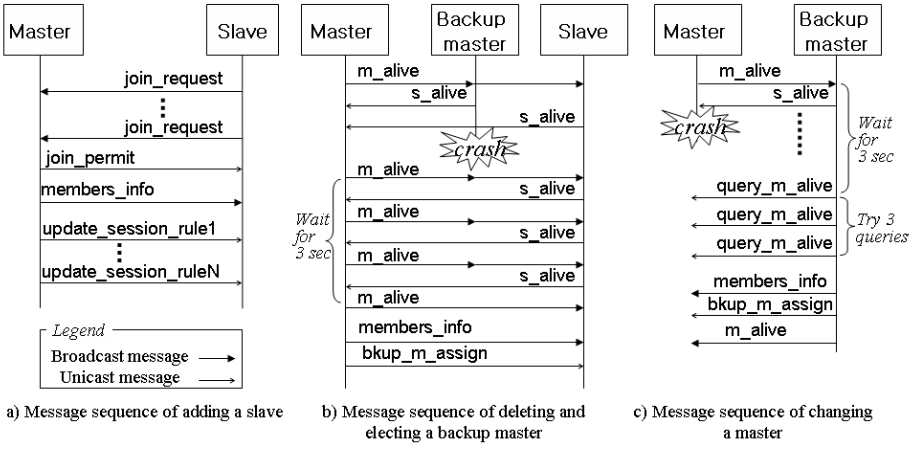


**Fig. 1.** Message sequences regarding the HA activities

Once the master allows the slave to join the group by sending a `join_permit` message to the slave, the master broadcasts new information to the group. Following this, the master sends all session information to the slave.

2. *Deletion of slaves* (see Fig. 1.b))
   A master constantly checks the status of slaves by receiving `s_alive` from every slave each second. If a master does not receive `s_alive` from a slave for three seconds, the master erases the slave from the group. If the erased slave is a backup master, the master elects another slave as a backup master and sends `bkup_m_assign` to the slave.
3. *Assignment of a backup master* (see Fig. 1.b))
   To prepare for a case in which a master crashes, the master assigns a slave as a backup master that will become 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.

A backup master constantly checks whether or not a master is operational by receiving `m_alive`, which is broadcasted by a master every second (see Fig. 1.c)). If a backup master does not receive `m_alive` for three seconds, the backup master sends `query_m_alive` three times to the master. If the backup master does not receive a response from the master, the backup master becomes a master and broadcasts its new status. The backup master then assigns another slave as a new backup master by sending `bkup_m_assign` and starts broadcasting `m_alive` messages. A security appliance starts working 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, will work as a master if there is no master when it recovers from a failure.

## 3 The HA protocol Model

We model the HA protocol in Promela [2]. Each machine, regardless of whether it is a master or a slave, is modeled as a process. The overall execution of each machine is depicted in Fig. 2.

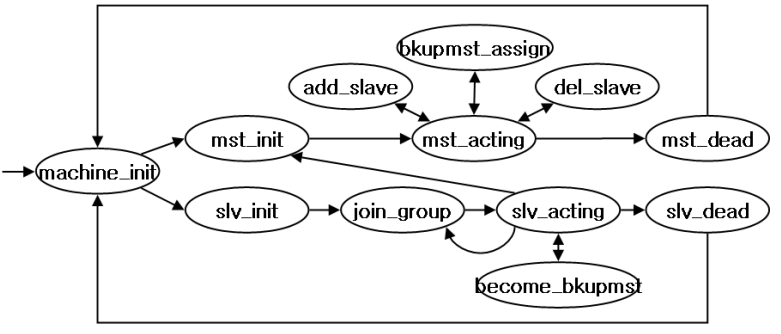


**Fig. 2.** Overview of the HA protocol model

Each machine starts from `machine_init` state (located at the left end of Fig. 2). Initially, machine 0 (whose process id is 0) is statically designated as a master and the machine moves to `mst_init` state to become a master. Then, the machine is working at `mst_acting` state 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)
- To assign a slave as a backup master (`bkupmst_assign` state)
- To delete a slave from the group if the slave is found dead (`del_slave` state)
- To exhibit a crash (`mst_dead` state)

Note that there exists only one crash point for master in this model; a master can fail/crash only at `mst_dead` state. Thus, this model does not exhibit failure while a master is adding/deleting a slave or assigning a slave as a backup master. This simplified failure model abstracts out the need of cleanup procedures in a case of failure, which reduces complexity of the HA model significantly.

Once a machine is determined as a slave, the machine moves to `slv_init` state to initialize settings to become a slave. Then, the slave moves to `join_group` state 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 performing the following tasks.

- To become a backup master (`become_bkupmst` state)
- To become a master if it is a backup master and there exists no master (a transition to `mst_init` state)
- To exhibit a crash (`slv_dead` state)

## 4 A New Debugging Technique to Detect Multiple Bugs

Model checking techniques are effectively used as a means to improve the reliability of computing systems by detecting bugs of formal system models [8,15]. The traditional way of debugging a formal model is as follows: First, a human engineer identifies a bug in a counter example. The bug is then fixed by modifying the model. Once the bug is fixed, the modified model is verified again in order to detect the next bug, if one exists, in a new counter example. This debugging process is repeated until there no more bugs are found. This approach toward debugging has the following limitations:

- There are cases where it is not feasible to fix a bug for several reasons. In such cases, no further debugging progress can be made.
- Fixing a bug may introduce other bugs so that the traditional debugging iterations may continue indefinitely, or never terminate in the worst cases.
- When a model is modified to fix one bug, all requirement properties must be verified once again. Considering that real-world applications often have several hundred properties to check, these repeated fix-and-verify trials consume considerable project time.

Therefore, we propose a new debugging technique to identify multiple bugs *without* modification of a model by analyzing all counter examples generated by model checker. There have been researches on analysis of counter examples with various goals such as model refinement and localization of bugs [6,16,3,4]. Our focus, which is orthogonal to these related works, is to provide an automated process to detect as many bugs as possible by analyzing multiple counter examples without modification of a model.

### 4.1 An Automated Process to Detect Multiple Bugs

First, we describe an automated process that detects multiple bugs that violate the requirement property $\phi$ without modification to the target model. A key point of the process is to construct a set of formulas $\psi_i$'s each of which captures/describes a bug $b_i$ revealed in a subset of counter example traces.[1] Following this, the traces that satisfy/conform to $\psi_i$'s are automatically detected, i.e., those that violate $\phi$ due to $\psi_i$. For this automatic trace analysis, it is necessary to formally specify $\psi_i$'s in a formal specification language such as Meta Event Definition Language (MEDL)(see Sect. 4.3). Notations are defined before describing this debugging process formally.

- $T_\phi$ is the set of all counter example traces of a requirement property $\phi$ such that $T_\phi = \{t_i |\ t_i$ is a counter example of $\phi\}$.
- $B_\phi$ is a set of formulas of the bugs that violate $\phi$, i.e., $B_\phi = \{\psi_i |\ \psi_i$ is a formula of a bug that violates $\phi\}$.

[1] A bug $b_i$ is identified through manual analysis as in the traditional debugging.

- $t \models_\phi \psi$ where $t \in T_\phi$ and $\psi \in B_\phi$ signifies that a counter example trace $t$ satisfies $\psi$ that is a cause of the violation of $\phi$ (i.e., $t$ violates $\phi$ due to $\psi$).
- $t_\phi : B_\phi \rightarrow \mathcal{P}(T_\phi)$ is a function such that $t_\phi(\psi) = \{t_i \in T_\phi |\ t_i \models_\phi \psi\}$.

An algorithm that detects multiple bugs without modifying the target model is described in Fig. 3. This algorithm is guaranteed to terminate if evaluation of a trace at Step 3 is decidable (which is true in most practical cases) as $T_\phi$ is finite in a finite state model. All steps of the algorithm can be automated except Step 2, which still requires human ingenuity to identify a bug and specify the bug as $\psi$ in a formal specification language.

1. Set $T$ with $T_\phi$.
2. Select the smallest trace $t_{init} \in T$. Then a user analyzes $t_{init}$ to identify a bug $b$ that violates $\phi$ and specifies the bug $b$ as $\psi$.
3. Obtain $t_\phi(\psi)$ by checking all traces of $T$ with $\psi$.
4. Set a new set of traces $T'$ with $T - t_\phi(\psi)$ and select the new smallest trace $t'_{init} \in T'$.
5. Set $T$ with $T'$ and $t_{init}$ with $t'_{init}$, then repeat from Step 2 until $T$ becomes $\emptyset$.

**Fig. 3.** An algorithm to detect multiple bugs that violate $\phi$

Fig. 4 illustrates the algorithm. Initially, the smallest trace $t_0$ that violates $\phi$ is manually analyzed and the bug $b_0$ revealed in $t_0$ is described as $\psi_0$. Following this, $t_\phi(\psi_0)$ is obtained, and the smallest trace $t_1 \in (T_\phi - t_\phi(\psi_0))$ is found. This process is repeated until $\psi_0, \psi_1$, and $\psi_2$ that cover $T_\phi$ completely are found (i.e., $t_\phi(\psi_0) \cup t_\phi(\psi_1) \cup t_\phi(\psi_2) = T_\phi$). Note that the algorithm of Fig. 3 does not strictly require a collection of $t_\phi(\psi_i)$ to be pairwise disjoint. It is possible that $t_\phi(\psi_i)$ overlaps $t_\phi(\psi_j)$, which, however, does not affect a result of the algorithm.

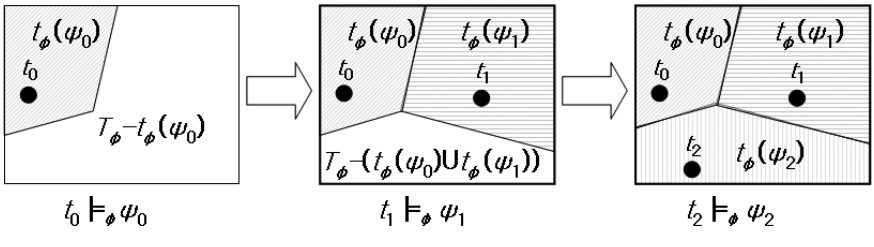


**Fig. 4.** A process of detecting bugs that violate a requirement property $\phi$

### 4.2 Overview of the MacDebugger Framework

The MacDebugger framework [13] (see Fig. 5), which is an extension of the MaC framework [12], is a general framework for analyzing a large volume of counter

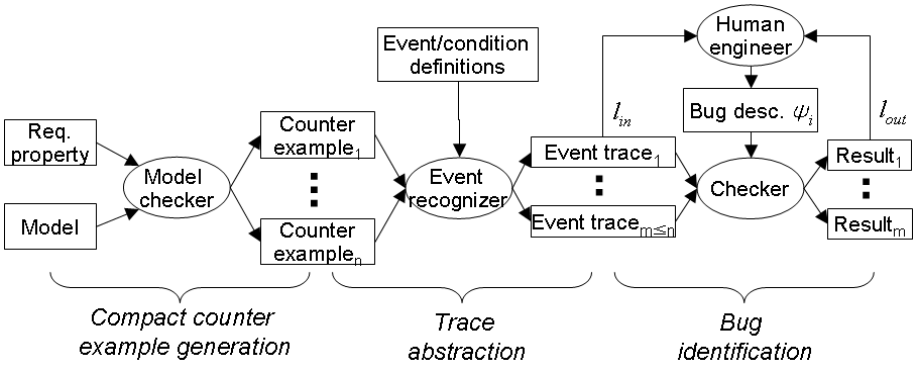


**Fig. 5.** Overview of the MacDebugger framework

example traces. MacDebugger is designed to work with any model checker that can generate multiple counter examples. As a prototype, however, it was implemented to work with the Spin model checker [8]. MacDebugger consists of the following three components - a *model checker*, an *event recognizer*, and a *checker*.

MacDebugger aims to analyze a large number of counter examples efficiently. Thus, performance of storing and analyzing counter examples is a critical issue, as even a simple model can generate hundreds of gigabytes of counter examples. For that purpose, we modified the Spin model checker to generate counter examples in a compact format. An event recognizer (an oval in the middle of Fig. 5) extracts sequences of *primitive events* and *conditions* from counter examples generated from a model checker and generates event traces which contain these sequences. A checker (an oval in the right of Fig. 5) receives a list of the event traces to analyze, for example $l_{in}$, and a bug description written in MEDL, in this example $\psi_i$, as its inputs. The checker analyzes all event traces in $l_{in}$ with respect to $\psi_i$ and returns a list of event traces, in this example $l_{out}$, which do not satisfy $\psi_i$. Following this, a human engineer investigates the shortest event trace in $l_{out}$ and identifies a new bug $\psi_{i+1}$ from the trace. The checker then repeats this debugging process using $l_{out}$ and $\psi_{i+1}$ as new inputs until all event traces/counter examples are covered by $\psi_0 ... \psi_n$ as described in Sect. 4.1.

### 4.3 Meta Event Definition Language

MEDL is based on an extension of linear temporal logic with auxiliary variables to record history of the event trace. MEDL distinguishes between two kinds of data that make up the trace of an execution - *events* and *conditions*. Events occur instantaneously during the system execution, whereas conditions represent information that holds for a duration of time [5]. A checker assumes that truth values of all conditions remain unchanged between updates from the event recognizer. For events, a checker makes the dual assumption, namely, that no events (of interest) happen between updates. Based on this distinction between

**Table 1.** The syntax of conditions, events, and guards

| |
|---|
| E ::= $e$ \| `start`(C) \| `end`(C) \| E&&E \| E\|\|E \| E `when` C |
| C ::= $c$ \| `defined`(C) \| [E,E) \| !C \| C&&C \| C\|\|C \| C⇒C |
| G ::= E → {*statements*} |

events and conditions, we have a simple two-sorted logic that constitutes MEDL. The syntax of events (E), conditions (C), and guards (G) is given in Table 1.

Here $e$ refers to primitive events that are reported in the trace by the event recognizer; $c$ is either a primitive condition reported in the trace or it is a boolean condition defined on the auxiliary variables. Guards (G) are used to update auxiliary variables. The semantics for boolean operations over conditions and events is defined naturally. There are some natural events associated with conditions, namely, the instant when the condition becomes *true* (`start`($c$)), and the instant when the condition becomes $false$ (`end`($c$)). Also, any pair of events define an interval of time, so forms a condition $[e_1, e_2)$ that is *true* from event $e_1$ until event $e_2$. The event ($e$ `when` $c$) is present if $e$ occurs at a time when condition $c$ is *true*. Finally, a guard $e \rightarrow \{statements\}$ updates auxiliary variables according to the assignments given in *statements* when $e$ happens.

A MEDL script defines a requirement property as a special event, called *alarm*. To check whether an alarm occurs or not, a checker evaluates the events and conditions defined in the script whenever it reads an element from the trace. For more detail on the formal semantics of MEDL, see [12].

## 5 Verification of the HA protocol

The full state space of the model is generated without stopping at violations. Statistics on the model with a different number of machines in a group are illustrated in Table 2. N/A indicates that the state space failed to be generated due to a lack of memory. A Pentium IV 3Ghz computer equipped with 2 GB of memory, and 80GB of hard disk running Spin 4.2.6 on Fedora Linux 4 was used. A maximum search depth was set as $5 \times 10^6$ and the estimated state space as $10^8$, of which the hash table and DFS stack took 227 Mb. The HA protocol model in Promela is approximately 200 lines long. We found that all HA models with $N \geq 2$ had deadlock and all counter examples causing deadlock were generated. Table 3 shows the statistics on the counter examples.

**Immediate Cause of the Deadlock.** Firstly, an immediate cause of the deadlock at $N = 2$ was identified. The shortest counter example was analyzed and it was found that deadlock occurred when all machines in the group were slaves, in other words when no master existed to admit slaves to join the group. In this situation, no progress could be made unless machine 0 crashed and revived as a master, which is clearly beyond the control of the HA protocol. Fig. 6 shows this fault that immediately causes deadlock formulated in MEDL. `deadlock` in line 2

**Table 2.** Statistics on the HA protocol model with a different number of machines

| Number of machines in a group ($N$) | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|
| States | 246 | 17489 | 551052 | $1.40 \times 10^7$ | N/A |
| Transitions | 409 | 43419 | $1.75 \times 10^6$ | $5.24 \times 10^7$ | N/A |
| Memory usage(in Mb) | 228 | 229 | 264 | 1321 | N/A |
| Time to generate state space (in sec) | 1.0 | 1.1 | 3.2 | 86.9 | N/A |

**Table 3.** Statistics on the counter examples showing deadlock

| Number of machines ($N$) | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| # of counter examples | 4 | 156 | 4440 | 123360 |
| Size of total counter examples (in bytes) | 0.3K | 628K | 53M | 36G |
| Avg. length of counter example (in steps) | 36 | 1271 | $2.8 \times 10^4$ | $8 \times 10^5$ |
| Time to generate all counter examples | 0.1 sec | 0.3 sec | 65 sec | 11 hour |

is a primitive event representing deadlock. The event recognizer recognizes this event by detecting the end of a counter example. `m0_slave` in line 3 is a primitive condition indicating whether or not machine 0 is a slave. `m1_slave` is similarly defined. Thus, if deadlock occurs when all machines are slaves, the `all_slaves` alarm in line 4 is triggered.

```
01:ReqSpec DeadlockDetector
02:    import event deadlock;
03:    import condition m0_slave, m1_slave;
04:    alarm all_slaves = deadlock when (m0_slave && m1_slave);
05:End
```

**Fig. 6.** MEDL specification of the fault causing deadlock ($N = 2$)

All counter examples of the models were checked with $N \geq 2$. It was found that all traces raised the `all_slave` alarm, indicating that the immediate cause of the deadlock was incorrect master election process.

**Identification of Design Flaws.** First, the shortest counter example in the smallest model ($N = 2$) was analyzed further and we found the following faulty scenario.

*$f_1$: A master (machine 1) died immediately after a backup master (machine 0) had died and revived as a slave. Machine 1 then revived as a slave and all machines became slaves.*

$f_1$ was formulated as `f1` in line 6 of Fig. 7. `mst_died` and `bkupmst_died` indicate crashes of the corresponding machines. `becomes_mst` occurs when a backup master becomes a master. `bkupmst_elected` indicates that a new backup master is elected, and `m0_alive` indicates that machine 0 is alive. `m0_working` indicates that machine 0 has joined the group and is cooperating with the other machines in the group. `f1` is triggered when a master dies without a backup master (line 6) with additional conditions for machine 0 (line 7) satisfied.

It is important to note that a bug triggering $f_1$ is hard to fix because fixing the HA protocol to work correctly with this scenario requires major redesign of the HA protocol, which was not feasible due to limited project resources. Thus, other bugs could not be detected if we used the traditional debugging method. We could, however, continue debugging process to detect other remaining bugs as explained below by using the new debugging technique.

```
01:ReqSpec f1Detector
02:  import event mst_died, bkupmst_died, becomes_mst, bkupmst_elected;
03:  import condition m0_working, m0_alive;
04:
05:  condition restriction = !m0_working && m0_alive;
06:  alarm f1 =mst_died when ([bkupmst_died||becomes_mst,bkupmst_elected)
07:        && value(mst_died,0) != 0 && restriction );
08:end
```

**Fig. 7.** Specification of $f_1$ in MEDL

It was found that 4 out of 4 ($N = 2$), 90 out of 156 ($N = 3$), 2703 out of 4440 ($N = 4$), as well as 70042 out of 123360 counter examples ($N = 5$) raised the `f1` alarm (see Table 4). This indicates that other faults exist, as $f_1$ does not cover all counter examples. The smallest counter example trace that did not raise the alarm in $N = 3$ was analyzed, and the following faulty scenario in the trace was found.

> *$f_2$: A master elected a machine that was dead, as a backup master without recognizing that the machine was dead. The master then died and it happened that there existed no master.*

This fault is caused by a bug in which a master assigns a slave as a backup master by only sending `bkup_m_assign` to the slave and not requiring acknowledgment from the slave. It was found that 62 ($N = 3$), 1560 ($N = 4$) and 51200 counter examples ($N = 5$) raised $f_2$ (see Table 4). $f_1$ and $f_2$, however, still do not cover all counter examples, indicating that there still are other faults.

In a similar manner, $f_3$ was detected and formulated to specify that a backup master died immediately after a master has died, making all machines slaves. Finally, $f_1, f_2$, and $f_3$ covered all counter examples, indicating that all of the bugs that cause deadlock had been founded. These analysis results are shown in

**Table 4.** Analysis results of counter examples due to $f_1$, $f_2$, and $f_3$

| Number of machines (N) | 2 | 3 | 4 | 5 |
|---|---|---|---|---|
| Total # of counter examples | 4 | 156 | 4440 | 123360 |
| # of event traces due to $f_1$ | 4 | 90 | 2703 | 70042 |
| # of event traces due to $f_2$ | 0 | 62 | 1560 | 51200 |
| # of event traces due to $f_3$ | 0 | 4 | 177 | 2118 |

Table 4. It takes less than one minute to analyze all counter examples for $N \leq 4$ and takes around 7 hours to analyze all counter examples to check each of $f_1$, $f_2$, and $f_3$ for $N = 5$.

## 6 Conclusion

In this paper, we present results of formal modeling and verification of the HA protocol of NXG2000. In this study, we could find several bugs in the HA protocol through analyzing counter examples generated by model checking. These bugs had not been noticed by the company before, and the company decided to adopt a distributed master election process [14] in the next version of NXG2000. We are convinced that the new debugging technique in this paper is effective to verify systems of industrial strength, which often have hard-to-fix bugs. We plan to develop this debugging technique further by adopting other works on counter example analysis and apply the technique to formally analyze more industrial systems.

As a future study, we plan to work to add backward analysis capability to MacDebugger. It was noticed that a root cause of the violation of a safety property most often exists at the end of a counter example. Thus, if it is possible to analyze a counter example backward (from the end to the start of the counter example), this may decrease the analysis time significantly. In addition, we will formulate identified bugs using Promela `never` claim and run model checker on the original model with the bug descriptions to know if there still exist unrevealed bugs or not. This approach eliminates the overhead of analyzing a large volume of counter examples at the cost of increased model size. The comparison between these two different approaches on analysis performance and convenience of formulating bugs can be an interesting research topic.

## References

1. High-availability technique in NXG 2000. Technical report, `http://www.secui.com/product/nxg/pdf/NXG_technique_03.pdf`
2. The Spin Model Checker Home Page, `http://www.spinroot.com`
3. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–136. Springer, Heidelberg (2003)

4. Basu, S., Saha, D., Smolka, S.A.: Localizing programs errors for cimple debugging. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 79–96. Springer, Heidelberg (2004)
5. Heitmeyer, C., Bull, A., Gasarch, C., Labaw, B.: Scr*: A toolset for specifying and analyzing requirements. In: Haveraaen, M., Dahl, O.-J., Owe, O. (eds.) COMPASS 1995. LNCS, vol. 1130, Springer, Heidelberg (1996)
6. Pasareanu, C.S., Dwyer, M.B., Visser, W.: Finding feasible counter-examples when model checking java programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 284–298. Springer, Heidelberg (2001)
7. Holzmann, G.J., Smith, M.H.: Automating software feature verification. Bell Labs Technical Journal 5(2), 72–87 (2000)
8. Holzmann, G.J.: The Spin Model Checker. Wiley, New York (2003)
9. Zakiuddin, I., Goldsmith, M., Whittaker, O., Gardiner, P.: A methodology for model-checking ad-hoc networks. In: Ball, T., Rajamani, S.K. (eds.) SPIN Workshop. LNCS, vol. 2648, Springer, Heidelberg (2003)
10. Bhargavan, K., Gunter, C.A., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., Viswanathan, M.: Verisim: Formal Analysis of Network Simulations. IEEE Transaction on Software Engineering 8(2) (2002)
11. Bhargavan, K., Obradovic, D., Gunter, C.: Formal verification of standards for distance vector routing protocols. Journal of the ACM 49(4), 538–576 (2002)
12. Kim, M., Viswanathan, M., Kannan, S., Lee, I., Sokolsky, O.: Java-mac: A run-time assurance approach for java programs. Formal Methods in System Design (2004)
13. Kim, M.: MacDebugger: A Monitoring and Checking (MaC) based Debugger for Formal Models, Technical Report CS-TR-2007-270, CS Dept. KAIST (2007)
14. Lynch, N.A.: Distributed Algorithms. Morgan Kaufmann, San Francisco (1997)
15. Ruys, T.C., Holzmann, G.J.: Advanced spin tutorial. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 304–305. Springer, Heidelberg (2004)
16. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. Principles of Programming Languages (2003)