Applying Model Checking to Destructive Testing and Analysis of Software System

Hiroki Kumamoto, Takahisa Mizuno, Kensuke Narita, Shin-ya Nishizaki
Department of Computer Science, Tokyo Institute of Technology, Tokyo, Japan.
Email: nisizaki@cs.titech.ac.jp, {hiroki.kumamoto, takahisa.mizuno, kensuke.narita}@lambda.cs.titech.ac.jp

Abstract—**Recently, model checking is widely applied to software and hardware verification. It can locate hard-to-find bugs in systems by exhaustively searching executing paths. In this paper, we propose a new software design method that enables us to evaluate the fault tolerance of software behavior at the specification level: we can check software behavior, not only when the hardware and network are in good order, but also when they are out of order; we can then improve fault tolerance of the target software using the model checker. We can test software under environments in which we destroy hardware and/or networks intentionally in computer simulation. The method is explained by taking an example of a network-connected AV appliance. We model the AV appliance by the modeling language Promela and analyze it by the SPIN model checker.**

Index Terms—**model checking, software verification, fault tolerance, SPIN model checker, Promela**

I. Introduction

In this section, we start with some backgrounds and then explain the purpose of our research.

A. Backgrounds

In software engineering, model checking is regarded as a genuine breakthrough, especially in regard to the improvement of software design and coding. Model check-ing is a technique for verifying whether a model satisfies a given specification. Models are extracted from descriptions presented as state-transition diagrams or in concurrent programming languages. The specifications are often represented by temporal logic formulae. A number of model checkers have been developed, including the SPIN model checker [1] and Uppall2k [2].

Although the models to be verified are formulated as automata in many automata, SPIN model checkers enables us to write in Promela, which is a concurrent programming language with message passing, non-deterministic choice, and parallelism. In Promela, the case selection is described as

```
    if
    :: guard1 -> option1;
    :: guard2 -> option2;
    :: else -> else_option;
     fallthrough_option;
    fi
```

If both `guard1` and `guard2` are satisfied, either `option1` or `option2` is executed non-deterministically. If neither `guard1` nor `guard2` is satisfied, `else_option` is executed. In all cases, `fallthrough_option` is executed.

```
    do
    :: guard1 -> option1;
    :: guard2 -> option2;
    :: else -> break;
    od
```

In this example, either option1 or option2 is selected as well as the case selection above-mentioned and then the loop repeats itself. Neither guard1 nor guard2 is satisfied, then the loop finished.

The following is a typical description of process' definition.

```
  active proctype
    process1(chan ch1;ch2){
      statement1;
      statement2;
      statement3;
    }
```

This code fragment means that an instance of a process in which *statement1*, *statement2*, and *statement3* are executed in sequel is generated, initialized, and activated.

Data structures in Promela are very limited because codes in Promela are translated into automata. A property to be verified is described in linear temporal logic (LTL) formulas [3].

The LTL formulas consist of

- propositional variables,
- logical operators such as $\neg\varphi, (\varphi \vee \psi), (\varphi \wedge \psi), (\varphi \rightarrow \psi)$, and
- temporal modal operators such as $\diamond$ ("in the future"), and $\square$("globally").

A formula $\diamond\varphi$ means that eventually φ becomes true; $\square\varphi$ means that φ always remains true. There are two typical kinds of properties which can be described in the linear temporal logic:

- *safety* properties state that something bad never happens, $\square\neg\varphi$;

This paper is based on "Destructive Testing of Software Systems by Model Checking" by H. Kumamoto, T. Mizuno, K. Narita, S. Nishizaki which appeared in the Proceedings of the 10th International Symposium on Communication and Information Technologies, Tokyo, Japan, October, 2010, IEEE. This work was supported by Grant-in-Aid for Scientific Research (C), (24500009).

- *liveness* properties state that something good keeps happening, ($\psi \rightarrow \diamond \varphi$).

An LTL formula is also translated into an automaton. A pair of two automata, one from a Promela code and one from an LTL formula, is simultaneously executed and checked. Two important features of the model checker related to our work are:

- The model checker exhaustively determined whether a given model satisfies a LTL formula query by tracing all the execution paths;
- The model checker gives us an execution path that is a counterexample of the LTL formula query. We can then improve the model with the results of model checking.

Fault tolerance [4,5] is forbearance that enables a system to continue operating properly in the event of the failure of (or faults within) some of its components, and *fault-tolerant design* a design that enables a system to continue operation, possibly at a reduced level, rather than failing completely, when some part of the system fails. Fault tolerance has been actively studied in operating systems research. In this paper, we discuss fault tolerance with the example of a network-connected AV appliance.

B. Puropose

In this paper, we discuss a method for applying model checking to the analysis of the fault tolerance of network-connected systems. We illustrate the method with an example of an AV appliance, connected in a local area network consisting of a DVD recorder and a hard disk drive (HDD) recorder. We analyze the AV appliance system with the SPIN model checker not only in normal circumstances, but also in hardware problem cases. If fragile points in the system are found, they can be improved with the assistance of the model checker.

II. Formalization for Model Checking

In this section, we first introduce an AV appliance system, which we consider as an example to be analyzed. We then formalize the system in the modeling language Promela of the SPIN model checker.

A. AV Appliance System

A simple audio-visual appliance system the fault tolerance of which can be analyzed is specified as a class diagram [6] in Figure 1.

(1) A video recorder with a hard disk drive ("HDD recorder" in Figure 1) and a video recorder with a DVD drive ("DVD recorder" in Figure 1) are connected by a local area network.
(2) Video content on the HDD recorder or the DVD recorder can be played on an LCD display.
(3) Video content on the HDD recorder can be duplicated on the DVD recorder.
(4) Storage devices like the hard disk drive and the DVD drive should be exclusively allocated.

The second operation (2) is described in detail:

(1) A user pushes the COPY button on the HDD recorder.
(2) The HDD recorder starts playing video content and transmits it to the DVD recorder through a line between them. The DVD recorder writes the received content on DVD media.
(3) When the transmission is finished, the DVD writing is terminated.

The details of this operation have been simplified. For example, to copy video content on the HDD recorder to the DVD recorder, one would have to select the content among several candidates; this is omitted for the sake of clarity.

Next, we will analyze system behavior when problems occur within the system.

B. Modeling for SPIN Model Checker

We model the AV appliance system in Promela, a model description language, which was proposed for writing target models for the SPIN model checker.

The Promela code of the AV appliance system is presented in the Appendix; here, we give a UML model [6] of the AV appliance system to facilitate understanding of the Promela code (Figure 2, 3, 4, and 5).

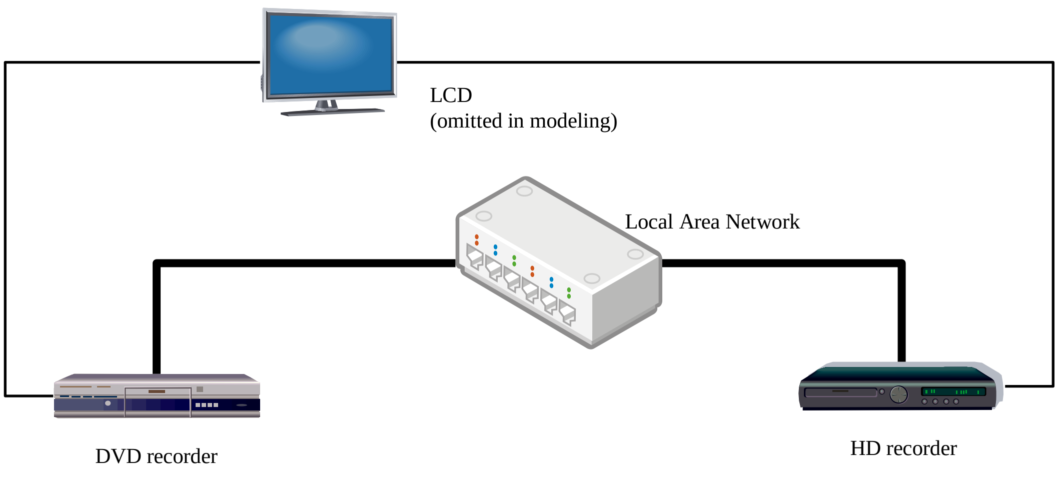


Figure 1. AV Appliance System

© 2013 ACADEMY PUBLISHER

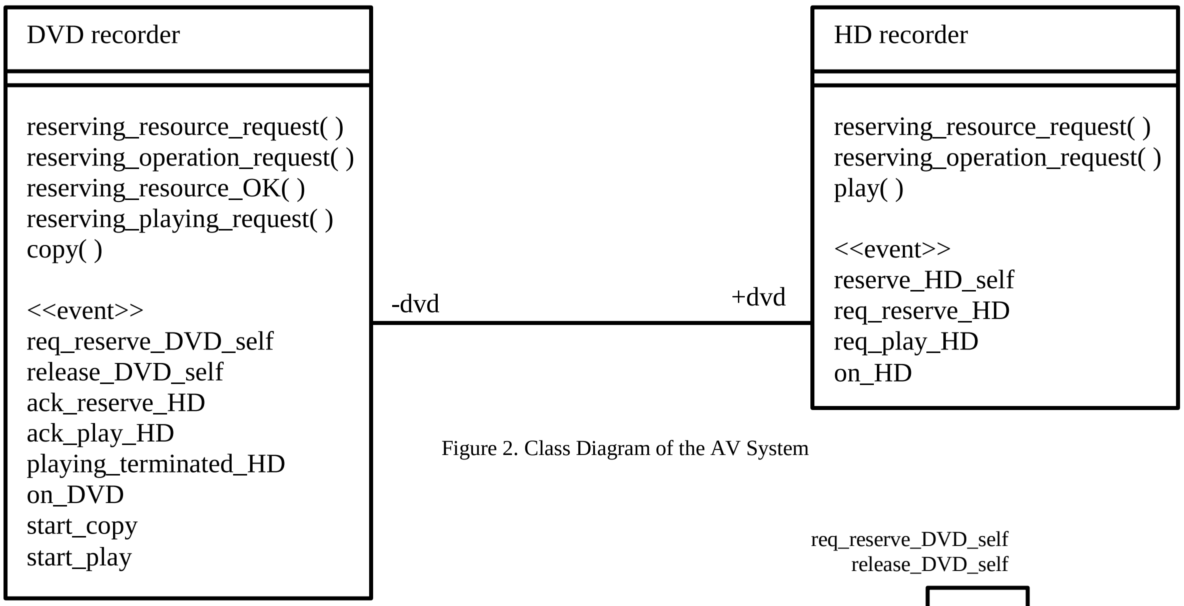


Figure 2. Class Diagram of the AV System

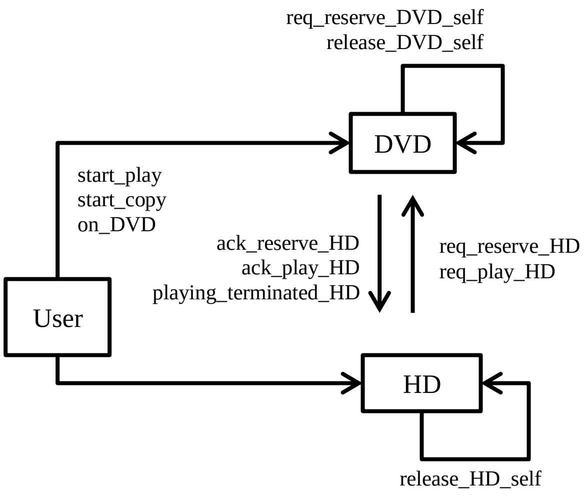


Figure 3. Communication Diagram of the AV System

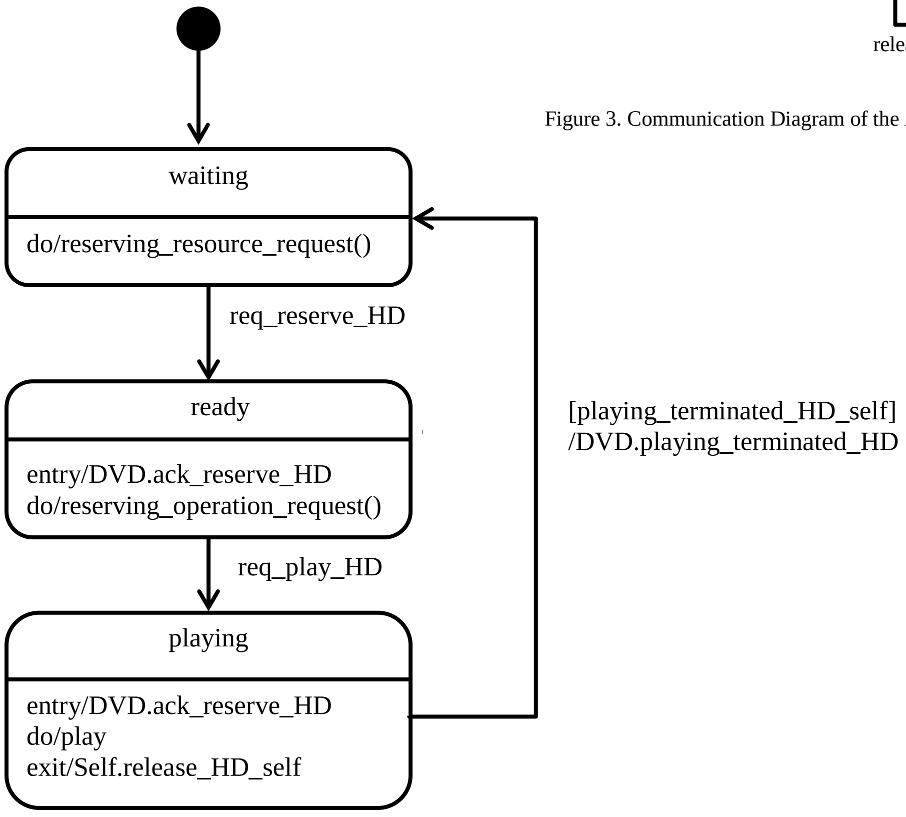

waiting
do/reserving_resource_request()
req_reserve_HD
ready
entry/DVD.ack_reserve_HD
do/reserving_operation_request()
req_play_HD
playing
entry/DVD.ack_reserve_HD
do/play
exit/Self.release_HD_self
[playing_terminated_HD_self]
/DVD.playing_terminated_HD

Figure 4. Statemachine Diagram of HD recorder

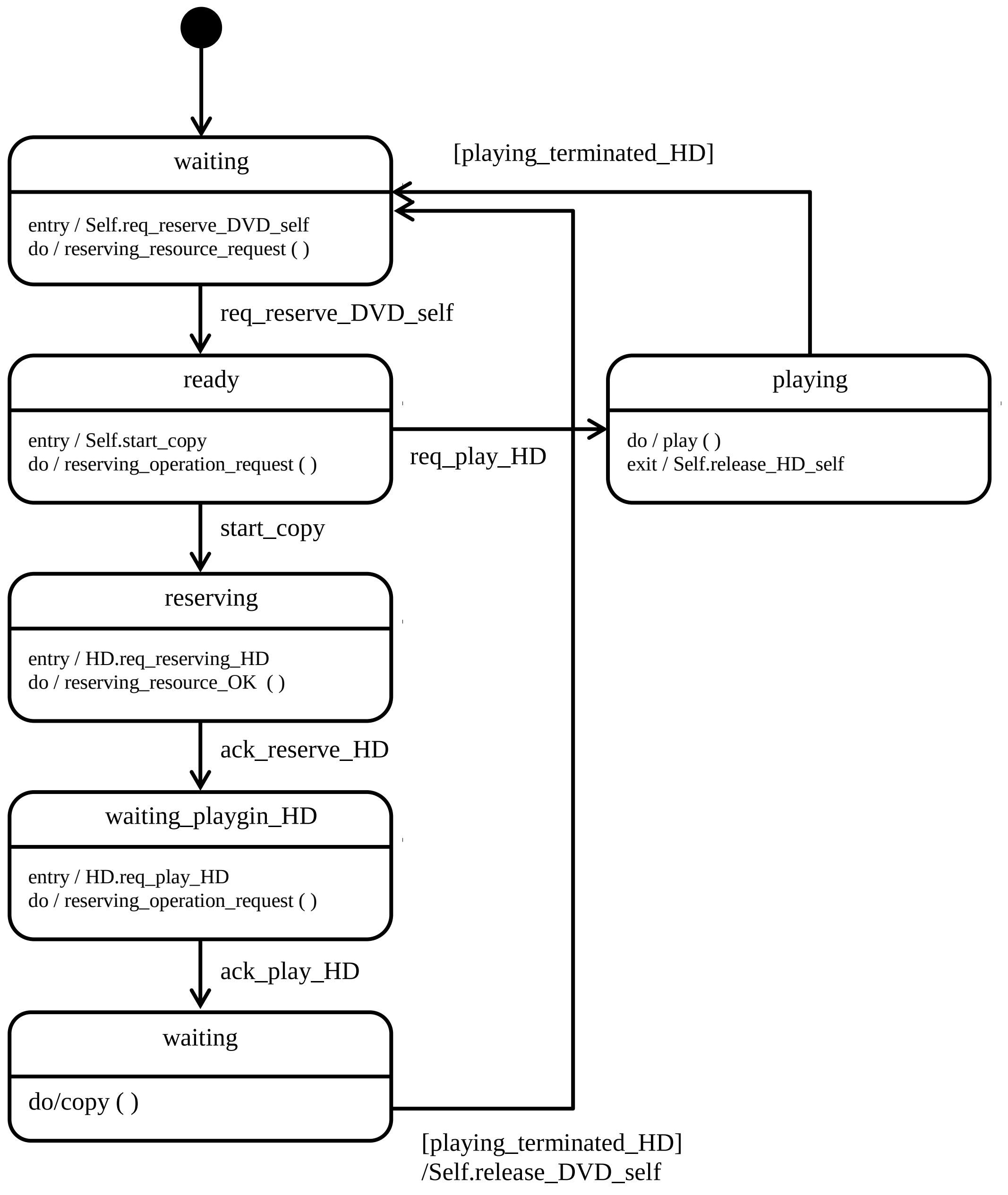
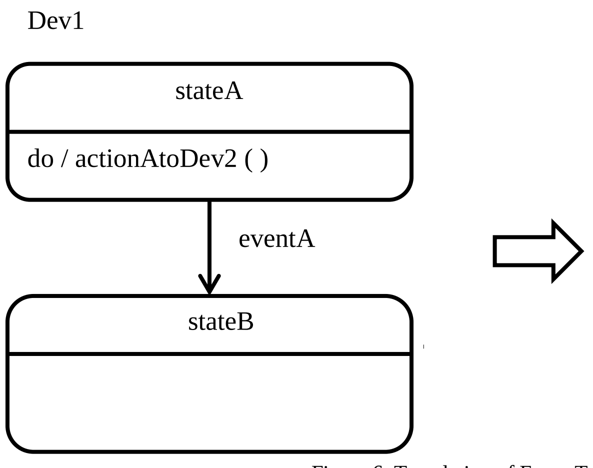


Figure 5. Statemachine Diagram of DVD recorder



```
do
:: Dev1_ch?event →
  if
  :: (state == stateA &&
      event == eventA) →
       state = stateB;
       Dev2_ch!actionAtoDev2;
  :: (state == stateB &&
      ...
  fi
  ...
od
```

Figure 6. Translation of Event Transition into Promera Code

A state transition is translated into a Promela code fragment as Figure 6.

The Promela code of the AV appliance system is provided in the Appendix.

Using the SPIN model checker, one can verify whether LTL formulas are satisfied with respect to the model written in Promela. In this case, we set out the following properties, which represent the expected behavior.

Properties on Progress:

(1) If the HDD recorder reaches `waiting` and receives a `start_copy` event, the HDD recorder can transition to `playing` state:

$$\Box\Diamond(\text{HD\_state} = \text{waiting}).$$

(2) Property (1) also holds for the DVD recorder:

$$\Box\Diamond(\text{DVD\_state} = \text{waiting}).$$

Properties on Liveness:

(3) If the HDD recorder reaches waiting and receives a start_copy event, the HDD recorder can transition to playing state:

$$\Box(p_3 \rightarrow \Diamond q_3),$$

where p_3 is

$$(\text{HD\_state} = \text{waiting} \wedge \text{event\_HD}=\text{start\_copy})$$

and q_3 is

$$(\text{HD\_state}=\text{playing}).$$

(4) If the DVD recorder reaches waiting, then it can transition to ready state:

$$\Box(p_4 \rightarrow \Diamond q_4),$$

where p_4 is

$$(\text{DVD\_state} = \text{waiting})$$

and q_4 is

$$(\text{DVD\_state}=\text{ready}).$$

(5) If the DVD recorder reaches ready and receives a start_copy event, the DVD recorder can transition to copying state:

$$\Box(p_5 \rightarrow \Diamond q_5),$$

where p_5 is

$$(\text{DVD\_state} = \text{ready} \wedge \text{event\_DVD}=\text{start\_copy})$$

and q_5 is

$$(\text{DVD\_state}=\text{copying}).$$

(6) If the DVD recorder reaches ready and receives a start_play event, the DVD recorder can transition to playing state:

$$\Box(p_6 \rightarrow \Diamond q_6),$$

where p_6 is

$$(\text{DVD\_state} = \text{ready} \wedge \text{event\_DVD}=\text{start\_play})$$

and q_6 is

$$(\text{DVD\_state}=\text{playing}).$$

(7) If the DVD recorder reaches copying, then it can transition to waiting state:

$$\Box(p_7 \rightarrow \Diamond q_7),$$

where p_7 is

$$(\text{DVD\_state} = \text{copying})$$

and q_7 is

$$(\text{DVD\_state}=\text{waiting}).$$

Safety

(8) The system cannot be deadlocked.

(9) The system cannot be livelocked.

The Promela code describing the AV appliance system satisfies these nine properties.

III. Modeling of Faults

In this section, we discuss the modeling of faults in the AV appliance system. In a later following section, we investigate fault-tolerance of the modeled faults using the SPIN model checker.

There are various kinds of faults in distributed systems. We focus on the following three cases:

Communication Fault
One instrument sends incorrect messages to another;

Sudden Termination Fault
The system suddenly and unexpectedly stops;

Irregular Transition Fault
The system makes an irregular transition.

Although these three behaviors are not improbable in normal situations, they are also possible in cases of hardware and network problems.

A. Modeling and Checking Communication Faults

In this section, we suppose that a communication fault develops between the HDD recorder and the DVD recorder in which the HDD recorder sends an erroneous message to the DVD recorder. For instance, the HDD recorder sends ack_play_HD instead of ack_reserve_HD. This faulty action is formulated by rewriting the Promela code as:

```
do ::HD_ch ? event ->
  if
    …
   ::(HD_state==waiting &&
      event==req_reserve_HD)->
        HD_state=ready;
/* Here is changed. */
        DVD_ch ! ack_play_HD;
  fi
    …
od
```

If we apply the SPIN model checker to the disordered model, we know that the properties (4), (6), (7), and (9) remain satisfied; on the other hand, (1), (2), (3), (5), and (8) are not satisfied by this model. After consideration of the result, we know that

if the DVD recorder receives an erroneous message `ack_play_HD`, the transition condition

```
DVD_state == reserving
&& event_DVD == ack_reserve_HD
```

is not satisfied, and consequently, the system is stalled.

B. Modeling and Checking Sudden Termination Faults

In this section, we consider the sudden termination of a part of the AV appliance system: the HDD recorder unexpectedly stops. To formalize this termination, we introduce a state \textrm{idol} and incorporate a transition that loops on the state into the model.

Specifically, we make the following changes in the Promela code:

```
do :: HD_ch ? event_HD ->
  …
  /* Additional Part */
  :: HD_state == ready ->
     HD_state = idle;
     clock = 0;
  /* Looping State */
  :: HD_state == idle ->
     if
/* non-deterministic choice */
           :: HD_state = idle ;
           :: HD_state = ready;
     fi
od
```

If this kind of fault is incorporated into the system, then the SPIN model checker gives the following result:

- the properties (4), (6), (7), and (8) remain satisfied;
- on the other hand, (1), (2), (3), (5), and (9) are not satisfied by the model.

This result indicates that:

The HDD recorder stays in `idle` state. On the other hand, the DVD recorder is stalled waiting for an appropriate message from the HDD recorder in waiting state or `playing` state.

C. Modeling and Checking Irregular Transition Faults

In this section, we consider the sudden termination of a part of the AV appliance system: the HDD recorder unexpectedly stops. To formalize this termination, we introduce a state idol and incorporate a transition that loops on the state into the model. Specifically, we make the following changes in the Promela code:

```
do
  :: HD_ch ? event ->
  …
  /* incorporated fault */
  :: HD_state == waiting ->
     HD_state = playing;
  …
```

If we verify the disordered model using the SPIN model checker, we know that the properties (3), (4), (6) and (7) remain satisfied; however, (1), (2), (5), (8), and (9) are not satisfied by the model. From this result, we know that:

The part of the recorder from which an event message ack_reserve_HD is sent does not start, and consequently, the condition
(DVD_state==reserving
&& event_DVD == ack_reserve_HD)
is not satisfied. Thus, the DVD recorder is stalled.

IV. Improving Robustness Using Model Checking

In the previous section, we showed that local trouble in part of the system can cause breakdowns in other parts. If a problem occurs in the HDD recorder and the copy action of the DVD recorder is initiated, then the DVD recorder is stalled. Local faults occurring in part of the system should be contained and their influence on other parts of the system should be minimized. We call this kind of property "*software robustness*." In this section, we propose a methodology for improving and redesigning software robustness using model checking, and explore it by rewriting Promela codes mentioned in previous sections.

From the previous sections' results, we suggest that the following properties should be satisfied when faults occur in the system.

(a) The system notifies the user of the occurrence of the problem.
(b) If the HDD recorder has a fault, the playing function of the DVD recorder should be maintained.
(c) If the HDD recorder has a fault, the system should not stall or fall silent.

To improve the system with respect to (a), we add a *warning lamp* to the system. In Promela, we represent it as a variable, error_lamp. Moreover, we introduce a variable, error, which is used as an indicator of fault occurrence. Items (a), (b) and (c) are modeled as the following LTL formulas for the model to be satisfied.

(10) If the DVD recorder is in ready state and receives start_copy event message, the DVD recorder transitions to copying state, or the warning lamp error_lamp activates:

$$\Box((\text{DVD\_state}=\text{ready}) \wedge (\text{event\_DVD}=\text{start\_copy}) \rightarrow \Diamond(\text{DVD\_state}=\text{copying} \vee \text{error\_lamp}=\text{HD})).$$

The value HD means that error_lamp is activated.

(11) If no fault occurs, the warning lamp does not activate:

$$\Box\neg(\text{error}=0 \wedge \text{error\_lamp}=\text{HD}).$$

(12) If a fault occurs, the warning lamp is eventually activated:

$$\Box((\text{error}=1) \rightarrow \Diamond(\text{error\_lamp}=\text{HD})).$$

The conditions (a), (b), and (c) are formulated by the LTL formulas (1), … , (12) as follows

- for condition (a), formulas (10) and (12) are required;
- for condition (b), (2),(4), and (6) are required;
- for condition (c), (8) and (9) are required.

Condition (11) represents correct error handling.

In the following section, we discuss how to improve the system by checking the LTL formulas with the SPIN model checker.

A. Improving robustness against communication faults

If the HDD recorder receives an event message other than ack_reserve_HD while it is waiting for ack_reserve_HD, the system activates the warning lamp and transitions to waiting. This error handling is formulated as an inline macro:

```
inline handle_error(){
  error_lamp = HD;
  DVD_state = waiting;
}
```

and is inserted as follows:

```
:: (DVD_state==reserving)->
if
::(event_DVD==ack_reserve_HD)->
    normal processing
::(event_DVD!=ack_reserve_HD)->
    handle_error()
```

This is also applied to the code for waiting for the ack_play_HD event message.

Model checking also demonstrates that the DVD recorder is stalled in "reserving" state after the HDD recorder sends the event message "req_reserve_HD" in the modified Promela code. To remedy this flaw, we introduce timeout detection, which enables the system to transition if there is no possibility of other transition triggers. Concretely, we add code that executes "error_lamp = HD" and transitions to waiting state if the value of "timeout" becomes true.

```
unless {
  timeout == 1 ->
    error_lamp = HD;
    DVD_state == waiting;
}
```

B. Improving robustness against sudden termination faults

If the DVD recorder waits for a response from the HDD recorder and the HDD recorder does not reply to the DVD recorder, then the DVD recorder should stop waiting, regard the situation as erroneous, and apply a solution.

The primitive timeout in Promela is merely an instruction to wait for other processes to stop. We therefore explicitly introduce a clock variable "clock" into the system. Due to restrictions in a number of states in the SPIN model checker, the value of "clock" is not more than 10; if it is incremented repeatedly, it does not become more than 10.

Increment is formulated as an online macro of Promela, "handle_increment_clock", as follows:

```
inline handle_increment_clock()
  {
    if ::(clock< 9)->clock++;
       ::(clock==9)->
          error=1;
          clock++;
       ::(clock==10)
          /* do nothing */
```

The online macro "handle_increment_clock" is inserted into the idling part of the system.

```
:: HD_state == idle
/* nondeterministic choice */
if :: HD_state=idle;
      handle_increment_clock()
   :: HD_state=ready;
      handle_increment_clock()
fi
```

After these improvements, the SPIN model checker shows that the properties other than (1), (3), (5) are satisfied by the modified model.

V. Concluding Remarks

In this paper, we proposed a methodology for destructive testing using the SPIN model checker. We used as an example a simple AV appliance system consisting of a DVD recorder and a HDD recorder, and provided a model in Promela modeling language. We then introduced faults into the model and analyzed the behavior of these faults. Investigating the result of the analysis, we improved the model from the viewpoint of fault tolerance and evaluated the improved model using SPIN.

The destruction of the AV appliance system, that is, the intentional introduction of faults to the model, was not automatic but manual. Thus, we may have overlooked other ways in which faults could occur. In the continuation of our work, we will investigate the automatic occurrence of faults. One of the promising approaches to this issue is considered as formal modeling in the process calculus [7,8].

Another interesting approach is Reliability Engineering, for example, Fault Tree Analysis [12,13]. Introducing this paper's results to such research areas is also promising.

Appendix A Modeling the system in Promela

We provide a Promela code that describes the AV appliance system before adding intentional faults.

```
mtype={
  /* states */
  waiting, ready, playing, reserving,
  waiting_HD_playing, copying, playing_self,
  /* event messages */
  start_play, start_copy,
  ack_reserve_HD, ack_play_HD, req_reserve_HD,
  req_play_HD, playing_terminated_HD, DVD, HD,
};

/* state variables */
show mtype DVD_state = waiting;
show mtype HD_state = waiting;
```

```
/* communication channels  */
chan DVD_ch = [0] of { mtype };
chan HD_ch = [0] of { mtype };
mtype event_HD; mtype event_DVD;

active proctype HD_recorder() {
  do :: HD_ch?event_HD->
    if ::(HD_state == waiting &&
          event_HD==req_reserve_HD)->
             HD_state = ready;
             DVD_ch!ack_reserve_HD;
         ::(HD_state == ready &&
            event_HD == req_play_HD)->
             progress1:
             HD_state = playing;
             DVD_ch!ack_play_HD;
         :: else -> skip
       fi
     :: (HD_state == playing ) ->
       DVD_ch!playing_terminated_HD;
       HD_state = waiting;
  od
}

active proctype DVD_recorder() {
  do :: DVD_ch?event_DVD ->
        if :: (DVD_state == reserving &&
            event_DVD == ack_reserve_HD)->
              progress2:
              HD_ch!req_play_HD;
              DVD_state = waiting_HD_playing;
         ::(DVD_state == waiting_HD_playing
            && event_DVD == ack_play_HD) ->
            DVD_state = copying;
         ::(DVD_state == copying &&
            event_DVD==playing_terminated_HD->
            DVD_state = waiting;
         ::else -> skip;
       fi
    :: (DVD_state == ready &&
        event_DVD == start_play)->
       DVD_state = playing_self
    :: (DVD_state == ready &&
        event_DVD == start_copy)->
       HD_ch!req_reserve_HD;
       DVD_state = reserving
    ::(DVD_state == waiting)->
       DVD_state = ready ->
         if :: event_DVD = start_play
            :: event_DVD = start_copy
       fi
    ::(DVD_state == playing_self) ->
      progress3:
      DVD_state = waiting;
  od
}
```

ACKNOWLEDGMENT

This work was supported by Grants-in-Aid for Scientific Research (C) (24500009). This paper is an extended work of [11].

REFERENCES

[1] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley Professional, 2003.*

[2] P. Pettersson and K. G. Larsen, “UPPAAL2k”, *Bulletin of the European Association for Theoretical Computer Science,* vol. 70, pp. 40—44, 2000.

[3] M. Huth and M. Ryan, *Logic in Computer Science: Modelling and Reasoning about System.* Cambridge University Press, 2004.

[4] P. J. Denning, “Fault tolerant operating systems”, ACM Computing Surveys, vol. 8, no. 4, pp. 359—389, 1976.

[5] B. Randell, P. Lee, and P. C. Treleaven, “Reliability issues in computing system design,” *ACM Computing Surveys,* vol. 10, no. 2, pp. 123—165, 1978.

[6] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2nd Edition).* Addison-Wesley, 2005.

[7] D. Tomioka, S. Nishizaki, and R. Ikeda, “A cost estimation calculus for analyzing the resistance to denial-of-service attack,” in *Software Security – Theories and Systems,* Lecture Notes in Computer Science, vol. 3233, 2004, pp. 25—44.

[8] R. Ikeda, K. Narita and S. Nishizaki, “Cooperation of model checking and network simulation for cost analysis of distributed system,” *International Journal of Computers and Applications,* vol. 33, no. 4, pp. 323—329, 2011.

[9] S. Nishizaki and H. Tamano, “*Design of Open Equation Archive Server Resistant Against Denial-of-Service Attacks*”, accepted and to appear in the Proceedings of International Conference on Advances in Information Technology and Mobile Communication – AIM2012.

[10] T. Sasajima and S. Nishizaki, “*Blog-based Distributed Computation – Implementation of Software Verification System*”, accepted and to appear in the Proceedings of the 3rd International Conference on Information Computing and Applications ICICA2012, 2012.

[11] H. Kumamoto, T. Mizuno, K. Narita, S. Nishizaki, “Destructive testing of software systems by model checking”, In *the Proceedings of International Symposium on Communications and Information Technology (ISCIT 2010)*, IEEE, pp. 26-29, 2010.

[12] W. E. Vesley, F. F. Goldberg, N. H. Roberts and D. F. Haasl: *Fault Tree Handbook*, Office of Nuclear Regulatory Research, 1981.

[13] A. Thums and G. Schellhorn, “Model Checking FTA”, In *FME 2003: Formal Methods*, Lecture Notes in Computer Science, vol. 2805, Springer-Verlag, pp. 739-757, 2003.

[14] S. Nishizaki and T. Ohata “Real-Time Model Checking for Regulatory Compliance”, accepted and to appear in the *Proceedings of International Conference on Advances in Information Technology and Mobile Communication – AIM2012,* 2012.

[15] G. Schellhorn, A. Thums, W. Reif, “Formal Fault Tree Semantics”, in *IDPT-2002,* Society for Design and Process Science (2002) pp. 739-757.

Hiroki Kumamoto received his M. Eng. Degree from the Tokyo Institute of Technology in 2011. He contributed to this paper when he was a student of the Tokyo Institute of Technology. Now he belongs to ACCESS, Co., Ltd.

Takahisa Mizuno received his M. Eng. Degree from the Tokyo Institute of Technology in 2012. He contributed to this paper when he was a student of the Tokyo Institute of Technology. Now he belongs to IBM Japan, Ltd.

Shin-ya Nishizaki is an Associate Professor of Computer Science at Tokyo Institute of Technology, Japan, where he leads a research group on formal theory on software systems. He received his Bachelors, Master's and Doctorate degrees from Kyoto University, in mathematical science. Before joining Tokyo Institute of Technology in 1998, Dr Nishizaki held appointments in computer science as Associate Professor at Chiba University for 2 years and Assistant Professor at Okayama University for 2 years.