PAPER *Special Section on Concurrent System Technology and Its Application to Multiple Agent Systems*

Verifying Fault Tolerance of Concurrent Systems by Model Checking

Tomoyuki YOKOGAWA[†a], *Nonmember*, **Tatsuhiro TSUCHIYA**[†], *and* **Tohru KIKUNO**[†], *Regular Members*

SUMMARY Model checking is a technique that can make a verification for finite state systems absolutely automatic. We propose a method for automatic verification of fault-tolerant concurrent systems using this technique. Unlike other related work, which is tailored to specific systems, we are aimed at providing an approach that can be used to verify various kinds of systems against fault tolerance. The main obstacle in model checking is state explosion. To avoid the problem, we design this method so that it can use a symbolic model checking tool called SMV (Symbolic Model Verifier). Symbolic model checking can overcome the problem by expressing the state space and the transition relation by Boolean functions. Assuming that a system to be verified is modeled as a guarded command program, we design a modeling language and propose a translation method from the modeling language to the input language of SMV. We show the results of applying the proposed method to various examples to demonstrate the feasibility of the method.

key words: *symbolic model checking, fault tolerance, SMV, concurrent systems, guarded command*

1. Introduction

In recent years, the growing demand for high availability and reliability of computer systems has led to a formal verification for fault-tolerant systems [12], [15], [17], [19]. Methods for formally verifying fault tolerance are classified into *deductive verification* and *model checking.*

The term *deductive verification* normally refers to the use of axioms and proof rules to prove the correctness of systems. The importance of deductive verification is widely recognized by computer scientists. There are some examples where the fault tolerance was verified by deductive verification [15], [19]. However, deductive verification is a time-consuming process that can be performed only by experts who are educated in logical reasoning and have considerable experience. The proof of a single protocol or circuit can last days or months. Moreover deductive verification cannot be performed fully automatically; thus the use of it is rare. An advantage of deductive verification is that it can be used for reasoning about infinite state systems. However, no limit can be placed on the amount of time or memory that may be needed in order to find a proof.

Model checking is an automatic technique for verifying concurrent systems. That can be performed absolutely automatically in stead of restriction that it can verify only finite state systems. Because of this property, it is preferable to deductive verification, whenever it can be applied. There are also some examples where fault tolerance property of concurrent systems was verified by model checking [4], [14], [20]. However, these methods for verification are specialized for specific systems, and a general approach does not exist.

In this paper, we propose for automatic verification of fault tolerance of concurrent systems using model checking. Our aim is to provide a single method that can be applied to various kinds of systems. We achieve this goal to adopting a model of fault-tolerant systems that is proposed by Arora and Gouda [2]. In recent years, the model has been accepted as a fundamentals for building and reasoning about fault-tolerant systems.

Of course, there are always situations where problem-specific properties, which cannot be handled by our method, need to be verified. This is most likely when the designs to be verified are detailed. However, we believe that our method is still useful especially in early stages of development, where designs are in highly abstract level.

This method uses a symbolic model checking tool called SMV (Symbolic Model Verifier) [18]. SMV is a tool for checking that finite-state systems described by the input language of SMV satisfy specifications given in CTL (Computation Tree Logic) [9].

In this paper we assume that a system to be verified is given in the form of a guarded command program [2]. We design a modeling language suited for describing guarded command programs, and then we propose a translation method from the modeling language to the SMV language. We present the CTL formula that describes fault tolerance. Finally we apply the proposed method to some examples to demonstrate the usefulness of the method.

Manuscript received March 22, 2002.
Manuscript revised June 19, 2002.
Final manuscript received August 5, 2002.

[†]The authors are with the Graduate School of Engineering Science, Osaka University, Toyonaka-shi, 560-8531 Japan.
a) E-mail: t-yokoga@ics.es.osaka-u.ac.jp

2. Model of Fault-Tolerant Systems

2.1 Guarded Command Programs

To describe systems to be verified, we adopt the model proposed in [2]. A system is described as a *program* that consists of a set of *variables* and a finite set of *processes*. Each variable has a predefined nonempty domain. Each process consists of a finite set of *actions*. Each action consists of a *guard* and a *statement*, where the guard is a Boolean expression over program variables, and the statement is a set of assignments that updates zero or more program variables and always terminates upon execution. The action is described in the form

$$\langle guard \rangle \rightarrow \langle statement \rangle.$$

A *state* of the system is defined as a valuation values of the program variables. Therefore a Boolean expression over the program variables describes a set of states where that expression evaluates to true, and a state transition is described by assignments that update the program variables.

An action is enabled at a state iff its guard evaluates to true at that state. At each state, a process is selected non-deterministicly, and if there exist enabled actions in the process, one of them is also selected non-deterministicly and then the statement updates the program variables. State transitions thus occur by execution of actions.

We assume that the sequence of state transitions is process-fair; that is, any process is infinitely often chosen for execution.

2.2 Faults

A formal approach to defining the term “fault” is usually based on the observation that systems change their state as a result of two quite similar event classes: normal system operation and fault occurrences [11]. Thus, a fault can be modeled as an unwanted (but nevertheless possible) state transition of a process. By using additional (virtual) variables to extend the actual state space of a process, various kinds of faults, such as, crash faults, omission faults, or some type of Byzantine faults, can be represented [1], [2], [13], [23].

In this model, we describe the occurrences of faults, that is, the unwanted transitions, by a set of actions, F, over the variables of the program.

We refer to actions in F as *fault actions*. These three types of faults are modeled by actions as follows.

(1) Crash faults

First, we add a Boolean variable *up* to the process and set the initial value of *up* to *true*. In addition, the guard of each action of the process is modified to the conjunction of the guard and *up*, as shown below.

$$up \wedge \langle guard \rangle \rightarrow \langle statement \rangle.$$

This means that no action is selected when $up = false$. Finally the fault action

$$fault : true \rightarrow up := false$$

is added to the process. If the action is selected, *up* is set to $false$ and no action becomes selectable from then. We thus can represent a crash fault.

(2) Omission faults

A fault that causes a process to not respond to some inputs is called an omission fault. This type of a fault can be represented in the same way as crash faults, except that an additional action

$$up \rightarrow up := false$$

is needed to represent that the process behaves incorrectly intermittently.

(3) Byzantine faults

Byzantine fault refers to fault which causes the process to behave in totally arbitrary manner. Incorrect computation faults are an important subset of the Byzantine fault. With this type of fault, a process simply produces an incorrect output. Consider the following action

$$\langle guard \rangle \rightarrow v := val_k.$$

Where v is a variable that has the range of values $\{val_1, \cdots, val_n\} (1 \leqq k \leqq n)$.

In the case, we can represent an incorrect computation fault by adding the fault action

$$fault : \langle guard \rangle \rightarrow v := \{val_1, \cdots, val_n\}$$

to the process. If the action is selected, the value of v changes arbitrarily.

2.3 Fault Tolerance

In the model, the fault tolerance of the system is formally defined as follows. We assume that a Boolean expression S that represents *legal* states is given. In addition, we assume that S is never invalidated by non-fault actions. This property is referred to as the *closure* property.

These assumptions stem from the following observation. A well-established method for verifying fault-free systems is to detect a predicate that is true throughout system execution. Such an invariant predicate identifies the legal states of system and asserts that the set of legal states is closed under system execution without fault. For example, Arora and Kulkarni

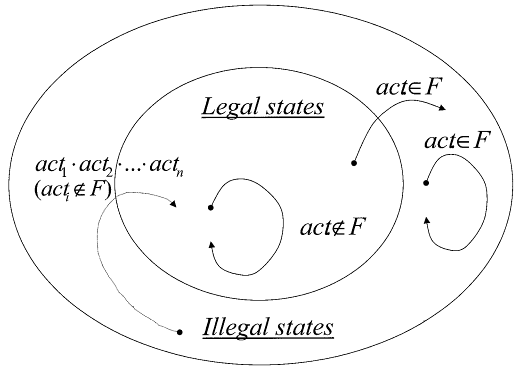


Fig. 1 Schematic overview of the fault tolerance property.

proposed a methodology for constructing fault-tolerant systems systematically [3]. In the methodology, fault-tolerant programs are incrementally constructing from non-fault-tolerant systems and each step of the construction, an invariant property is required to be identified and verified. Following this observation, we require that for each fault-tolerant system there exists a predicate S that is invariant throughout fault-free system execution.

Let c be any legal state, that is, any state of the program where S holds. If S is not invalidated in c by any action in the set F of fault actions, then the program is said to be tolerant to F. (This type of fault tolerance is referred to as *masking* fault tolerance.)

Otherwise, executing an enabled action in F in c may yield an illegal state, where $\neg S$ holds. If continuous execution of a sufficiently large number of actions that are not in F always yields a legal state from any illegal state, then the program is also said to be tolerant to F. (This type of fault tolerance is referred to as *nonmasking* fault tolerance.) Figure 1 illustrates this concept.

3. Model Checking

Model checking is an automatic technique for verifying finite state concurrent systems. Model checking methods search the finite state space to determine if some specification is true or not [10]. One benefit of the restriction to finite state systems is that verification can be performed automatically. Although this restriction may seem to be a major disadvantage, model checking is applicable to several very important classes of systems. For example, hardware controllers are finite state systems, and so are many communication protocols. In many cases errors can be found by restricting unbounded data structures to specific instances that are finite state. For example, programs with unbounded message queues can be debugged by restricting the size of the queues to a small number like two or three.

3.1 Symbolic Model Checking

The main challenge in model checking is dealing with the state explosion problem. The problem occurs in systems with many components that can interact concurrently. To cope with the problem, a method has been proposed that expresses the state space and the transition relation by Boolean functions, and verifies systems by processing the Boolean functions. This method is called *symbolic model checking* [7], [18].

In symbolic model checking, a Boolean function is expressed by using OBDDs (Ordered binary decision diagrams) [6]. OBDDs provide a canonical form for Boolean functions that are often substantially more compact than conjunctive or disjunctive normal form, and very efficient algorithms have been developed for manipulating them. Therefore, it achieves the conciseness to express the state space and the transition relation, and enables the avoidance of the state explosion problem.

In model checking, it is necessary to describe the properties that the system must satisfy as a specification. The specification is usually given as a formula in some logic. For concurrent systems, it is common to use temporal logic, which can assert how the behavior of the system evolves over time. A well-used temporal logic is CTL [9]. Time is not mentioned explicitly in CTL; instead a formula might specify that *eventually* some designated state is reached, or that an error state is *never* entered. Properties like *eventually* or *never* are specified using special temporal operators. These operators can be combined with Boolean connectives or nested arbitrarily.

CTL formulas describe properties of *computation trees*. The tree is formed by unwinding the execution sequences into an infinite tree with the designated initial state at the root. The computation tree shows all of the possible executions starting from the initial state. In CTL, formulas are composed of path quantifiers and temporal operators. The path quantifiers are used to describe the branching structure in the computation tree. There are two such quantifiers; one is A ("for all computation paths") and another is E ("for some computation path"). These quantifiers are used in a particular state to specify that all of the paths or some of the paths starting at that state have some property. The temporal operators describe properties of a path through the tree. There are four basic operators, X ("next time"), F ("in the future"), G ("globally"), U ("until"). In this paper, we use only AF and AG. The formula AGp holds in state s if p holds in all states along all sequences of states starting from s, while the formula AFp holds in state s if p holds in some states along all sequences of states starting from s. An atomic proposition is a CTL formula. If f and g are CTL formulas, so are $\neg f$, $f \wedge g$, $f \vee g$, AFf, and AGf.

```
MODULE main
VAR   request:boolean;
      state:{ready, busy};
INIT  state = ready
TRANS (state = ready & request)
      & next(state) = busy
SPEC  AG(request -> AF state = busy)
```

Fig. 2 SMV program.

3.2 Symbolic Model Verifier (SMV)

SMV is a software tool that implements symbolic model checking [18]. It is based on a language for describing hierarchical finite-state systems. Programs described in the language contain specifications expressed by CTL. The model checker extracts a state space and a transition system from a program in the SMV language and uses an OBDD-based search algorithm to determine whether the system satisfies its specification. If the system does not satisfy the specification, the verification tool will produce an execution trace that shows why the specification is false.

Figure 2 is an example of an SMV program. A state of the transition system is represented by a collection of state variables. The variables may be of Boolean, integer subrange, or enumerated type. The keyword `VAR` is used to declare variables. The variable `request` is declared to be a Boolean in the program, while the variable `state` can take on the symbolic values `ready` or `busy`.

In the SMV language, the transition relation is described by specifying changes of the values of variables with `ASSIGN` declaration, or by using a Boolean-valued function with `TRANS` declaration. When using `ASSIGN`, the change of the value is individually described for every variable. This is not appropriate for describing guarded command programs in which each action updates multiple variables and selection of actions can be non-deterministic. We therefore use `TRANS` and describe the transition relation as a Boolean formula over the program variables. Similarly, initial states are described by a Boolean formula.

Specifically the transition relation is a set of the pairs of the current state and the next state that satisfy the Boolean formula defined in the `TRANS` statement. Also the initial states are a set of states where the Boolean formula defined in the `INIT` statement holds. The expression `next(x)` is used to refer to the variable `x` in the next state.

The specification is described as a formula in CTL under the keyword `SPEC`. SMV verifies whether all possible initial states satisfy the specification. In this case, the specification signifies that invariantly if `request` is true, then eventually the value of `state` will be `busy`.

In model checking, only the correctness along fair computation paths is interested in many cases. For example, we do not consider a computation where a certain process has never selected as an object of verification. Such properties are expressed by keyword `FAIRNESS` in SMV. The keyword `FAIRNESS` and a CTL formula force SMV to verify only computation paths where the associated CTL formula becomes true infinitely often.

4. The Proposed Modeling Language

To describe and verify fault-tolerant systems, we propose a modeling language for describing guarded command programs. By translating programs written in this language into the SMV language, it becomes possible to model check fault tolerance. Using this proposed language, we need not describe the non-determinism of systems, the fairness property of the selection of processes and the fault-tolerance property explicitly. We note that because of the lack of flexibility of the SMV language, it is difficult and tedious to represent these properties by hand. By representing a given system as a guarded command program and the legal states as a Boolean formula, we can verify fault tolerance automatically. In this section, we show the syntax of the modeling language and explain how to translate it to the SMV language.

4.1 Syntax

The program is described in the following form.

```
program :: "program"
             macros_definition
             legal_states_description
             process_description1
             process_description2
             ...
```

The set of legal states is specified as a Boolean formula.

```
legal_states_description :: "spec" expression
```

The processes are described in the following form.

```
process_description :: "process" process_name
                         "begin"
                           var_declaration
                           macros_definition
                           action_description
                           fault_description
                         "end"
```

The variables of a process are declared with two elements. One is the type of the variable, while the other is a set of the initial values of the variable.

The type associated with a variable declaration can be either Boolean, a set of integers, or enumeration of symbols. An integer type is defined either by upper and lower bounds like $\{1\dots5\}$ or by an enumeration of elements like $\{1,2,3,4,5\}$.

Actions (including fault actions) which specify the transition relation of the system are described in the following form.

```
action_description :: "action" seq_of_actions
fault_description  :: "fault" seq_of_actions
seq_of_actions     :: action1 ";"
                      action2 ";" ...
action             :: guard ":>" statement ";"
guard              :: expression
statement          :: assignment1 ","
                      assignment2 "," ...
assignment         :: left ":=" right
left               :: variable_name
                      | process_name "." variable_name
right              :: expression
                      | "{" val1 "," val2 "," ... "}"
```

The left hand side of an assignment denotes the variable that will change by the action. If the right hand side is an expression, the assignment means that the variable changes to the value of the right hand side. On the other hand, if the right hand side is a set, the variable changes to one value of the set non-deterministicly.

4.2 Translation Method to the SMV Language

4.2.1 Action

As stated above, an action is represented in the proposed language as follows.

```
P :> x1:=expr1, x2:=expr2, ··· , xn:=exprn
```

The changes of the variable values caused by the action can be represented as a Boolean formula `next(x1)=expr1 & next(x2)=expr2 & ··· & next(xn)=exprn`. Note that the action can be selected only in the states represented by a Boolean formula `P`. Consequently, the state transition by the action is described as the following formula.

```
P & next(x1)=expr1 & ··· & next(xn)=exprn
  & next(y1)=y1 & ··· & next(ym)=ym
```

Here `y1,··· ,ym` are the variables that do not change in the next state. The formula holds iff this action is enabled and the value of each variable in the next state is assigned as designated by this action. The formula thus represents that this action is selected. A fault is expressed as an action and can be described similarly.

Let a_i be this formula for an action. Then the transition of a process that has N actions is expressed as formula $A = a_1 \vee a_2 \vee \cdots \vee a_N$.

4.2.2 State Transitions

Let A_i be a formula that expresses the transitions of process i. Since only one process is selected simultaneously, the transitions of the system that has m processes is represented as formula $(A_1 \wedge run_1) \vee (A_2 \wedge run_2) \vee \cdots \vee (A_m \wedge run_m)$, where a Boolean variable run_i represents that a process i is selected. The constraint that only one process is selected can be expressed by setting only one element in $run_1, run_2, \cdots, run_m$ to true.

Consequently the transition relation of the system is represented as the following formula.

$$(A_1 \wedge run_1) \vee (A_2 \wedge run_2) \vee \cdots \vee (A_m \wedge run_m) \wedge \\ ((run_1 \wedge \neg run_2 \wedge \cdots \wedge \neg run_m) \vee (\neg run_1 \wedge run_2 \wedge \\ \cdots \wedge \neg run_m) \cdots \vee (\neg run_1 \wedge \neg run_2 \wedge \cdots \wedge run_m))$$

We assume the fairness for selection of processes; that is, each process must be selected infinitely often. Thus, only execution sequences where each run_i holds infinity often are verified. This can be specified using `FAIRNESS` as follows.

```
FAIRNESS run1
      ⋮
FAIRNESS runm
```

The set of initial states is also described by a Boolean formula. When a variable x has initial values $x_0, x_1, \cdots$, the set of states where x has the initial values is described by a formula $(x = x_0) \vee (x = x_1) \vee \cdots$. Since the initial states are those where such a formula holds for each variable, the conjunction of each formula represents the set of the initial states.

4.2.3 Specifying Fault Tolerance

To use SMV, we have to express the property to be verified as a formula of a temporal logic called CTL.

So far we have shown the method for expressing the transition relation of a system in the SMV language, without considering verification of fault tolerance. In order for verification to be carried out, it is necessary to describe the fault tolerance property explicitly. For this purpose, we introduce a Boolean variable f and modify guards of fault actions such that they can be selected only when $f = 0$. When $f = 1$, only non-fault actions are selected.

We let the value of f to change as follows. If the system is in the legal states, the value of f is always false. If the system is not in the legal states, the value of f changes to true or false non-deterministicly. Once the value of f has changed to true, it remains true invariantly in the illegal states. This is intended to represent the fact that faults will stop occurring. If the system has come back to the legal states, the value of f changes to false.

The guard of each fault action is modified as follows. Suppose that a fault action is given in the modeling language as shown below.

```
P :> v1:=expr1, v2:=expr2, ···, vn:=exprn
```

Then the condition of execution is modified to $P \wedge \neg f$, and another action is obtained as follows.

$$P \wedge \neg f \to v_1 := expr_1, v_2 := expr_2, \cdots, v_n := expr_n$$

Note that the only difference between the two actions is that the latter is not enabled when f is true. In addition, we do not exclude the possibility that f is always false. Thus this modification does not deviate the resulting transition system from the behavior of the given program.

The change of the value of f is described as the following formula F. Here S is a Boolean formula that represents the set of legal states.

```
F = S & next(f)=0
    | !S & !f & (next(f)=1|next(f)=0)
    | !S & f & next(f)=f
```

(`!` represents negation.)

The value of f changes independently from the executions of actions. So by adding F to the formula that represents the behavior of the whole system as a conjunctive, the changes of the value of f are incorporated into the transition relation. Thus the `TRANS` statement becomes as follows.

```
TRANS
  (A1 & run1 | A2 & run2 | ··· | Am & runm)
  & (  ( run1 & !run2 & ··· & !runm)
                 ⋮
     | (!run1 & !run2 & ··· &  runm))
  & F
```

Using f, the property to be verified, that is, the fault tolerance, is expressed in CTL as follows.

$$AG(f \to AF(S))$$

This CTL formula expresses the property that if f holds, then S will always hold eventually. The CTL formula holds iff the system is either masking fault-tolerant or non-masking fault-tolerant. In the case of nonmasking fault-tolerant systems, even though the system falls into the illegal state where S does not hold by a fault action, if f changes to true and faults stop occurring, then S will hold eventually by execution of non-fault actions. In the case of masking fault-tolerant systems, S always holds. Thus $AF(S)$ always holds, so does this CTL formula.

The proposed method focuses on checking the fault tolerance property. Using different CTL formulas, however, other properties can also be verified. For example, the closure property can also be checked by using another CTL, as explained below.

Remark

As stated in 2.3, we assume that the closure property holds; that is, S is never invalidated by non-fault actions. It should be noted that we can also check the closure property as follows. First, with the method described in this section, a given guarded command program is translated into an SMV program by considering non-fault actions only. Second, CTL formula $AG(S \to AG(S))$, which represents the closure property, is checked by the SMV tool.

5. Case Studies

In this section, we show the results of applying the proposed method to several examples. These examples are known to be fault-tolerant and all verification results coincided completely. The first two examples, namely atomic commitment and Byzantine agreement protocols are masking fault-tolerant, while the third one is non-masking fault-tolerant. All experiments were performed on a Linux machine with a 500 MHz Pentium III processor and 256 Mbytes of memory.

5.1 Atomic Commitment Protocol

The first example is the *atomic commitment protocol* [2], [5]. In the protocol, each process casts one of two votes, Yes or No, then reaches one of two decisions, Commit or Abort. If no faults occur and all processes vote Yes, all processes reach a Commit decision. A process reaches a Commit decision only when all process voted Yes. And all processes that have reached a certain decision reach the same decision. We consider using the *two-phase commit protocol* to implement the atomic commitment protocol. We assume that faults may stop processes.

In the first phase, each process casts its vote and sends the vote to a distinguished *coordinator* process c. In the second phase, the coordinator process reaches a decision based on the votes received from other processes and broadcasts the decision to all processes.

The coordinator process c has the following two phases and can be described as three actions.

Phase 1: Process c casts its vote, enters the second phase, and starts waiting for the votes of other processes (the first action).

Phase 2: If c detects that all processes have voted Yes and not stopped, it reaches a Commit decision (the second action). If c detects that some process has voted No or has stopped, it reaches an Abort decision (the third action).

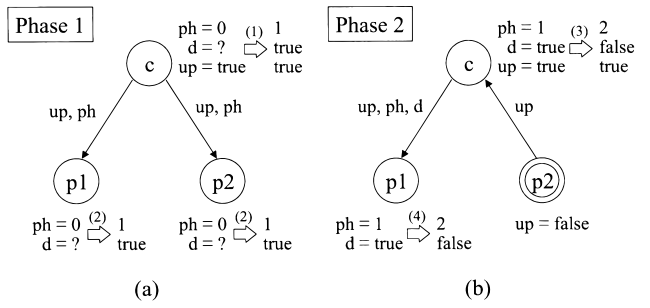


Fig. 3 Example of atomic commitment.

```
process c
 begin
 var
   ph : {0...2}{0};
   up : boolean{true};
    d : boolean{true, false};
 action
   up & ph=0 :> ph:=1, d:={false, true}, up:=up ;
   up & ph=1 &
   ((up & ph=1 & d) & (p1.up & p1.ph=1 & p1.d)
    & (p2.up & p2.ph=1 & p2.d) & ··· )
     :> ph:=2, d:=true, up:=up ;
   up & ph=1 &
   ((!up | (ph>=1 & !d)) | (!p1.up | (p1.ph>=1 & !p1.d))
    | (!p2.up | (p2.ph>=1 & !p2.d)) | ··· )
     :> ph:=2, d:=false, up:=up ;
 fault
   true :> ph:=ph, d:=d, up:=0;
 end
```

Fig. 4 The coordinator process.

Each process other than the coordinator has following two phases and can be described as three or more actions.

Phase 1: If the process detects that c has voted and entered the second phase, it casts its vote, enters the second phase, and starts waiting for the vote of some process (the first action). If the process detects that c has stopped, it reaches an Abort decision (the second action).

Phase 2: If the process detects that some process has not stopped and completed its second phase, reaches the same decision as that process has (the third or other actions).

Using Fig. 3, we illustrate how the protocol works. We assume that the number of processes is 3.

Phase 1: (Step 1) The coordinator process c casts its vote and the value of ph is set to 1. Here we assume that c votes Yes ($d = true$). (Step 2) Process p_1 and p_2 check that c has not stopped and voted ($c.up \wedge c.ph = 1$), and cast their votes. The value of $p_1.ph$ and $p_2.ph$ are set to 1.

Phase 2: Now suppose a crash fault occurs in process p_2. (Step 3) Since p_2 has stopped, c reaches an Abort decision. The value of ph is set to 2 and c completes the phase. (Step 4) Process p_1 checks that c has not stopped and has completed the second phase ($c.up \wedge c.ph = 2$) and reaches the same decision as c ($d = false$). The value of $p_1.ph$ is set to 2 and p_1 completes the phase.

```
const
   condition1 :=
     c.ph=0 ->
     (c.ph=0 | (c.ph=2 & !c.d))
     & (p1.ph=0 | (p1.ph=2 & !p1.d)) & ··· ;
   condition2 :=
     c.ph=1 ->
     (c.ph!=2 | !c.d) & (p1.ph!=2 | !p1.d) & ··· ;
   condition3 :=
     c.ph=2 & c.d ->
     (c.ph!=0 & c.d) & (p1.ph!=0 & p1.d) & ··· ;
   condition4 :=
     c.ph=2 & !c.d ->
     (c.ph!=2 | !c.d) & (p1.ph!=2 | !p1.d) & ··· ;
 spec
     condition1 & condition2 & condition3 & condition4
```

Fig. 5 Legal states of atomic commitment.

Finally all processes has completed or stopped. At this stage, the processes that have not stopped (that is, c and p_1) have reached an Abort decision.

The coordinator process can be described by using the proposed input language as shown in Fig. 4. The variable `ph` represents the current phase of the process. The value of `ph` is 0 initially, 1 after the process has cast its vote and entered phase 2, and 2 after the process has reached a decision and completed phase 2. The variable `d` represents (depending upon the current phase) the vote or the decision of the process. The value of `d` is $true$ if the vote is Yes or the decision is Commit, and $false$ if the vote is No or the decision is Abort. The variable `up` represents the current status of the process. The value of `up` is $true$ if the process is being executed, and $false$ if the process is stopped. Other processes can be described similarly.

We assume that if the following four conditions are satisfied, the system is in the legal state. (1) If c has not voted ($c.ph = 0$), then each process has either not voted or (detected that c had stopped and) reached an Abort decision. (2) If c has voted but not reached a decision ($c.ph = 1$), then each process has either not reached a decision or (detected that c had stopped and) reached an Abort decision. (3) If c has reached a Commit decision ($c.ph = 2 \wedge c.d$), then each process has either voted Yes (and not reached a decision) or reached a Commit decision. (4) If c has reached an Abort decision ($c.ph = 2 \wedge \neg(c.d)$), then each process has either not reached a decision or reached an Abort decision. Thus the legal states can be described as shown in Fig. 5.

By applying the translation method to this example described above, we verified the fault tolerance by SMV. We applied the method to the systems where the number of processes were 3, 4, 5, and 6. When the number of processes was 6, the time required for verification was about 0.65 seconds and the number of reachable states was about 2^{25}. Figure 6 shows the output of SMV in case the number of processes was 6. The performance of verification is shown in Table 1.

```
% smv -r 2phase.smv
 -- specification AG (f -> AF S) is true
 resources used:
 user time:  0.65 s, system time:  0.03 s
 BDD nodes allocated:  38479
 Bytes allocated:  1900544
 BDD nodes representing transition relation:  9391 + 14
 reachable states:
 3.10518e+07(2^24.8882) out of 3.82206e+08(2^28.5098)
```

Fig. 6 Verification result produced by SMV (atomic commitment).

Table 1 Performance of verification.

Protocol (♯ of processes)	Time	States	
	(sec)	Reachable	Total
Atomic Commit (3)	0.03	5312	$\approx 2^{15}$
Atomic Commit (4)	0.08	91392	$\approx 2^{19}$
Atomic Commit (5)	0.21	$\approx 2^{20}$	$\approx 2^{24}$
Atomic Commit (6)	0.65	$\approx 2^{25}$	$\approx 2^{29}$
Leader Election (3)	0.04	11664	11664
Leader Election (4)	0.26	$\approx 2^{21}$	$\approx 2^{21}$
Leader Election (5)	2.27	$\approx 2^{29}$	$\approx 2^{29}$
Leader Election (6)	9.68	$\approx 2^{38}$	$\approx 2^{38}$
Byzantine Agreement(4)	315.89	$\approx 2^{25}$	$\approx 2^{81}$

5.2 Byzantine Agreement

The second example is the *Byzantine agreement problem* [2]. Each process is either Reliable or Unreliable. Each Reliable process reaches one of two decisions, *false* or *true*. One process g is distinguished and has associated with it a Boolean value B. It is required that:

1. If g is Reliable, the decision value of each Reliable process is B.
2. All Reliable processes reach the same decision.

We assume authenticated communication; messages sent by Reliable processes are correctly received by Reliable processes, and Unreliable processes cannot forge messages on behalf of Reliable processes [8], [21].

Agreement is reached within $N+1$ rounds of communication, where N is the maximum number of processes that can be Unreliable. In each round r, where $r \leqq N$, every Reliable process j that has not yet reached a decision of *true* checks whether g and at least $r-1$ other processes have reached a decision of *true*. If the check is successful, j reaches a decision of *true*. If j does not reach a decision of *true* in the first N rounds, it reaches a decision of *false* in round $N+1$.

Let d^r be a Boolean value denoting the tentative decisions of a process up to round r, and let $c^r.k$ be a Boolean value that is *true* iff the process knows that process k has reached a decision of *true* in round r. We assume that the system is in legal states when the following four conditions are satisfied. (1) The number of Unreliable processes is at most N. (2) Before the first round, the tentative decision of each Reliable process j is *false*, and for each k, $c^r.k$ of j is *false*. (3) In each round q, the tentative decision of each Reliable process j is set to *true* iff its previous tentative decision is *true* or j knows g and at least $q-1$ other processes have reached a decision of *true*, and $c^r.j$ of each other process k is set to *true* only if d^q of j is *true*. (4) In each round q, for any two Reliable processes j and k, if the current tentative decision of j is *false* then $c^q.k$ of j is *true* iff the previous tentative decision of k is *true* or some process knows k has reached a decision of *true*.

We can show that each computation of the protocol that starts at a state in the legal states satisfies the Byzantine agreement specification as follows. If the tentative decision of g before the first round was *true*, because the third and fourth conditions of legal states stated above hold, $c^1.g$ of each Reliable process becomes *true* and the decisions of the Reliable processes become *true* as well as g. If the tentative decision of g before the first round was *false*, because of the third condition, the decisions of the Reliable processes never change *true*. Thus the Reliable processes reach the same decision as process g.

Figure 7 illustrates how the protocol works. We assume that the number of processes is 4 and $N=1$. The program variables in round 0 (that is, the initial state) have the values as shown in Fig. 7(a). Now suppose a fault has occurred in the process p_2 and p_2 has become Unreliable.

In round 1, each process acts as follows. First, each process sets the values of $c^1.k$ ($k = g, p_1, p_2, p_3$). The value of $c^1.k$ is set to *true* when d^0 is *true* for process k or there exists a process such that $c^0.k$ is *true*. For example, for p_1 $c^1.g$ is *true*, $c^1.p_1$ is *false*, $c^1.p_2$ is *false* and $c^1.p_3$ is *false*, while each $c^1.k$ of Unreliable process p_2 is *false*.

Next, each process sets the value of d^1. The value of d^1 is set to *true* when d^0 is *true* or $c^1.g$ is *true*. For example, d^1 for p_1 is *true*, while d^1 for p_2 is *false*.

When round 1 has been completed, the program variables have the values as shown in Fig. 7(b).

In round 2, each process acts as follows. First, each process sets the values of $c^2.k$ similarly as in round 1. For example, for p_1 $c^2.g$ is *true*, $c^2.p_1$ is *true*, $c^2.p_2$ is *false* and $c^2.p_3$ is *true*, while each $c^2.k$ of Unreliable process p_2 is *false*.

Next, each process sets the value of d^2. The value of d^2 is set to *true* when d^1 is *true* or $c^2.g$ and at least one of ($c^2.p_1, c^2.p_2, c^2.p_3$) are *true*. For example, d^2 for p_1 is *true*, while d^2 for p_2 is *false*.

When round 2 has been completed, the program variables have the values as shown in Fig. 7(c), and each Reliable process reaches the same decision.

We described the Byzantine agreement problem as a program in the language that we proposed. Figure 8 describes process g. Here we consider the case which the number of processes is 4 and N is 1. The variables `d0,d1,d2` denote d^r for round $0, 1, 2$. The variables `c0k,c1k,c2k` denote $c^r.k$ for round $0, 1, 2$. The variables `b` is a Boolean value that is *true* iff the process

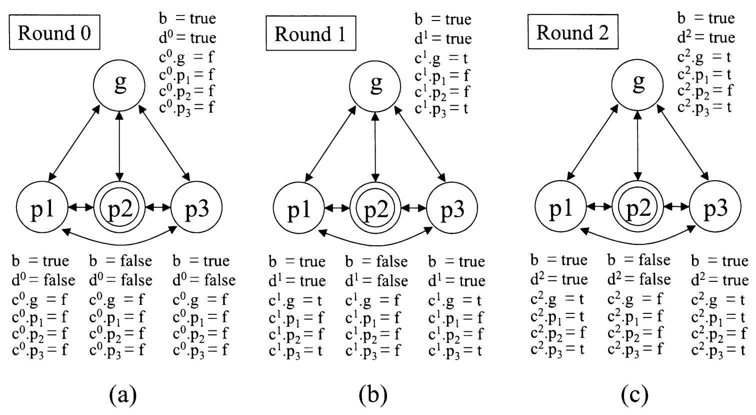


Fig. 7 Example of Byzantine agreement.

```
process g
 begin
 var   r,rr:{0,1,2}{0};   b:boolean{true};
       d0,d1,d2:boolean{true,false};
       c0g,c0p1,c0p2,c0p3:boolean{false};
       c1g,c1p1,c1p2,c1p3:boolean{false};
       c2g,c2p1,c2p2,c2p3:boolean{false};
 const csum1:=true;
       csum2:=(c2p1|c2p2|c2p3);
 action
   r=0 & rr=0 :> rr:=1,
        c1g:=d0 | (c0g|p1.c0g|p2.c0g|p3.c0g),
        c1p1:=p1.d0 | (c0p1|p1.c0p1|p2.c0p1|p3.c0p1),
        c1p2:=p2.d0 | (c0p2|p1.c0p2|p2.c0p2|p3.c0p2),
        c1p3:=p3.d0 | (c0p3|p1.c0p3|p2.c0p3|p3.c0p3);
   r=0 & rr=0 & !b & p1.b & p2.b & p3.b :> rr:=1,
     c1g:={true,false},c1p1:=false,
     c1p2:=false,c1p3:=false;
   r=0 & rr=0 & !b & !p1.b & p2.b & p3.b :> rr:=1,
     c1g:={true,false},c1p1:={true,false},
     c1p2:=false,c1p3:=false,
     ···
   r=0 & rr=1 & p1.rr=1 & p2.rr=1 & p3.rr=1
     :> d1:=d0|csum1&c1g,r:=1;
   r=0 & rr=1 & p1.rr=1 & p2.rr=1 & p3.rr=1 & !b
     :> d1:={true,false},r:=1;
     ···
 fault   b & p1.b & p2.b & p3.b :> b:=false;
 end
```

Fig. 8 Process g of Byzantine agreement.

```
const
   condition1 := g.b & p1.b & p2.b & p3.b
                | !g.b &  p1.b &  p2.b &  p3.b
                |  g.b & !p1.b &  p2.b &  p3.b
                |  g.b &  p1.b & !p2.b &  p3.b
                |  g.b &  p1.b &  p2.b & !p3.b;
   condition2 := (g.b -> (g.d0 = g.b) & !g.c0g)
                &(g.b -> (g.d0 = g.b) & !g.c0p1)
                &( ··· );
   condition3 :=
     ((g.r>=1 & g.rr>=1 & p1.r>=1 & p1.rr>=1
       & p2.r>=1 & p2.rr>=1 & p3.r>=1 & p3.rr>=1)
       -> ( (g.b -> (g.d1<>(g.d0|c1g&g.csum1)))
           &( ··· )
           &(g.b -> (g.c1g -> g.d1)&(p1.c1g -> g.d1)
                   &(p2.c1g -> g.d1)&(p3.c1g -> g.d1))
           &( ··· )))
    &( ··· );
   condition4 :=
    ((g.r>=1 & g.rr>=1 & p1.r>=1 & p1.rr>=1
       & p2.r>=1 & p2.rr>=1 & p3.r>=1 & p3.rr>=1)
       -> ( (g.b & g.b & !g.d0
             -> (g.c1g<>
                (g.d0|g.c0g|p1.c0g|p2.c0g|p3.c0g)))
           &(g.b & p1.b & !g.d0
             -> (g.c1p1<>
                (p1.d0|g.c0p1|p1.c0p1|p2.c0p1|p3.c0p1)))
           &( ··· ))
   &( ··· );
 spec condition1 & ··· & condition4
```

Fig. 9 Legal states of Byzantine agreement.

is Reliable. The variables `r` and `rr` denote the current round. If `rr` is 1, then it means that the current round is 1 and that $c^1.k$ of each k has been set to some value. Similarly when `r` is 1, the current round is 1 and d^1 has been set to some value. The variables `csum1` and `csum2` denote whether the process knows that g and at least $q-1$ other processes have reached a decision of *true* for $q=1$ and 2 respectively. Other processes can be described similarly. The legal states are described as shown in Fig. 9.

The time required for verification was about 316 seconds and the number of reachable states was about 2^{25}. The performance of verification is shown in Table 1.

5.3 Leader Election

The third example is the *leader election problem* on rings. The leader election problem is the problem of selecting one process as a leader on a ring where no distinguished process initially exists. This problem originally arose in the study of *token ring* networks. In such a network, a single "token" circulates around the network. Sometimes, however, the token may be lost due to faults, and it becomes necessary for the processes to execute an algorithm to regenerate the lost token. This regeneration procedure amounts to electing a leader. We consider a ring consisting of N processes, $p_0, p_1, \cdots, p_{N-1}$, that are connected in this order. The process p_{i-1} is said to be a *predecessor* of the process p_i in the ring. The processes are assumed to have unique ids. The id for process p_i is denoted by id_i.

Here we consider a leader election algorithm proposed in [16]. In the algorithm, the process with the maximum id is selected as the leader. Each process p_i

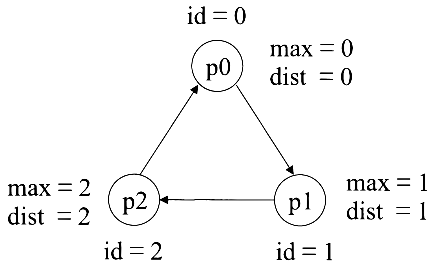


Fig. 10 Example of leader election.

has two variables, max_i and $dist_i$. max_i means the maximum id the process i knows, and $dist_i$ means the distance to the process p_j where id_j is max_i.

Each process p_i has the following three actions:

1. If id_i is larger than max_i, max_i is set to id_i and $dist_i$ is set to 0. While if $dist_i$ is 0 and max_i is not equal to id_i, max_i is set to id_i. And if max_i is equal to id_i and $dist_i \neq 0$, $dist_i$ is set to 0.
2. If $dist_{i-1}+1 < N$ and id_i is smaller than max_{i-1}, max_i is set to max_{i-1} and $dist_i$ is set to $dist_{i-1}+1$.
3. If $dist_{i-1}+1 \geqq N$, or if id_i is larger than the id_{i-1} and id_i is equal to or larger than max_{i-1}, max_i is set to id_i and $dist_i$ is set to 0.

Let K be the maximum id of any process in the ring. The leader is successfully elected if the system reaches the state that satisfies the following conditions.

1. For all processes i, $max_i = K$.
2. If j is the process with id K, then $dist_j = 0$. For any other process $i \neq j$, $dist_i = 1 + dist_{(i-1)modN}$.

Since each process p_i only have the two variables, max_i and $dist_i$, there is exactly one such state. Clearly, this state is the only legal state. We consider transient faults. A fault changes the values of the variables of a process arbitrarily.

Using Fig. 10, we explain the protocol. We assume that the number of processes is 3 ($K = 2$) and that some faults have occurred at the initial state and the program variables have the values as shown in Fig. 10.

As an example, suppose that p_1, p_2, p_0, and p_1 are selected to be executed in this order. First, the process p_1 executes the first action and sets *max* to 1 and *dist* to 0. Next the process p_2 also executes the first action and sets *max* to 2 and *dist* to 0. Then the process p_0 executes the second action and sets *max* to 2 and *dist* to 1. Finally the process p_1 executes the second action and sets *max* to 2 and *dist* to 2 and the system has reached the legal state.

When N is 4, each process is described as shown in Fig. 11. The variable `max` denotes max_i. Similarly, the variable `dist` denotes $dist_i$. The legal state is described as shown in Fig. 12.

We apply the method to the systems where $N = 3, 4, 5, 6$. In case $N = 6$, the time requires for verification was about 9.68 seconds and the number of reachable states was about 2^{21}. The performance of verification is shown in Table 1.

```
process p1
 begin
 var
   max:{0,1,2,3}{3};
   dist:{0,1,2,3}{2};
 const
   id := 1;
 action
   (id>max) | (id != max & dist=0)
   | (id=max & dist != 0)
   :> max:=id,dist:=0;
   (p0.dist+1<N) & (id<p0.max)
   & !(max=p0.max & dist=p0.dist+1)
   :> max:=p0.max,dist:=p0.dist+1;
   ((p0.dist+1>=N) | ((id>p0.id) & (id>=p0.max)))
   & !(max=id & dist=0)
   :> max:=id,dist:=0;
 fault
   true :> max:={0,1,2,3},dist:={0,1,2,3};
 end
```

Fig. 11 A process of leader election.

```
const
   condition1 := p0.max=K & ··· & p3.max=K;
   condition2
    :=  (p0.id=K -> p0.max=K & p1.dist=1+p0.dist
          & p2.dist=1+p1.dist & p3.dist=1+p2.dist)
       & ···
       &(p3.id=K -> p3.max=K & p0.dist=1+p3.dist
          & p1.dist=1+p0.dist & p2.dist=1+p1.dist);
 spec
   condition1 & condition2
```

Fig. 12 Legal states of leader election.

Table 2 Performance of verification of the closure property.

Protocol (♯ of processes)	Time	States	
	(sec)	Reachable	Total
Atomic Commit (3)	0.01	552	13824
Atomic Commit (4)	0.03	4432	$\approx 2^{18}$
Atomic Commit (5)	0.11	37920	$\approx 2^{23}$
Atomic Commit (6)	0.25	$\approx 2^{18}$	$\approx 2^{27}$
Leader Election (3)	0.01	8	5832
Leader Election (4)	0.01	16	$\approx 2^{20}$
Leader Election (5)	0.03	32	$\approx 2^{28}$
Leader Election (6)	0.05	64	$\approx 2^{37}$
Byzantine Agreement(4)	59.52	$\approx 2^{20}$	$\approx 2^{80}$

To our knowledge, there is no other research that can be directly compared to ours; however, since the time required for verification was only approximately 5 minutes even for the largest example, we think that the proposed verification method is practical, at least for systems with small number of processes. From our experience, design errors may often be observed even when the number of processes is rather few [22]. Thus we think that the proposed method is useful especially in early stages of system development.

5.4 Other Results

Closure property As stated in 4.2.3 we can check the closure property of the system by extending the proposed method. We checked the closure property for the three examples. Table 2 shows the performance of the

Table 3 Quantity of description.

Protocol (♯ of processes)	♯ of tokens	
	Proposed language	SMV
Atomic Commit (3)	771	2500
Atomic Commit (4)	1107	4499
Atomic Commit (5)	1459	7445
Atomic Commit (6)	1875	11440
Leader Election (3)	613	1768
Leader Election (4)	845	2662
Leader Election (5)	1095	3826
Leader Election (6)	1363	5200
Byzantine Agreement(4)	9575	77315

verification of the closure property. Since this property can be checked without considering faults, the state space to be explored is significantly smaller than the case of fault tolerance verification.

Length of programs As stated before, we developed the modeling language and its translation method to facilitate describing the system to be verified. To support our claim, we compared the length of the program described in the proposed language and the resulting SMV program. Table 3 compares both programs in terms of the total number of tokens encountered in the parsing process. This result clearly shows that using the proposed language significantly reduced the quantity of description.

6. Conclusions

In this paper, we proposed a formal method for verification of fault tolerance of concurrent systems. We use a model checking method to carry out the verification automatically. Differing from other related work, which is tailored to specific systems, we are aimed at providing a single approach that can be applied to various systems. Specifically, we proposed a method that can deal with any system if it is given as a guarded command program based on the model proposed in [2].

We designed this method so that it can use a symbolic model checking tool called SMV, which can avoid the state explosion problem. Automatic verification of fault tolerance is performed by translating the program to the SMV language. For this purpose, we first proposed a modeling language suited for describing fault-tolerant systems in the form of guarded command programs. We then proposed a translation method from the modeling language to the input language of SMV.

In the case studies, we demonstrated that various fault-tolerant systems can be automatically verified by the proposed method. The results showed that the verification was completed with practical time.

References

[1] A. Arora, A Foundation of Fault-Tolerant Computing, Ph.D. Dissertation, The University of Texas, Austin, 1992.

[2] A. Arora and M. Gouda, "Closure and convergence: A foundation of fault-tolerant computing," IEEE Trans. Software Engineering, vol.19, no.11, pp.1015–1027, Nov. 1993.

[3] A. Arora and S. Kulkarni, "Designing masking fault-tolerance via nonmasking fault-tolerance," IEEE Trans. Software Engineering, vol.24, no.6, pp.435–450, 1998.

[4] C. Bernardeschi, A. Fantechi, and L. Simoncini, "Formally verifying fault tolerant system designs," The Computer Journal, vol.43, no.3, pp.191–205, 2000.

[5] P. Bernstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Addison-Wesley, Reading, MA, 1987.

[6] R.E. Bryant, "Graph-based algorithms for Boolean function manipulation," IEEE Trans. Comput., vol.C-35, no.8, pp.677–691, 1985.

[7] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hawng, "Symbolic model checking: 10^{20} states and beyond," Information and Computation, vol.98, no.2, pp.142–170, 1992.

[8] K. Chandy and J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, Reading, MA, 1988.

[9] E.M. Clarke, E.A. Emerson, and A.P. Sistla, "Automatic verification of finite-state concurrent systems using temporal-logic specifications," ACM Trans. Programming Languages and Systems, vol.8, no.2, pp.244–263, 1986.

[10] E.M. Clarke, O. Grumberg, and D.A. Peled, Model Checking, MIT Press, 1999.

[11] F. Cristian, "A rigorous approach to fault-tolerant programming," IEEE Trans. Software Engineering, vol.11, no.1, pp.23–31, 1985.

[12] F.C. Gärtner, "Fundamentals of fault-tolerant distributed computing in asynchronous environments," ACM Computing Surveys, vol.31, no.1, pp.1–26, March 1999.

[13] F.C. Gärtner, "Specifications for fault tolerance: A comedy of failures," Technical Report TUD-BS-1998-03, Darmstadt University of Technology, Germany, 1998.

[14] S. Gnesi, G. Lenzini, D. Latella, C. Abbaneo, A. Amendola, and P. Marmo, "An automatic SPIN validation of a safety critical railway control system," The International Conference on Dependable Systems and Networks (DSN 2000), pp.119–124, IEEE, 2000.

[15] J. Kljaich, Jr., B.T. Smith, and A.S. Wojcik, "Formal verification of fault tolerance using theorem-proving techniques," IEEE Trans. Comput., vol.38, no.3, pp.366–376, March 1989.

[16] X. Lin and S. Ghosh, "Maxima finding in a ring," Proc. 28th Ann. Allerton Conf. on Computers, Communication, and Control, pp.662–671, 1991.

[17] Z. Liu and M. Joseph, "Verification of fault tolerance and real time," Proc. 26th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-26), pp.220–229, IEEE, June 1996.

[18] K.L. McMillan, Symbolic Model Checking, Kluwer Academic Publishers, Boston, 1993.

[19] P.M. Melliar-Smith and R.L. Schwartz, "Formal specification and mechanical verification of SIFT: A fault-tolerant flight control system," IEEE Trans. Comput., vol.C-31, no.7, pp.616–630, July 1982.

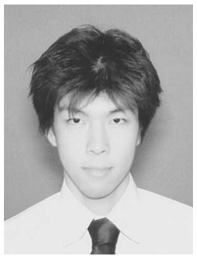
[20] F. Schneider, S.M. Easterbrook, J.R. Callahan, and G.J. Holzmann, "Validating requirements for fault tolerant systems using model checking," Proc. International Conference on Requirements Engineering, ICRE, pp.4–14, IEEE, April 1998.

[21] T.K. Srikanth and S. Toueg, "Simulating authenticated broadcast to derive simple fault tolerant algorithms," Distrib. Computing, vol.2, no.2, pp.80–94, 1987.

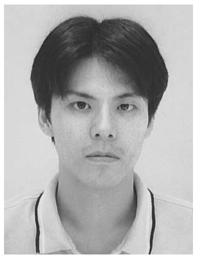
[22] T. Tsuchiya, S. Nagano, R.B. Paidi, and T. Kikuno, "Symbolic model checking for self-stabilizing algorithms," IEEE Trans. Parallel and Distributed Systems, vol.12, no.1,

pp.81–95, 2001.

[23] H. Völzer, "Verifying fault tolerance of distributed algorithms formally: An example," Proc. International Conference on Application of Concurrency to System Design (CSD98), pp.187–197, IEEE, March 1998.



Tomoyuki Yokogawa was born in 1977. He received the M.E. degree from Osaka University in 2000. He is currently studying towards the Ph.D. degree in the Graduate School of Engineering Science at the same university. He has been engaged in research on automatic verification.



Tatsuhiro Tsuchiya was born in 1972. He received the M.E. and Ph.D. degrees in computer science from Osaka University in 1995 and 1998, respectively. He is currently an associate professor in the Department of Information Systems Engineering at Osaka University. His research interests are in the areas of distributed fault-tolerant systems. He is a member of IEEE.

Tohru Kikuno was born in 1947. He received the M.S. and Ph.D. degrees from Osaka University in 1972 and 1975, respectively. He was with Hiroshima University from 1975 to 1987. Since 1990, he has been a Professor in the Department of Information Systems Engineering at Osaka University. His research interests include the quantitative evaluation of software development process, the analysis and design of fault-tolerant systems and the design of procedures for testing communication protocols. He served as a program co-chair of the 1st International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'98) and 5th International Conference on Real-Time Computing Systems and Applications (RTCSA'98). He is a member of IEEE and ACM.