OUKA
The University of Osaka
Institutional Knowledge Archive

| | |
|---|---|
| Title | Formal Verification for Dependable Systems by Model Checking |
| Author(s) | 横川, 智教 |
| Citation | 大阪大学, 2004, 博士論文 |
| Version Type | VoR |
| URL | https://hdl.handle.net/11094/1455 |
| rights | |
| Note | |

*The University of Osaka Institutional Knowledge Archive : OUKA*

https://ir.library.osaka-u.ac.jp/

The University of Osaka

# Formal Verification for Dependable System by Model Checking

Tomoyuki Yokogawa

December 2003

# Contents

# Chapter 1

# Introduction

## 1.1 Overview

In recent years, the growing demand for high availability and reliability of computer systems has led to a formal verification for dependable systems. Among them, there are a number of researches of formal verification for fault-tolerant systems [18, 23, 27, 30]. 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 [23, 30]. Kljaich proposed a formal verification system based on the use of automated reasoning techniques to validate fault tolerance [23]. An extend Petri net representation, called flow nets, is used to describe the system to be verified. Melliar-Smith showed the methodology employed to demonstrated rigorously that the SIFT fault-tolerant computer meets its requirements [30]. The process of formal specification and verification of SIFT discovered four design errors which would have been difficult or impossible to detect by testing. 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. For realistic designs, however, the number of states of the system can be very large and the explicit traversal of the state space may become infeasible. This problem is usually called the state explosion problem.

*Symbolic model checking* [29] is one of the most successful approaches to state explosion. This method alleviates the problem by symbolically representing the state space by Boolean functions. Many symbolic model checking tools use *Binary Decision Diagrams* (BDDs) as the data structure to manipulate Boolean functions efficiently. Since Boolean functions can often be represented by BDDs very compactly, the symbolic model checking method can reduce the memory and time required for analysis.

There are also some examples where fault tolerance property of concurrent systems was verified by model checking [5, 19, 37]. Bernardeschi presented an approach for the verification of the correctness of fault tolerant system [5]. The approach is based on process algebras, equivalence theory and temporal logic. The usability of the approach is supported by the availability of automatic tools for equivalence checking and for proving the temporal logic properties by model checking. Gnesi described an experiment in formal specification and verification performed in the context of a safety critical railway control system [19]. In this research, verification of safety and liveness properties had been performed using the verification tool suite SPIN. Schneider showed a practical application of model checking for validating the requirements for a complex embedded system [37]. The system verified in the case study is a dually redundant spacecraft controller, in which a checkpoint and rollback scheme is used to provide fault tolerance during the execution of critical control sequences.

In this dissertation, we aim to achieve formal verification for dependable systems using model checking. There are a number of methods to verify dependable systems using modelchecking. However, these methods for verification are specialized for specific systems, and a general approach does not exist. Thus we propose a general method

for automatic verification for dependable systems using model checking. We propose general approaches for automatic verification using model checking for two problems. The first is verification for fault-tolerant systems, and the second is detection of feature interaction. First, we propose a method to verify fault-tolerant systems, which is the common dependable system, using model checking. We achieve verification for fault tolerance using symbolic model checking by providing the framework to model fault-tolerant systems and extracting transition relations as a Boolean formula from the model. Second, we propose a method to detect *feature interactions* in telecommunication services using model checking. *Feature interaction* refers to situations where a combination of different services induce unexpected behaviors. We adopt a variant of State Transition Rules (STR) [21, 34] to describe services and the behavior of the system. And we detect feature interactions using symbolic model checking by extracting transition relations from the model in a similar way. These methods use a symbolic model checking tool called SMV (Symbolic Model Verifier) [29]. 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) [12].

First, we propose a method to verify fault-tolerant systems automatically 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.

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.

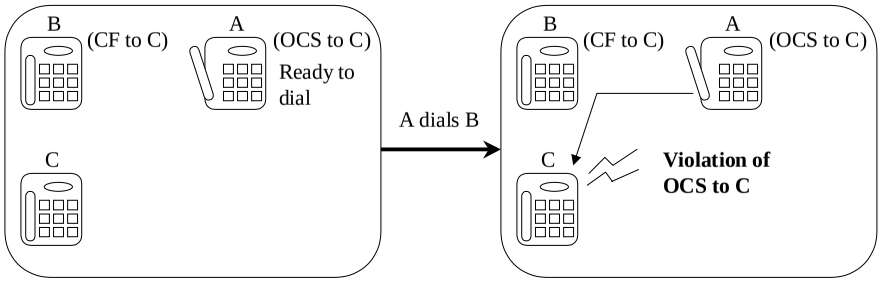


Figure 1.1: Interaction example.

Second we propose a method to detect *feature interactions* in telecommunication services using model checking. *Feature interaction* refers to situations where a combination of different services behaves differently than expected from the single services' behaviors. For example, consider a situation where user A has subscribed to the service *Originating Call Screening* (OCS) and does not want calls to user C to be put through, and user B has activated the service *Call Forwarding* (CF) to user C. In this situation, if A calls B, the intention of OCS not to be connected to C is invalidated since the call is put through to C by way of B. Figure 1.1 illustrate this interaction.

In today's intelligent telecommunication networks, feature interaction is considered a major obstacle to the introduction of new features and the provision of reliable services. In practical service development, however, the analysis of interactions has often been conducted in an ad hoc manner. This leads to time-consuming service design and testing without any interaction-free guarantee.

Many approaches have been explored to overcome this situation. Keck and Kuehn surveyed these approaches [24]. They also surveyed classification schemes, and mentioned the one by Bouma and Velthuijsen [8] as a generally accepted categorization framework. In this framework, the approaches to feature interaction are classified into on-line and off-line and into avoidance, detection, and resolution.

We propose a formal approach which falls into the off-line detection category; that is, the proposed approach is aimed at detecting latent feature interaction in given communication service specifications. Although formal approaches have been well studied in this category, ours is different in that it uses *model checking*.

In our method, we achieve to detect feature interaction by the SMV tool by following

processes. First we extract the transition relation from given specification as a boolean formula and describe SMV program using the formula. And second we describe the property of interaction occurrence as a CTL formula. As experiments, we demonstrate that we can detect all interactions for given service specifications by SMV.

As stated above, we achieved automatic detection for feature interaction using symbolic model checking. However, there are some case where SMV requires much time for detection process. To solve this problem, we propose the method to detect feature interaction using *bounded model checking*.

*Bounded model checking* [7, 39, 38] is a new symbolic model checking method which does not use BDDs. The central idea behind this method is to reduce the model checking problem to the propositional satisfiability (SAT) checking problem and to look for counterexamples that are shorter than some fixed length $k$ for a given property. The formula to be checked is constructed by unwinding the transition relation of the system $k$ times such that truth assignments satisfying the formula correspond to counterexamples.

In the literature, it has been reported that bounded model checking can work efficiently, especially for the verification of digital circuits. An advantage of this method is that it works efficiently even when compact BDD representation cannot be obtained. It is also an advantage that it can exploit recent advances in decision procedures of satisfiability. (The latest SAT tools include, for example, Grasp [28] and Chaff [31].)

In contrast, this method does not work well for asynchronous systems, because the encoding scheme into propositional formulas is not suited for such systems. When applying this technique to asynchronous systems, a large formula would be required to represent the transition relation, thus resulting in large execution time and low scalability.

To overcome this problem we have been working on new encoding. In this dissertation we describe the encoding scheme. (Preliminary results were presented as a short paper [42].) The new encoding reduces the size of the resultant formula by exploiting the property that usually only small fraction of state variables take in part of each state transition. Interestingly, as a side-effect, the new scheme often explores a larger state space than the existing bounded model checking does for the same $k$. By applying the proposed scheme and other model checking methods to feature interaction detection, we show the effectiveness of the propose method.

The remainder of this dissertation is organized as follows. In Chapter2, we describe model checking method. In Chapter3, we describe the method for automatic verification for fault-tolerant systems and show the result of case studies. In Chapter4, we describe the method to detect feature interactions and show the experimental reslts. In Chapter5, we propose to use bounded model checking for feature interaction detection and show its effectiveness. In Chapter6, we conclude the dissertation.

# Chapter 2

# Model Checking

## 2.1 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 [13]. 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.

### 2.1.1 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* [10, 29].

In symbolic model checking, a Boolean function is expressed by using OBDDs (Ordered binary decision diagrams)[9]. 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 [12]. 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 $AGf$.

### 2.1.2 Symbolic Model Verifier (SMV)

SMV is a software tool that implements symbolic model checking [29]. 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 2.1 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

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

Figure 2.1: SMV program

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.

# Chapter 3

# Fault Tolerance Verification

## 3.1 Model of Fault-Tolerant Systems

### 3.1.1 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.

### 3.1.2 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 [14]. 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, 43, 17].

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 $\{val_1, \cdots, val_n\}(1 \leq k \leq n)$.

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.

### 3.1.3 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 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 3.1 illustrates this concept.

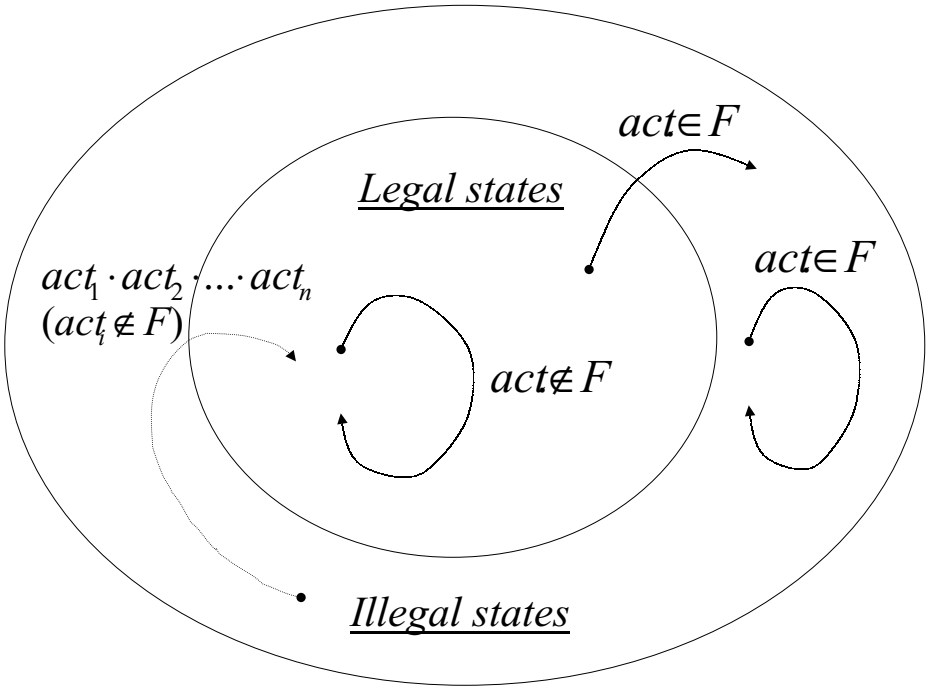


Figure 3.1: Schematic overview of the fault tolerance property

## 3.2 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.

### 3.2.1 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 $\{1..5\}$ 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.

### 3.2.2 Translation Method to the SMV Language

#### 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 $A = a_1 \vee a_2 \vee \cdots \vee a_N$.

### 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.

**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.

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

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.

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

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 nonmasking 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 $AG(S \rightarrow AG(S))$, which represents the closure property, is checked by the SMV tool.

## 3.3 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 500MHz Pentium III processor and 256 Mbytes of memory.

### 3.3.1 Atomic Commitment Protocol

The first example is the *atomic commitment protocol* [2, 6]. 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

```
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
```

Figure 3.2: The coordinator process.

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).

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).

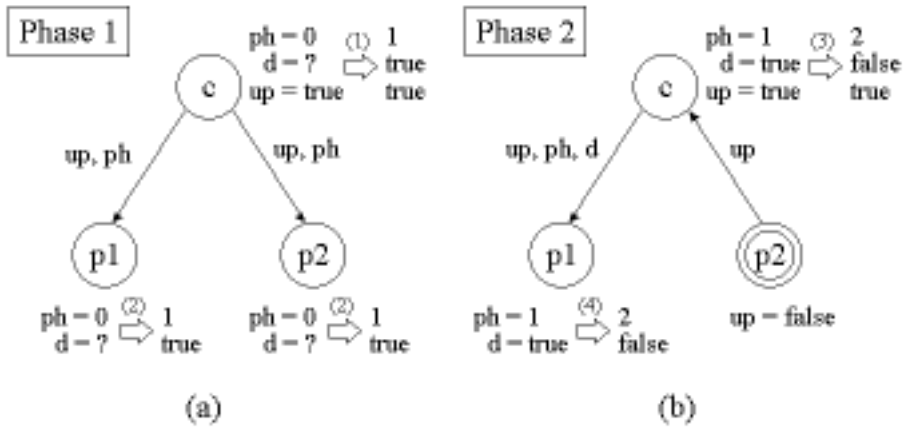


Figure 3.3: Example of atomic commitment.

Using Figure 3.3, 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.

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 Figure 3.2. 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

```
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
```

Figure 3.4: Legal states of atomic commitment.

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 Figure 3.4.

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

```
% 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)
```

Figure 3.5: Verification result produced by SMV (atomic commitment).

$2^{25}$. Figure 3.5 shows the output of SMV in case the number of processes was 6. The performance of verification is shown in Table 3.1.

### 3.3.2 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 [11, 40].

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 $r \leq N$, 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 $g$.

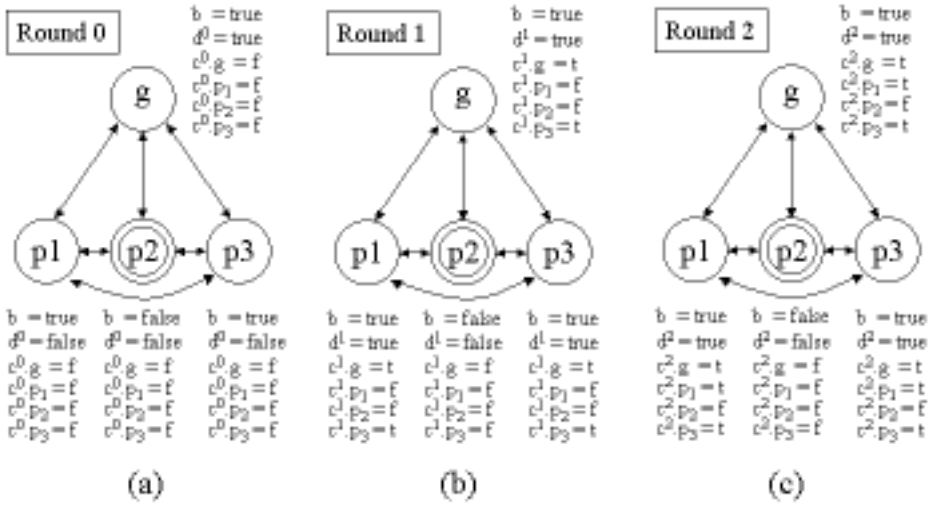


Figure 3.6: Example of Byzantine agreement.

Figure 3.6 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 Figure 3.6 (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 Figure 3.6 (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 Figure 3.6 (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 3.7 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 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 Figure 3.8.

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 3.1.

### 3.3.3 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 [26]. In the algorithm, the process with the maximum id is selected as the leader. Each process $p_i$ 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 $dist_{i-1} + 1 \geq N$, 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 $i \neq j$, $dist_i = 1 + dist_{(i-1) mod N}$.

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 Figure 3.9, 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 Figure 3.9.

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 Figure 3.10. The variable `max` denotes $max_i$. Similarly, the variable `dist` denotes $dist_i$. The legal state is described as shown in Figure 3.11.

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 3.1.

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 [41]. Thus we think that the proposed method is useful especially in early stages of system development.

### 3.3.4 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 3.2 shows the performance of the verification of the closure property. Since this

Table 3.1: Performance of verification.

| Protocol (♯ of processes) | Time (sec) | States | |
|---|---|---|---|
| | | 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 | $\approx 2^{25}$ | $\approx 2^{81}$ |

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 3.3 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.

Table 3.2: 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 | $\approx 2^{20}$ | $\approx 2^{80}$ |

Table 3.3: 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 |

```
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
```

Figure 3.7: 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
```

Figure 3.8: Legal states of Byzantine agreement.

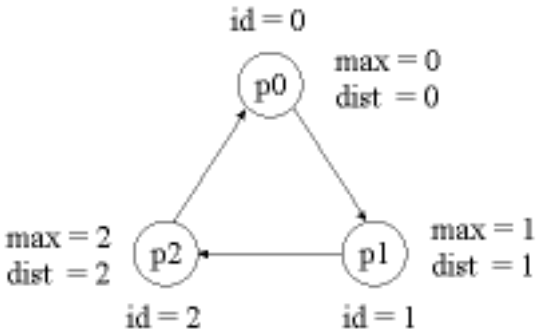


Figure 3.9: Example of leader election.

```
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
```

Figure 3.10: 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
```

Figure 3.11: Legal states of leader election.

# Chapter 4

# Feature Interaction Detection

## 4.1 Services and Interaction

### 4.1.1 Communication Services

From ITU-T recommendation [22] (*ITU-T Recommendations Q.1200 Series* - Intelligent Network Capability Set 1 (CS1)) and Bellcore's feature standards [4] (*Bellcore* - LSSGR Features Common to Residence and Business Customers I, II, III), we selected the following seven services (features) to consider:

**Call Waiting (CW):** This service allows the subscriber to receive a second incoming call while he or she is already talking. Suppose that $x$ subscribes to CW. Even when $x$ is busy taking with $y$, $x$ can receive a call from a third party $z$.

**Call Forwarding (CF):** This service allows the subscriber to have his or her incoming calls forwarded to another address. Suppose that $x$ subscribes to CF and that $x$ specifies $y$ to be a forwarding address. Then, any incoming call to $x$ is automatically forwarded to $y$.

**Originating Call Screening (OCS):** This service allows the subscriber to specify that outgoing calls be either restricted or allowed according to a screening list. Suppose that $x$ subscribes OCS and that $x$ puts $y$ in the OCS screening list. Then, any outgoing call to $y$ from $x$ is restricted, while any other call from $x$ is allowed. Suppose that $x$ receives dialtone. At this time, even if $x$ dials $y$, $x$ receives busytone instead of calling $y$.

**Terminating Call Screening (TCS):** This service allows the subscriber to specify that incoming calls be either restricted or allowed according to a screening list. Suppose that $x$ subscribes TCS and that $x$ puts $y$ in the TCS screening list. Then, any incoming call from $y$ to $x$ is restricted, while any other call to $x$ is allowed. Suppose that $y$ receives dialtone. At this time, even if $y$ dials $x$, $y$ receives busytone instead of calling $x$.

**Denied Origination (DO):** This service allows subscriber to disable any call originating from the terminal. Only terminating calls are permitted. Suppose that $x$ subscribes to DO. Then, any outgoing call from $x$ is restricted. Even if $x$ offhooks when the terminal is idle, $x$ receives busytone instead of dialtone.

**Denied Termination (DT):** This service allows subscriber to disable any call terminating at the terminal. Only originating calls are permitted. Suppose that $x$ subscribes to DT. Then, any incoming call to $x$ is restricted. Even if another user $y$ dials $x$, $y$ receives busytone without calling $x$.

**Direct Connect (DC):** This service is a so-called *hot line* service. Suppose that $x$ subscribes to DC and that $x$ specifies $y$ as the destination address. Then, by only offhooking, $x$ is directly calling $y$. It is not necessary for $x$ to dial $y$.

### 4.1.2 Feature Interaction

In this paper we consider two types of feature interaction. As shown below, the properties of the absence of these types of interaction can be viewed as safety properties, and hence detecting these types of interactions involves checking reachability from the initial state to undesirable states.

#### Nondeterminism

The first type we consider is *nondeterminism*. Nondeterminism is one of the best known types of feature interactions [15, 16, 20, 25, 33, 34]. Nondeterminism refers to a situation where an event can simultaneously activate two or more functionalities of different services, and as a result, it cannot be determined exactly which functionality should be activated.

It is known that this type of interaction occurs between CW and CF. Suppose that $A$ subscribes both services. Now consider the situation where (1) A is taking with B, (2) C is ready to dial, and (3) D is in A's forwarding address list and is idle. In this situation, if C dials A, then either the call from C to A may be received by A because of A's CW feature, or it may be forwarded to D by the CF feature.

This type of interaction can be detected by checking reachability from the initial state to the states that cause nondeterminism. We call such states *nondeterministic states*.

#### Invariant Violation

The next type of interaction we consider is *invariant violation*. It is usually the case that services require for some specific properties to be satisfied at any time. For example, for OCS service, the service designer may describe that "If $x$ specifies $y$ in the screening list, then $x$ is never calling $y$ at any time". Such a property is generally referred to as an invariant property. It is known that combining multiple services can result in violation of invariant properties. The OCS plus CF example described in the first section falls in this type.

This type of feature interaction can also be detected by checking reachability from the initial state to the undesirable states where the invariant properties are violated.

## 4.2 Model

In this paper we adopt a variant of State Transition Rules (STR) [21, 34] to describe services and model the behavior of the system in a rigorous fashion.

### 4.2.1 Notation

A service specification is defined as 6-tuple $\langle U, V, P, E, R, s_{init} \rangle$, where $U$ is a set of constants representing service users, $V$ is a set of variables, $P$ is a set of predicates, $E$ is a set of events, $R$ is a set of rules, and $s_{init}$ is the (*initial*) state. Each rule $r \in R$ is defined as follows:

$$r : pre{-}condition\ [event]\ post{-}condition.$$

$U = \{A, B\}$

$V = \{x, y\}$

$P = \{idle(x), dialtone(x), busytone(x), calling(x, y), path(x, y)\}$

$E = \{onhook(x), offhook(x), dial(x, y)\}$

$R = \{$

$pots1 : idle(x)\ [offhook(x)]\ dialtone(x).$

$pots2 : dialtone(x)\ [onhook(x)]\ idle(x).$

$pots3 : dialtone(x), idle(y)\ [dial(x, y)]\ calling(x, y).$

$pots4 : dialtone(x), \neg idle(y)\ [dial(x, y)]\ busytone(x).$

$pots5 : calling(x, y)\ [onhook(x)]\ idle(x), idle(y).$

$pots6 : calling(x, y)\ [offhook(y)]\ path(x, y), path(y, x).$

$pots7 : path(x, y), path(y, x)\ [onhook(x)]\ idle(x), busytone(y).$

$pots8 : busytone(x)\ [onhook(x)]\ idle(x).$

$pots9 : dialtone(x)\ [dial(x, x)]\ busytone(x).$

$\}$

$s_{init} = \{idle(A), idle(B)\}$

Figure 4.1: Rule-based specification for POTS.

A *predicate* is of the form $p(x_1, \ldots, x_k)$ where $p \in P$ and $x_i \in V$. *Pre-condition* consists of predicates or negations of predicates, or both, while *Post-condition* consists of predicates only. An *event* is of the form $e(x_1, \ldots, x_k)$, where $e \in E$ and $x_i \in V$.

Figure 4.1 shows an example of a specification. This specification describes the Plain Old Telephone Service (POTS). Additional communication features, such as those described in the previous subsection, can be described by modifying this specification (for example, adding rules or predicate symbols). Specifications for the above services are shown in [32]. In all these specifications, it is assumed that at the initial state, all users are idle and no user subscribes to any service yet.

### 4.2.2 State Transition Model

Here we define the state transition system specified by the rule-based specification. Let $\langle U, V, P, E, R, s_{init}\rangle$ be a service specification. For $r \in R$, let $x_1, \ldots, x_n$ $(x_i \in V)$ be variables appearing in $r$, and let $\theta = \langle x_1|a_1, \ldots, x_n|a_n\rangle$ $(a_i \in U, a_i \neq a_j (i \neq j))$ be a

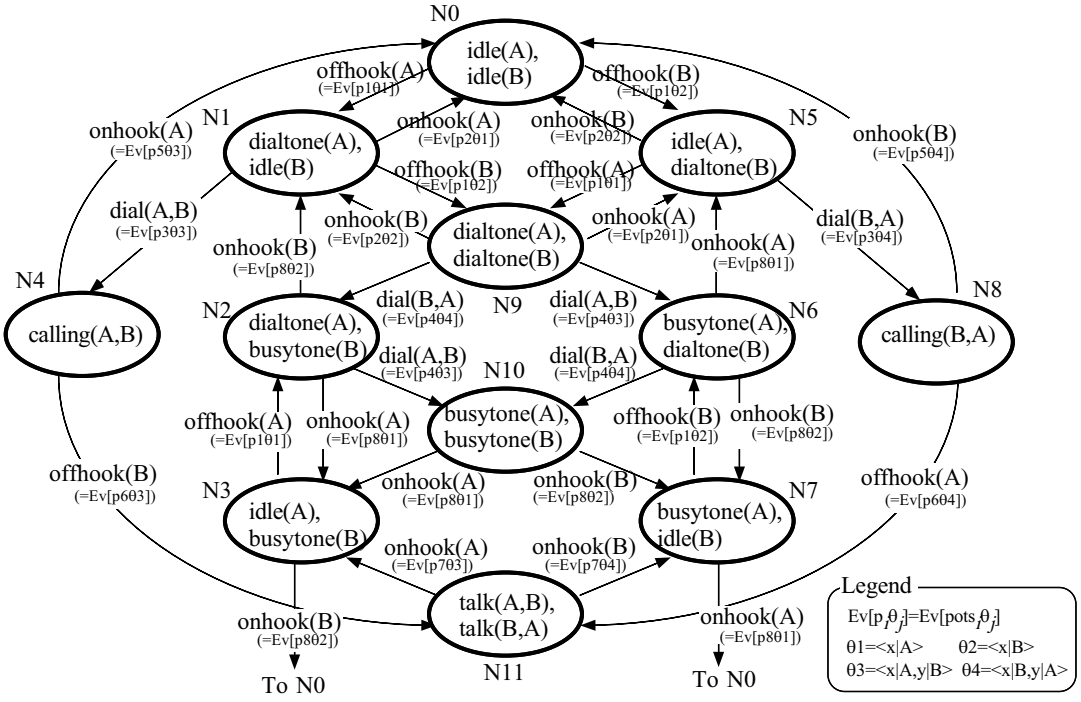


Figure 4.2: State transition diagram.

substitution replacing each $x_i$ in $r$ with $a_i$. Then, an *instance* of $r$ based on $\theta$ (denoted by $r\theta$) is defined as a rule obtained from $r$ by applying $\theta = \langle x_1|a_1, \ldots, x_n|a_n\rangle$ to $r$. We represent the event and the post-condition of an instance $r\theta$ of a rule as $e[r\theta]$ and $Post[r\theta]$, respectively. In addition, we denote by $Pre[r\theta]$ and $\hat{P}re[r\theta]$ the set of predicates in the pre-condition and the set of predicates whose negations are in the pre-conditions. Hence the precondition of an instance $r\theta$ of a rule is $Pre[r\theta] \cup \{\neg p \mid p \in \hat{P}re[r\theta]\}$.

A *state* is defined as a set of *instances of predicates* $p(a_1, \ldots, a_k)$ where $p \in P$ and $a_i \in U$. We think of each state as representing those that hold in the state.

Let $s$ be a state. We say that an instance of rule, $r\theta$, is *enabled* for $e(r\theta)$ at $s$ iff all instances in $Pre[r\theta]$ hold and no instances in $\hat{P}re[r\theta]$ hold at $s$. The execution of the enabled rule causes the *next state* $s'$ of $s$ by deleting all instances in $Pre[r\theta]$ from $s$ and adding all instances in $Post[r\theta]$ to $s$; that is,

$$s' = (s \backslash Pre[r\theta]) \cup Post[r\theta]$$

For example, suppose that $r = pots4$ in Figure 4.1, $\theta = \langle x|A, y|B\rangle$, and $s = \{dialtone(A), dialtone(B)\}$. Then $Pre[r\theta] = \{dialtone(A)\}$, $\hat{P}re[r\theta] = \{idle(B)\}$, $Post[r\theta] = \{busytone(A)\}$, and rule $pots4$ with substitution $\theta$ is enabled for event $dial(A, B)$. If subscriber $A$

dials $B$, that is, this event happens, then a state transition occurs, resulting in $s' = \{busytone(A),\ dialtone(B)\}$. Figure 4.2 shows the state transition diagram that is obtained from the STR specification shown in Figure 4.1. In this diagram each circle represents a state and each arc between two states represents a state transition caused by execution of a rule instance. States that is not reachable from the initial state $\{idle(A), idle(B)\}$ are omitted in the diagram.

Let $V$ denote the set of states. For each instance $t$ of a rule, we define a relation $\xrightarrow{t}$ over states ($\xrightarrow{t} \subseteq V \times V$) as follows: $s \xrightarrow{t} s'$ iff the execution of $t$ causes $s'$ from $s$. We also define a *computation* as a sequence of states $s_0 s_1 \cdots s_k$. such that for each $0 \leq i < k$, (i) $s_i \xrightarrow{t} s_{i+1}$ for some $t$, or (ii) no rule is enabled at $s_i$ and $s_i = s_{i+1}$. We think of the length of the computation as $k$.

## 4.3 Symbolic Representation

### 4.3.1 State Transition

To apply symbolic model checking to service specifications, it is necessary to encode the state space and the transition relation by Boolean functions.

Let $\mathcal{P} = \{p_1, \cdots, p_m\}$ be the set of all instances of predicates and let $\mathcal{T} = \{t_1, \cdots, t_n\}$ be the set of all instances of rules ($m = |\mathcal{P}|$ and $n = |\mathcal{T}|$). A state $s$ can then be viewed as a Boolean vector $s = (b_1, \cdots, b_m)$ such that $b_i = true$ iff an instance $p_i$ of a predicate holds in that state.

Any set of states can be represented as a Boolean function such that

$$f(s) = \begin{cases} true & s \in \text{the set} \\ false & \text{otherwise.} \end{cases}$$

We say that $f$ is a *characteristic function* of the state set.

For example, the characteristic function $E_t(s)$ of the set of states where $t \in \mathcal{T}$ is enabled is

$$E_t(s) = \bigwedge_{p_i \in Pre[t]} b_i \wedge \bigwedge_{p_i \in \hat{P}re[t]} \neg b_i.$$

Any relation $R$ over states can be similarly encoded since they are simply sets of

tuples.

$$F(s,s') = \begin{cases} true & sRs' \\ false & \text{otherwise.} \end{cases}$$

Now consider representing the relation $\xrightarrow{t}$ by Boolean function $T_t(s,s')$. Since execution of $t$ causes (i) predicate instances in $Post[t]$ to hold, (ii) those in $Pre[t]$ but not in $Post[t]$ not to hold, and (iii) those in neither $Pre[t]$ nor $Post[t]$ to be unchanged, we have

$$T_t(s,s') = E_t(s) \wedge \bigwedge_{p_i \in Post[t]} b'_i \ \wedge \bigwedge_{p_i \in Pre[t] \setminus Post[t]} \neg b'_i \wedge \bigwedge_{p_i \in \mathcal{P} \setminus (Pre[t] \cup Post[t])} (b_i \leftrightarrow b'_i)$$

where $s' = (b'_1, \cdots, b'_m)$.

For example, consider the specification shown in Figure 4.1. Let $t$ be the instance of the rule '$pots4 : dialtone(x), \neg idle(y)[dial(x,y)]busytone(x)$' with substitution $(x,y) = (A,B)$. Since $Pre[t] = \{dialtone(A)\}$, $\hat{P}re[t] = \{idle(B)\}$, and $Post[t] = \{busytone(A)\}$, we have

$$\begin{aligned} T_t \ = \ & dialtone(A) \wedge \neg idle(B) \wedge busytone(A)' \wedge \neg dialtone(A)' \\ & \wedge (idle(A) \leftrightarrow idle(A)') \wedge (idle(B) \leftrightarrow idle(B)') \\ & \wedge (dialtone(B) \leftrightarrow dialtone(B)') \wedge (busytone(B) \leftrightarrow busytone(B)') \\ & \wedge (calling(A,B) \leftrightarrow calling(A,B)') \wedge (calling(B,A) \leftrightarrow calling(B,A)') \\ & \wedge (path(A,B) \leftrightarrow path(A,B)') \wedge (path(B,A) \leftrightarrow path(B,A)') \end{aligned}$$

(The same symbol is used to denote each predicate and its corresponding Boolean variable, since this is convenient and causes no confusion.)

In SMV program, the formula $T_t$ is described as follows.

```
dialtone_A & !idle_B
& next(busytone_A) & !next(dialtone_A)
& next(idle_A)=idle_A
& next(idle_B)=idle_B
& next(dialtone_B)=dialtone_B
& next(busytone_B)=busytone_B
& next(calling_A_B)=calling_A_B
& next(calling_B_A)=calling_B_A
& next(path_A_B)=path_A_B
& next(path_B_A)=path_B_A
```

The transition relation of the whole system is represeted by $T_{t_i}(s, s')$ as the following formula $T_{t_1}(s, s') \vee T_{t_2}(s, s') \vee \cdot \vee T_{t_n}(s, s')$ where $T_{t_i}(s, s')$ represents the transition by the instance $t$.

### 4.3.2 Interaction State

As described in the previous chapter, to use SMV, it is necessary to express the property to be verified as a formula of CTL. To do this, first we describe the set of states where interactions occur as a formula $f_G(s)$. And we model check the CTL formula $AG\neg(f_G(s))$. If this CTL formula is not true, the system can be in the state where an interaction occurs. Thus we can detect the interaction.

#### Nondeterminism

Nondeterminism occurs at a state $s$ iff two rules, $r1$ and $r2$, can be triggered by the same event $e$ at $s$. As shown in the previous example, when such $r1$, $r2$, and $e$ are given, the set of states where they are enabled simultaneously is represented by

$$\bigvee_{\{\theta 1,\theta 2\}:e[r1\theta 1]=e[r1\theta 2]} E_{r1\theta 1} \wedge E_{r2\theta 2}$$

Thus the characteristic function for the set of all states where nondeterminism occurs is

$$\bigvee_{\{r1,r2\}:r1,r2\in R} \bigvee_{\{\theta 1,\theta 2\}:e[r1\theta 1]=e[r2\theta 2]} E_{r1\theta 1} \wedge E_{r2\theta 2}$$

For example, the caracteristic function $f_G(s)$ representing nondeterministic states for

the specification shown in Fig. 4.1 is described as follows.

$$
\begin{aligned}
& idle(A) \wedge calling(B, A) \\
& \vee idle(B) \wedge calling(A, B) \\
& \vee dialtone(A) \wedge calling(A, B) \\
& \vee dialtone(A) \wedge path(A, B) \wedge path(B, A) \\
& \vee dialtone(A) \wedge busytone(A) \\
& \vee calling(A, B) \wedge path(A, B) \wedge path(B, A) \\
& \vee calling(A, B) \wedge busytone(A) \\
& \vee path(A, B) \wedge path(B, A) \wedge busytone(A) \\
& \vee dialtone(B) \wedge calling(B, A) \\
& \vee dialtone(B) \wedge path(B, A) \wedge path(A, B) \\
& \vee dialtone(B) \wedge busytone(B) \\
& \vee calling(B, A) \wedge path(B, A) \wedge path(A, B) \\
& \vee calling(B, A) \wedge busytone(B) \\
& \vee path(B, A) \wedge path(A, B) \wedge busytone(B) \\
& \vee dialtone(A) \wedge idle(B) \wedge dialtone(A) \wedge \neg idle(B) \\
& \vee dialtone(B) \wedge idle(A) \wedge dialtone(B) \wedge \neg idle(A)
\end{aligned}
$$

**Invariant Violation**

Given an invariant that is intended to be satisfied by a service, whether it is satisfied or not can be decided by checking the reachability to states where the property does not hold. In this case

$$f_G(s) = \neg Inv(s)$$

where $Inv(s)$ is the Boolean function representing the set of states where the invariant property holds.

**Other Types of Interaction**

Although we limit our discussion on detecting nondeterminism and invariant violation in this paper, other types of interaction can be detected by the proposed method, if the problem of detecting interaction can be reduced to reachability checking. Deadlock is such an example. In our context, deadlock means the situation where functional conflicts

of two or more services cause a mutual prevention of their service execution. The problem of deciding whether deadlock occurs or not can be reduced to the problem of checking reachability to states where no rule is enabled. In this case $f_G$ will be

$$f_G(s) = \bigwedge_{t \in \mathcal{T}} \neg E_t(s).$$

## 4.4 Experimental Results

In this section, we show the experimental results for interaction detection for the seven services described in Section 4.1 using symbolic model checking. Combining two of the seven services, we examined a total of the 21 pairs. The experiments were performed on a Linux workstation with a 853 MHz Pentium III processor.

### 4.4.1 Nondeterminism

Table 4.1 shows the results of applying SMV to interaction detection. As shown in this table, interactions are detected in only one or two minites for almost all pairs of services by SMV. However, for the example which has large states such as CW+CF, SMV required more than three hours to complete detection.

By enabling 'early' option, it is possible to force SMV to work on-the-fly; that is, when using this option, SMV incrementally checks whether or not the property holds in a breadth-first manner, and terminates immediately if it finds that the property can be violated. Table 4.1 also shows the running time of SMV with this option enabled. As expected, this resulted in short detection time for some service combinations. However, for some cases such as CW+CF, CW+DO, CW+DT, CW+DC, CW+OCS, or CW+TCS, it ended up with much larger running time. A common characteristic of these combinations is that the formula representing $f_G$ is very large. Hence the reason is thought to be that the benefit of early termination was diminished by time consumed at each stage of the incremental checking.

### 4.4.2 Invariant Violation

We consider invariant properties for four of the seven services as follows

**OCS** "If $x$ puts $y$ in the OCS screening list, $x$ is never calling $y$ at any time" $(\neg OCS(x,y) \vee \neg calling(x,y))$

**TCS** "If $x$ puts $y$ in the TCS screening list, $y$ is never calling $x$ at any time" $(\neg TCS(x,y) \vee \neg calling(y,x))$

**DO** "If $x$ subscribes to $DO$, $x$ never receives dialtone at any time" $(\neg DO(x) \vee \neg dialtone(x))$

**DT** "If $x$ subscribes to $DT$, $y$ is never calling $x$ at any time" $(\neg DT(x) \vee \neg calling(y,x))$

Tables 4.2 shows the results of applying SMV to detection of invariant violation. As shown in this table, for all pairs of services interactions are detected in only one minite. In this case, the formula representing $f_G$ is not so large. Thus the incremental checking does not require so much time that early termination makes the detection efficient.

Table 4.1: Performance of SMV for nondeterminism detection.

| | interaction | SMV | SMV(-early) | ♯ of states |
|---|---|---|---|---|
| CW+CF | detected | 12859.40 | 90473 | $2^{78}$ |
| CW+OCS | detected | 44.23 | 194.91 | $2^{60}$ |
| CW+TCS | detected | 39.28 | 168.28 | $2^{60}$ |
| CW+DO | - | 70.21 | 726.40 | $2^{57}$ |
| CW+DT | detected | 82.12 | 410.37 | $2^{57}$ |
| CW+DC | - | 66.55 | 666.63 | $2^{60}$ |
| CF+OCS | detected | 22.79 | 5.51 | $2^{57}$ |
| CF+TCS | detected | 27.59 | 5.55 | $2^{57}$ |
| CF+DO | - | 6.47 | 13.01 | $2^{54}$ |
| CF+DT | detected | 12.48 | 8.57 | $2^{54}$ |
| CF+DC | - | 20.20 | 31.26 | $2^{57}$ |
| OCS+TCS | detected | 1.86 | 0.29 | $2^{39}$ |
| OCS+DO | - | 0.94 | 1.01 | $2^{36}$ |
| OCS+DT | detected | 1.23 | 0.24 | $2^{36}$ |
| OCS+DC | - | 1.59 | 1.72 | $2^{39}$ |
| TCS+DO | - | 1.24 | 1.27 | $2^{36}$ |
| TCS+DT | detected | 1.66 | 0.24 | $2^{36}$ |
| TCS+DC | - | 2.19 | 2.35 | $2^{39}$ |
| DO+DT | - | 0.65 | 0.76 | $2^{33}$ |
| DO+DC | detected | 1.22 | 0.25 | $2^{36}$ |
| DT+DC | - | 0.65 | 0.76 | $2^{36}$ |

Table 4.2: Performance of SMV for invariant violation detection.

| | interaction | SMV | SMV(-early) | ♯ of states |
|---|---|---|---|---|
| CW+OCS | detected | 23.51 | 13.41 | $2^{60}$ |
| CW+TCS | detected | 24.96 | 13.81 | $2^{60}$ |
| CW+DO | - | 37.22 | 37.37 | $2^{57}$ |
| CW+DT | detected | 40.29 | 35.43 | $2^{57}$ |
| CF+OCS | detected | 22.46 | 1.10 | $2^{57}$ |
| CF+TCS | detected | 27.34 | 1.16 | $2^{57}$ |
| CF+DO | - | 6.25 | 6.32 | $2^{54}$ |
| CF+DT | detected | 10.83 | 0.97 | $2^{54}$ |
| OCS+TCS | detected | 1.83 | 0.30 | $2^{39}$ |
| OCS+DO | - | 0.98 | 1.00 | $2^{36}$ |
| OCS+DT | - | 1.30 | 1.27 | $2^{36}$ |
| OCS+DC | detected | 1.83 | 0.32 | $2^{39}$ |
| TCS+DO | - | 1.28 | 1.28 | $2^{36}$ |
| TCS+DT | - | 1.70 | 1.72 | $2^{36}$ |
| TCS+DC | detected | 2.50 | 0.33 | $2^{39}$ |
| DO+DT | - | 0.66 | 0.65 | $2^{33}$ |
| DO+DC | - | 1.31 | 1.23 | $2^{36}$ |
| DT+DC | - | 0.36 | 0.38 | $2^{36}$ |

# Chapter 5

# Bounded Model Checking

Bounded model checking has received recent attention as an efficient verification method [7]. The basic idea of this method is to reduce the model checking problem to the propositional satisfiability decision problem.

For asynchronous systems, however, the existing bounded model checking does not work well because the propositional formula to be checked tends to become very large for such systems. Because of the asynchronous nature of telecommunication systems, it is thus not practical to apply the original method to feature interaction detection.

In order to avoid this problem we develop a new encoding scheme. We describe the scheme in detail in this section.

## 5.1 Existing Scheme

Similar to SMV, to apply bounded model checking to service specifications, it is necessary to encode the state space and the transition relation by Boolean functions. We use the same manner as shown in Section 4.3 for symbolic representation.

Let $G$ denote the set of states whose reachability is to be decided and let $f_G(S)$ be the characteristic function for $G$. Although there are some variations [38], the basic formula used for checking reachability in bounded model checking is as follows.

$$I(s_0) \wedge T(s_0, s_1) \wedge T(s_1, s_2) \cdots \wedge T(s_{k-1}, s_k) \wedge (f_G(s_0) \vee \cdots \vee f_G(s_k))$$

where $I(S)$ is the characteristic function of the set of the initial states, and

$$T(s,s') = \begin{cases} true & s' \text{ is reachable from } s \text{ in one step,} \\ & \text{or } s \text{ has no next states and } s = s'. \\ false & \text{otherwise.} \end{cases}$$

Clearly, $I(s_0) \wedge T(s_0,s_1) \wedge T(s_1,s_2) \cdots \wedge T(s_{k-1},s_k) = true$ iff $s_0, s_1, \cdots, s_k$ is a computation from the initial states. Hence the above formula is satisfiable iff there is a state that is in $G$ and reachable from one of the initial states in at most $k$ steps. By checking the satisfiability of the formula, therefore, the verification can be carried out.

In practice, the formula often needs to be transformed into conjunctive normal form (CNF), since most of SAT solvers require for input formulas to be in that form. However the logically equivalent CNF formula can be exponential with respect to the size of the original formula. To avoid this, it is usual to use *structure preserving transformation* [36], which guarantees that the size of the resulting CNF formula is linear with the original formula. Thus the efficiency of verification critically depends on the size of the original formula in textual form.

Since we assume that exactly one rule is executed at a time, $T(s,s')$ will be

$$T(s,s') = T_{t_1}(s,s') \vee \cdots \vee T_{t_n}(s,s') \vee ((\bigwedge_{p_i \in \mathcal{P}} b_i \leftrightarrow b_i') \wedge \neg E_{t_1}(s) \wedge \cdots \wedge \neg E_{t_n}(s))$$

It should be noted that this formula would be very large in size in practice. Since $T_t$ contains at least $m$ literals, the total number of the literals in $T$ is greater than $m * n$ literals.

## 5.2 Proposed Scheme

### 5.2.1 Encoding

Our proposed scheme alleviates the above problem with a new encoding. Let $Chng[t]$ denote the set of predicate instances that change as a result of execution of rule instance $t$; that is, $Chng[t] = (Post[t] \backslash Pre[t]) \cup (Pre[t] \backslash Post[t])$. Then $T_t$ can be transformed as

follows:

$$T_t(s,s') = E_t(s) \wedge \bigwedge_{p_i \in Post[t]} b_i' \wedge \bigwedge_{p_i \in Pre[t] \setminus Post[t]} \neg b_i' \wedge \bigwedge_{p_i \in \mathcal{P} \setminus (Pre[t] \cup Post[t])} (b_i \leftrightarrow b_i')$$

$$= \bigwedge_{p_i \in Pre[t]} b_i \wedge \bigwedge_{p_i \in \hat{P}re[t]} \neg b_i$$

$$\wedge \bigwedge_{p_i \in Post[t] \setminus Pre[t]} b_i' \wedge \bigwedge_{p_i \in Pre[t] \setminus Post[t]} \neg b_i' \wedge \bigwedge_{p_i \in \mathcal{P} \setminus Chng[t]} (b_i \leftrightarrow b_i').$$

Now let $D_t(s,s')$ be defined as follows.

$$D_t(s,s') = T_t(s,s') \vee \bigwedge_{p_i \in \mathcal{P}} (b_i \leftrightarrow b_i')$$

We have

$$D_t(s,s') = T_t(s,s') \vee \bigwedge_{p_i \in \mathcal{P}} (b_i \leftrightarrow b_i')$$

$$= (( \bigwedge_{p_i \in Pre[t]} b_i \wedge \bigwedge_{p_i \in \hat{P}re[t]} \neg b_i \wedge \bigwedge_{p_i \in Post[t] \setminus Pre[t]} b_i' \wedge \bigwedge_{p_i \in Pre[t] \setminus Post[t]} \neg b_i')$$

$$\vee \bigwedge_{p_i \in Chng[t]} (b_i \leftrightarrow b_i'))$$

$$\wedge \bigwedge_{p_i \in \mathcal{P} \setminus Chng[t]} (b_i \leftrightarrow b_i')$$

For example, let $t$ be the instance of the rule *pots*4 in Figure 4.1 with substitution $(x,y) = (A,B)$. Then

$$\begin{aligned} D_t = {} & ((dialtone(A) \wedge \neg idle(B) \wedge busytone(A)' \wedge \neg dialtone(A)') \\ & \vee ((dialtone(A) \leftrightarrow dialtone(A)') \wedge (busytone(A) \leftrightarrow busytone(A)'))) \\ & \wedge (idle(A) \leftrightarrow idle(A)') \wedge (idle(B) \leftrightarrow idle(B)') \\ & \wedge (dialtone(B) \leftrightarrow dialtone(B)') \wedge (busytone(B) \leftrightarrow busytone(B)') \\ & \wedge (calling(A,B) \leftrightarrow calling(A,B)') \wedge (calling(B,A) \leftrightarrow calling(B,A)') \\ & \wedge (path(A,B) \leftrightarrow path(A,B)') \wedge (path(B,A) \leftrightarrow path(B,A)') \end{aligned}$$

It is easy to see that $D_t(S,S') = true$ iff $S \xrightarrow{t} S'$ or $S = S'$. In other words, $D_t(S,S')$ differs from $T_t(S,S')$ only in that $D_t(S,S')$ evaluates to true also when $S = S'$. Using

this property, a step (or more) can be represented by a conjunction of $D_t$ as follows.

$$D_{t_1}(s_0, s_1) \wedge D_{t_2}(s_1, s_2) \wedge \cdots \wedge D_{t_n}(s_{n-1}, s_n)$$

Note that this is in contrast to the traditional encoding, where a disjunction of $T_t(S, S')$ is used to represent one step. By definition, $s_0, s_1, \cdots, s_n$ satisfies this formula iff for any $0 \leq i < n$, $s_i \stackrel{t_{i+1}}{\rightarrow} s_{i+1}$ or $s_i = s_{i+1}$. This means that if the formula evaluates to true, $s_n$ is reachable from $s_0$ in at most $n$ steps (including 0 steps), and that if there is at least one $t_i$ such that $s_0 \stackrel{t_i}{\rightarrow} s'$, it is satisfiable with an assignment such that $s_0 = \cdots = s_{i-1}, s_i = \cdots = s_n = s'$.

As a result, our proposed scheme uses the following formula for the verification.

$$\begin{aligned}
\varphi \;=\; & I(s_0) \\
& \wedge D_{t_1}(s_0, s_1) \wedge D_{t_2}(s_1, s_2) \wedge \cdots \wedge D_{t_n}(s_{n-1}, s_n) \\
& \wedge D_{t_1}(s_n, s_{n+1}) \wedge D_{t_2}(s_{n+1}, s_{n+2}) \wedge \cdots \wedge D_{t_n}(s_{2n-1}, s_{2n}) \\
& \cdots \\
& \wedge D_{t_1}(s_{(k-1)*n}, s_{(k-1)*n+1}) \wedge \cdots \wedge D_{t_n}(s_{k*n-1}, s_{k*n}) \\
& \wedge f_G(s_{k*n})
\end{aligned}$$

If the formula $\varphi$ is satisfiable, then we can conclude that there is a state in $G$ that can be reached from the initial state in at most $k * n$ steps. On the other hand, if the formula $\varphi$ is unsatisfiable, then there is no state in $G$ that can be reached from the initial state in less than or equal to $k$ steps.

An important observation here is that the method may be able to find a state in $G$ that requires more than $k$ transition executions to reach.

### 5.2.2 Constructing a Succinct Formula

The most important advantage of our scheme is that $\varphi$ can be converted into a much succinct formula that is not logically equivalent but has the same satisfiability. Let $s_j = (b_{1,j}, b_{2,j}, \cdots, b_{m,j})$ and $s_{j+1} = (b_{1,j+1}, b_{2,j+1}, \cdots, b_{m,j+1})$. In each $D_t(s_j, s_{j+1})$ in $\varphi$, term $(b_{i,j} \leftrightarrow b_{i,j+1})$ for any $p_i \in \mathcal{P} \backslash Chng[t]$ appears as a conjunct. Because of this, $\varphi$ is satisfiable only if $b_{i,j}$ and $b_{i,j+1}$ have the same value. Hence a shorter formula that maintains the satisfiability is obtained by removing $(b_{i,j} \leftrightarrow b_{i,j+1})$ and replacing $b_{i,j+1}$

with $b_{i,j}$. That is, for each $D_t(s_j, s_{j+1})$ in $\varphi$, $(b_{i,j} \leftrightarrow b_{i,j+1})$ for all $p_i \in \mathcal{P} \backslash Chng[t]$ can be removed by quantifying away $b_{i,j+1}$ by applying the formula below.

$$\exists b_{i,j+1}(F \wedge (b_{i,j} \leftrightarrow b_{i,j+1})) = F|_{b_{i,j+1} \rightarrow b_{i,j}}$$

where $F$ is an intermediate formula obtained from $\phi$ and $F|_{y \rightarrow x}$ denotes the formula obtained from $F$ by replacing $b_{i,j+1}$ with $b_{i,j}$.

Note that $b_{i,j}$ may also be replaced by a further earlier version of variable $b_{i,j-1}$. In that case $b_{i,j+1}$ is replaced with $b_{i,j-1}$ as a result.

Consequently, $D_t$ in $\phi$ can be replaced with

$$((\bigwedge_{p_i \in Pre[t]} b_i \wedge \bigwedge_{p_i \in \hat{P}re[t]} \neg b_i \wedge \bigwedge_{p_i \in Post[t] \backslash Pre[t]} b'_i \wedge \bigwedge_{p_i \in Pre[t] \backslash Post[t]} \neg b'_i) \vee \bigwedge_{p_i \in Chng[t]} (b_i \leftrightarrow b'_i))$$

by appropriately replacing some variables.

The number of literals occurring in the above formula is $4 * |Pre[t] \backslash Post[t]| + |Pre[t] \cap Post[t]| + 3 * |Post[t] \backslash Pre[t]| + |Post[t] \cap \hat{P}re[t]| + |\hat{P}re[t] \backslash Post[t]|$. $T_t$, which is the counterpart in the traditional encoding, contains at least $|\mathcal{P}|$ literals. Hence the proposed scheme can exploit its advantage if $Pre[t] \cup \hat{P}re[t] \cup Post[t]$ is a small fraction of the whole set of predicate instances $\mathcal{P}$. This is usually the case for the service specifications we consider and is more likely when the number of users is large.

Figure 5.1 shows the algorithm that directly constructs the shorter formula for a given $k$. In the algorithm, variable $c_i$ is used to denote the earlier version of the variable that is substituted for $b_{i,j}$; that is, $b_{i,j}$ will be replaced with $b_{i,c_i}$.

### 5.2.3 Illustrative Example

Consider an erroneous communication service that is obtained by replacing rule $pots3$ in the POTS specification in Figure 4.1 with

$$pots3' : dialtone(x)\ [dial(x, y)]\ calling(x, y).$$

Here we demonstrate how we can verify that a state where two different rules $pots1$ and $pots6$ are simultaneously enabled for the same event $onhook(x)$ is reachable. The set of such nondeterministic states can be represented by a characteristic function

$$f_G(s) = (idle(A) \wedge calling(B, A)) \vee (idle(B) \wedge calling(A, B))$$

for $p_i \in \mathcal{P}$

$c_i := 0$;

$j := 0$;

$X := I(s)|_{s_i \to s_{i,0} \text{ for all } p_i \in \mathcal{P}}$;

for $step = 1, \cdots, k$ {

for $t \in \mathcal{T}$ {

$j := j + 1$;

$X := X \wedge$

$$((\bigwedge_{p_i \in Pre[t]} b_{i,c_i} \wedge \bigwedge_{p_i \in \hat{P}re[t]} \neg b_{i,c_i} \wedge \bigwedge_{p_i \in Post[t] \backslash Pre[t]} b_{i,j} \wedge \bigwedge_{p_i \in Pre[t] \backslash Post[t]} \neg b_{i,j}) \vee \bigwedge_{p_i \in Chng[t]} (b_{i,c_i} \leftrightarrow b_{i,j}))$$

for $p_i \in Chng[t]$ $(= (Post[t] \backslash Pre[t]) \cup (Pre[t] \backslash Post[t]))$

$c_i := j$;

}

}

$X := X \wedge f_G(S)|_{b_i \to b_{i,c_i} \text{ for all } p_i \in \mathcal{P}}$;

Figure 5.1: Algorithm for constructing the formula used for verification.

$I(s)$, which represents the set of the initial states, is

$$\begin{aligned} I(s) \quad = \quad & idle(A) \wedge idle(B) \wedge \neg dialtone(A) \wedge \neg dialtone(B) \\ & \wedge \neg busytone(A) \wedge \neg busytone(B) \\ & \wedge \neg path(A, B) \wedge \neg path(B, A) \wedge \neg calling(A, B) \wedge \neg calling(A, B) \end{aligned}$$

When $k = 1$, the formula shown in Figure 5.2 is obtained by the algorithm in Figure 5.1.

This formula is satisfiable by, for example, assigning $true$ to the following variables and $false$ to the others.

$$idle_0(A), idle_0(B), dialtone_1(A), idle_2(B), dialtone_3(A), idle_4(B), calling_5(A, B),$$
$$idle_9(B), calling_9(A, B), idle_{10}(B), calling_{11}(A, B), idle_{14}(B), idle_{16}(B)$$

Assignments satisfying the formula represent computations from the initial state to the states where a confliction occurs. For example, the above assignment corresponds to

the following computation.

$$\{idle(A), idle(B)\} \overset{pots1<X|A,Y|B>}{\rightarrow} \{dialtone(A), idle(B)\} \overset{pots3'<X|A,Y|B>}{\rightarrow} \{calling(A,B), idle(B)\}$$

Note that $pots1$ and $pots6$ are both enabled for event $offhook(B)$ at state $\{calling(A,B), idle(B)\}$. As can be seen, computations of length more than $k$ can be checked by the proposed method.

Although this is not an example of feature interaction detection (since no feature is considered), once a specification specifying multiple service is given, feature interaction detection can be carried out the same way as described here.

## 5.3 Representing Interaction State

As the remaining problem is to represent states where interaction occurs by Boolean function $f_G(s)$, it is previously described in Section 4.3.2. For example, the caracteristic function $f_G(s)$ representing nondeterministic states for erroneous specification which is shown in Section 5.2.3 is described as follows.

$$\begin{aligned}
&idle(A)_{15} \wedge calling(B,A)_{12} \\
&\vee idle(B)_{16} \wedge calling(A,B)_{11} \\
&\vee dialtone(A)_{17} \wedge calling(A,B)_{11} \\
&\vee dialtone(A)_{17} \wedge path(A,B)_{14} \wedge path(B,A)_{14} \\
&\vee dialtone(A)_{17} \wedge busytone(A)_{17} \\
&\vee calling(A,B)_{11} \wedge path(A,B)_{14} \wedge path(B,A)_{14} \\
&\vee calling(A,B)_{11} \wedge busytone(A)_{17} \\
&\vee path(A,B)_{14} \wedge path(B,A)_{14} \wedge busytone(A)_{17} \\
&\vee dialtone(B)_{18} \wedge calling(B,A)_{12} \\
&\vee dialtone(B)_{18} \wedge path(B,A)_{14} \wedge path(A,B)_{14} \\
&\vee dialtone(B)_{18} \wedge busytone(B)_{18} \\
&\vee calling(B,A)_{12} \wedge path(B,A)_{14} \wedge path(A,B)_{14} \\
&\vee calling(B,A)_{12} \wedge busytone(B)_{18} \\
&\vee path(B,A)_{14} \wedge path(A,B)_{14} \wedge busytone(B)_{18} \\
&\vee dialtone(A)_{17} \wedge dialtone(A)_{17} \wedge \neg idle(B)_{16} \\
&\vee dialtone(B)_{18} \wedge dialtone(B)_{18} \wedge \neg idle(A)_{15}
\end{aligned}$$

$$
\begin{aligned}
&(idle_0(A) \wedge idle_0(B) \wedge \neg dialtone_0(A) \wedge \neg dialtone_0(B) \wedge \neg busytone_0(A) \wedge \neg busytone_0(B) \\
&\quad \wedge \neg path_0(A,B) \wedge \neg path_0(B,A) \wedge \neg calling_0(A,B) \wedge \neg calling_0(A,B)) \\
&\wedge ((idle_0(A) \wedge dialtone_1(A) \wedge \neg idle_1(A)) \\
&\quad \vee ((idle_0(A) \leftrightarrow idle_1(A)) \wedge (dialtone_0(A) \leftrightarrow dialtone_1(A))) \\
&\wedge ((idle_0(B) \wedge dialtone_2(B) \wedge \neg idle_2(B)) \\
&\quad \vee ((idle_0(B) \leftrightarrow idle_2(B)) \wedge (dialtone_0(B) \leftrightarrow dialtone_2(B))) \\
&\wedge ((dialtone_1(A) \wedge idle_3(A) \wedge \neg dialtone_3(A)) \\
&\quad \vee ((idle_1(A) \leftrightarrow idle_3(A)) \wedge (dialtone_1(A) \leftrightarrow dialtone_3(A))) \\
&\wedge ((dialtone_2(B) \wedge idle_4(B) \wedge \neg dialtone_4(B)) \\
&\quad \vee ((idle_2(B) \leftrightarrow idle_4(B)) \wedge (dialtone_2(B) \leftrightarrow dialtone_4(B))) \\
&\wedge ((dialtone_3(A) \wedge calling_5(A,B) \wedge \neg dialtone_5(A)) \\
&\quad \vee ((dialtone_3(A) \leftrightarrow dialtone_5(A)) \wedge (calling_0(A,B) \leftrightarrow calling_5(A,B))) \\
&\wedge ((dialtone_4(B) \wedge calling_6(B,A) \wedge \neg dialtone_6(B)) \\
&\quad \vee ((dialtone_4(B) \leftrightarrow dialtone_6(B)) \wedge (calling_0(B,A) \leftrightarrow calling_6(B,A))) \\
&\wedge ((dialtone_5(A) \wedge \neg idle_4(B) \wedge busytone_7(A) \wedge \neg dialtone_7(A)) \\
&\quad \vee ((dialtone_5(A) \leftrightarrow dialtone_7(A)) \wedge (busytone_0(A) \leftrightarrow busytone_7(A))) \\
&\wedge ((dialtone_6(B) \wedge \neg idle_3(A) \wedge busytone_8(B) \wedge \neg dialtone_8(B)) \\
&\quad \vee ((dialtone_6(B) \leftrightarrow dialtone_8(B)) \wedge (busytone_0(B) \leftrightarrow busytone_8(B))) \\
&\wedge ((calling_5(A,B) \wedge idle_9(A) \wedge idle_9(B) \wedge \neg calling_9(A,B)) \\
&\quad \vee ((idle_3(A) \leftrightarrow idle_9(A)) \wedge (idle_4(B) \leftrightarrow idle_9(B)) \wedge (calling_5(A,B) \leftrightarrow calling_9(A,B))) \\
&\wedge ((calling_6(B,A) \wedge idle_{10}(A) \wedge idle_{10}(B) \wedge \neg calling_{10}(B,A)) \\
&\quad \vee ((idle_9(A) \leftrightarrow idle_{10}(A)) \wedge (idle_9(B) \leftrightarrow idle_{10}(B)) \wedge (calling_6(B,A) \leftrightarrow calling_{10}(B,A))) \\
&\wedge ((calling_9(A,B) \wedge path_{11}(A,B) \wedge path_{11}(B,A) \wedge \neg calling_{11}(A,B)) \\
&\quad \vee ((calling_9(A,B) \leftrightarrow calling_{11}(A,B)) \wedge (path_0(A,B) \leftrightarrow path_{11}(A,B)) \wedge (path_0(B,A) \leftrightarrow path_{11}(B,A))) \\
&\wedge ((calling_{10}(B,A) \wedge path_{12}(A,B) \wedge path_{12}(B,A) \wedge \neg calling_{12}(B,A)) \\
&\quad \vee ((calling_{10}(B,A) \leftrightarrow calling_{12}(B,A)) \wedge (path_{11}(A,B) \leftrightarrow path_{12}(A,B)) \wedge (path_{11}(B,A) \leftrightarrow path_{12}(B,A))) \\
&\wedge ((path_{12}(A,B) \wedge path_{12}(B,A) \wedge idle_{13}(A) \wedge busytone_{13}(B) \wedge \neg path_{13}(A,B) \wedge \neg path_{13}(B,A)) \\
&\quad \vee ((idle_{10}(A) \leftrightarrow idle_{13}(A)) \wedge (busytone_8(B) \leftrightarrow busytone_{13}(B)) \wedge (path_{12}(A,B) \leftrightarrow path_{13}(A,B)) \\
&\qquad \wedge (path_{12}(B,A) \leftrightarrow path_{13}(B,A))) \\
&\wedge ((path_{13}(A,B) \wedge path_{13}(B,A) \wedge idle_{14}(B) \wedge busytone_{14}(A) \wedge \neg path_{14}(A,B) \wedge \neg path_{14}(B,A)) \\
&\quad \vee ((idle_{10}(B) \leftrightarrow idle_{14}(B)) \wedge (busytone_7(A) \leftrightarrow busytone_{14}(A)) \wedge (path_{13}(A,B) \leftrightarrow path_{14}(A,B)) \\
&\qquad \wedge (path_{13}(B,A) \leftrightarrow path_{14}(B,A))) \\
&\wedge ((busytone_{14}(A) \wedge idle_{15}(A) \wedge \neg busytone_{15}(A)) \\
&\quad \vee ((idle_{13}(A) \leftrightarrow idle_{15}(A)) \wedge (busytone_{14}(A) \leftrightarrow busytone_{15}(A))) \\
&\wedge ((busytone_{13}(B) \wedge idle_{16}(B) \wedge \neg busytone_{16}(B)) \\
&\quad \vee ((idle_{14}(B) \leftrightarrow idle_{16}(B)) \wedge (busytone_{13}(B) \leftrightarrow busytone_{16}(B))) \\
&\wedge ((dialtone_7(A) \wedge busytone_{17}(A) \wedge \neg dialtone_{17}(A)) \\
&\quad \vee ((dialtone_7(A) \leftrightarrow dialtone_{17}(A)) \wedge (busytone_{15}(A) \leftrightarrow busytone_{17}(A))) \\
&\wedge ((dialtone_8(B) \wedge busytone_{18}(B) \wedge \neg dialtone_{18}(B)) \\
&\quad \vee ((dialtone_8(B) \leftrightarrow dialtone_{18}(B)) \wedge (busytone_{16}(B) \leftrightarrow busytone_{18}(B))) \\
&\wedge (idle_{15}(A) \wedge calling_{12}(B,A)) \vee (idle_{16}(B) \wedge calling_{11}(A,B))
\end{aligned}
$$

Figure 5.2: Resulting formula for an incorrect POTS specification.

## 5.4 Comparison Results

In order to evaluate the effectiveness of the proposed method, we conducted experimental evaluation for the seven services described in Section 4.1. We used the same ordering as in the given specification in the experiment. Combining two of the seven services, we examined a total of the 21 pairs.

The experiments were performed on a Linux workstation with a 853 MHz Pentium III processor. The number of users was assumed to be four. ZChaff, an implementation of Chaff [31], was used as a SAT solver.

For each problem we incremented $k$ until interaction was detected. Tables 5.1 and 5.3 show the value of $k$ for which interaction was first found and the time (in seconds) required by ZChaff to find a satisfying assignment for that value of $k$.

### 5.4.1 Nondeterminism

It has been known that out of a total of the 21 pairs of the seven services, 11 pairs cause nondeterminism. Since the proposed method in itself cannot prove the absence of feature interaction, we evaluated the performance of the detection method for these combinations only.

#### Comparison with Traditional Scheme

Table 5.1 compares the proposed encoding and the traditional one with respect to the running time, in seconds, required to detect nondeterministic states for these specifications. Items in the 'length' column represent the length of the shortest counterexample, that is, the shortest computation from the initial state to a nondeterministic state.

As can be seen in this table, when using the proposed encoding, interaction was detected with $k$ of less than or equal to three for all cases. For CW plus CF case, for example, $k = 2$ was sufficient while the shortest counterexample computation is of length 10. This is because it may be possible to check execution of two or more rules by one formula $D_{t_1} \wedge D_{t_2} \wedge \cdots \wedge D_{t_n}$. In this experiment, we used the same ordering of rules as in the given specification in encoding the formula. Thus if two rules are executed in the order as in the specification, they can be checked by this single formula.

Table 5.1: Performance of bounded model checking for nondeterminism detection.

| | $k$ | time | Trad. scheme | length |
|---|---|---|---|---|
| CW+CF | 2 | 3.02 | 4934.76 | 10 |
| CW+DT | 3 | 4.81 | 212.10 | 8 |
| CW+OCS | 2 | 2.90 | 330.15 | 8 |
| CW+TCS | 2 | 3.80 | 1470.37 | 8 |
| CF+DT | 2 | 0.02 | 53.52 | 5 |
| CF+OCS | 2 | 0.02 | 89.32 | 5 |
| CF+TCS | 2 | 0.02 | 65.10 | 5 |
| DC+DO | 1 | 0.02 | 0.87 | 2 |
| DT+OCS | 2 | 0.05 | 1.91 | 3 |
| DT+TCS | 1 | 0.02 | 1.86 | 3 |
| OCS+TCS | 1 | 0.01 | 1.01 | 2 |

Note that the length of the shortest counterexample coincides with the smallest $k$ value at which the traditional scheme can find such a computation. This resulted in large detection time of the traditional scheme, as shown in this table.

**Comparison with other Method**

We also applied two other model checking tools to the same set of problems. The first one is SMV, and the second one is SVAL, which is a tool which we had developed for feature interaction detection [33]. The SVAL tool employs explicit state enumeration with symmetry and partial order state reduction techniques.

Table 5.2 shows the results of applying SMV and SVAL to interaction detection. Comparing with Table 5.1, it is clear that the proposed method detected interaction much more efficiently than SMV. The difference is most clear for CW plus CF. For this case, the running time of the proposed scheme was only three seconds, while SMV required more than three hours to complete detection.

As can be seen in Table 5.2, the propose method and SVAL exhibited similar performance for four cases, namely, DC+DO, DT+OCS, DT+TCS, and OCS+TCS. The common characteristic of these cases is that nondeterminism occurs at a state that is very

Table 5.2: Performance of SMV and SVAL for nondeterminism detection.

| | SMV | SMV(-early) | SVAL | length |
|---|---|---|---|---|
| CW+CF | 12859.40 | 90473.00 | 17.45 | 10 |
| CW+DT | 82.12 | 410.37 | 3.29 | 8 |
| CW+OCS | 44.23 | 194.91 | 3.37 | 8 |
| CW+TCS | 39.28 | 168.28 | 9.65 | 8 |
| CF+DT | 12.51 | 8.21 | 1.83 | 5 |
| CF+OCS | 22.80 | 5.55 | 6.11 | 5 |
| CF+TCS | 27.52 | 5.55 | 2.45 | 5 |
| DC+DO | 1.21 | 0.25 | 0.31 | 2 |
| DT+OCS | 1.23 | 0.24 | 0.06 | 3 |
| DT+TCS | 1.66 | 0.24 | 0.11 | 3 |
| OCS+TCS | 1.86 | 0.29 | 0.11 | 2 |

close to the initial state. In these cases, therefore, it is possible to detect interaction by exploring a small number of states, thus resulting in very small detection times of SVAL.

On the other hand, for the cases of CW+CF, CW+OCS, CW+TCS, CF+DT, CF+OCS, and CF+TCS, computations of relatively large length have to be examined to conclude the existence of nondeterministic states. For these cases, the proposed method outperformed the previous method, by efficiently exploring the large state space with symbolic representation.

### 5.4.2 Invariant Violation

We consider the same invariant properties described in Section 4.4.2.

Tables 5.3 and 5.4 show the performance for bounded model checking and SMV, respectively. SVAL is excluded because it does not support invariant violation checking. Comparing with Tables 5.1 and 5.2, it can be seen that these three methods exhibited similar tendencies.

Table 5.3: Performance of bounded model checking for invariant violation detection.

| | $k$ | time | Trad. scheme | length |
|---|---|---|---|---|
| CW+DT | 3 | 1.00 | 1318.41 | 10 |
| CW+OCS | 2 | 0.24 | 2795.61 | 10 |
| CW+TCS | 2 | 0.21 | 1744.04 | 10 |
| CF+DT | 2 | 0.01 | 149.74 | 6 |
| CF+OCS | 2 | 0.02 | 173.00 | 6 |
| CF+TCS | 2 | 0.03 | 1850.80 | 6 |
| DC+OCS | 2 | 0.03 | 3.57 | 3 |
| DC+TCS | 2 | 0.04 | 3.68 | 3 |
| OCS+TCS | 2 | 0.12 | 3.13 | 3 |

Table 5.4: Performance of SMV for invariant violation detection.

| | SMV | SMV(-early) | length |
|---|---|---|---|
| CW+DT | 40.29 | 35.43 | 10 |
| CW+OCS | 23.51 | 13.41 | 10 |
| CW+TCS | 24.96 | 13.81 | 10 |
| CF+DT | 10.83 | 0.97 | 6 |
| CF+OCS | 22.46 | 1.10 | 6 |
| CF+TCS | 27.34 | 1.16 | 6 |
| DC+OCS | 1.83 | 0.32 | 3 |
| DC+TCS | 2.50 | 0.33 | 3 |
| OCS+TCS | 1.83 | 0.30 | 3 |

# Chapter 6

# Conclusions

## 6.1 Achievements

First, 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.

Second, we proposed a formal method for detection of feature interactions in telecommunication services. We also use symbolic model checking method to detect interactions. In the experimental results, we can detect all interactions for given service specifications. However, in some cases the detection processes result in much time spent to finish the

detection.

To solve this, we propose to use bounded model checking to detect feature interactions. We developed a new encoding scheme that is tailored to this purpose. We demonstrated its effectiveness by applying it to practical services.

# Bibliography


[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 Transactions on Software Engineering*, 19(11):1015–1027, November 1993.

[3] A. Arora and S. Kulkarni. Designing masking fault-tolerance via nonmasking fault-tolerance. *IEEE Transactions on Software Engineering*, 24(6):435–450, 1998.

[4] Bellcore. *Advanced Intelligent Network (AIN) Release 1, Switching Systems Generic Requirements*. Bellcore Technical Advisory TA-NWT-001123, 1991.

[5] C. Bernardeschi, A. Fantechi, and L. Simoncini. Formally verifying fault tolerant system designs. *The Computer Journal*, 43(3):191–205, 2000.

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

[7] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without BDDs. In *Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), number 1579 in LNCS*, pages 193–207, 1999.

[8] L. Bouma and H. Velthuijsen. *Feature Interactions in Telecommunications Systems*. IOS Press, 1994.

[9] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transaction on Computers*, C-35(8):677–691, 1985.

[10] 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*, 98(2):142–170, 1992.

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

[12] 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*, 8(2):244–263, 1986.

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

[14] F. Cristian. A rigorous approach to fault-tolerant programming. *IEEE Transaction on Software Engineering*, 11(1):23–31, 1985.

[15] R. Dssouli, S. Some, J. W. Guillery, and N. Rico. Detection of feature interactions with REST. In *Proceedings of Fourth Workshop on Feature Interactions in Telecommunications Systems*, pages 271–283, 1997.

[16] A. Gammelgaard and E. J. Kristensen. Interaction detection, a logical approach. In *Proceedings of Second Workshop on Feature Interactions in Telecommunications Systems*, pages 178–196, 1994.

[17] F. C. Gärtner. *Specifications for Fault Tolerance: A Comedy of Failures*. Technical Report TUD-BS-1998-03, Darmstadt University of Technology, Germany, 1998.

[18] F. C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys*, 31(1):1–26, March 1999.

[19] S. Gnesi, G. Lenzini, D. Latella, A. A. C. Abbaneo, and P. Marmo. An Automatic SPIN Validation of a Safety Critical Railway Control System. In *Proc. of The International Conference on Dependable Systems and Networks (DSN 2000)*, pages 119–124. IEEE, 2000.

[20] Y. Harada, Y. Hirakawa, T. Takenaka, and N. Terashima. A conflict detection support method for telecommunication service descriptions. *IEICE Transactions on Communication*, E75-B(10):986–997, October 1992.

[21] Y. Hirakawa and T. Takenaka. Telecommunication service description using state transition rules. In *Proceedings of IEEE Int'l Workshop on Software Specification and Design*, pages 140–147, October 1991.

[22] ITU-T Recommendations Q.1200 Series. *Intelligent Network Capability Set 1 (CS1)*. ITU-T, September 1990.

[23] J. K. Jr., B. T. Smith, and A. S. Wojcik. Formal verification of fault tolerance using theorem-proving techniques. *IEEE Transaction on Computers*, 38(3):366–376, March 1989.

[24] D. O. Keck and P. J. Kuehn. The feature and service interaction problem in telecommunications systems: A survey. *IEEE Transactions on Software Engineering*, 24(10):779–796, October 1998.

[25] A. Khoumsi. Detection and resolution of interactions between services of telephone networks. In *Proceedings of Fourth Workshop on Feature Interactions in Telecommunications Systems*, pages 78–92, 1997.

[26] X. Lin and S. Ghosh. Maxima Finding in a Ring. In *Proc. of 28th Ann. Allerton Conf. on Computers, Communication, and Control*, pages 662–671, 1991.

[27] Z. Liu and M. Joseph. Verification of Fault Tolerance and Real Time. In *Proc. of the 26th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-26)*, pages 220–229. IEEE, June 1996.

[28] J. P. Marques Silva and K. A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48:506–521, 1999.

[29] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

[30] P. M. Melliar-Smith and R. L. Schwartz. Formal specification and mechanical verification of sift: A fault-tolerant flight control system. *IEEE Transaction on Computers*, C-31(7):616–630, July 1982.

[31] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff : Engineering an efficient sat solver. In *Proceedings of 39th Design Automation Conference*, 2001.

[32] M. Nakamura. *Design and evaluation of efficient algorithms for feature interaction detection in telecommunication services*. Ph.D. Dissertation, Osaka University, 1998.

[33] M. Nakamura and T. Kikuno. Feature interaction detection using permutation symmetry. In *Proc. of Fifth Int'l. Workshop on Feature Interactions in Telecommunication Networks and Distributed Systems (FIW'98)*, pages 193–207, 1998.

[34] T. Ohta and Y. Harada. Classification, detection and resolution of service interaction in telecommunication services. In *Proceedings of Second Workshop on Feature Interactions in Telecommunications Systems*, pages 60–72, 1994.

[35] E. Pastor, J. Cortadella, and O. Roig. Symbolic analysis of bounded petri nets. *IEEE Transactions on Computers*, 50(5):432–448, May 2001.

[36] D. A. Plaisted and S. Greenbaum. A structure-preserving clause form translation. *Journal of Symbolic Computation*, 2:293–304, September 1986.

[37] F. Schneider, S. M. Easterbrook, J. R. Callahan, and G. J. Holzmann. Validating Requirements for Fault Tolerant Systems using Model Checking. In *Proc. of International Conference on Requirements Engineering (ICRE)*, pages 4–14. IEEE, April 1998.

[38] M. Sheeran, S. Singh, and G. Stålmarck. Checking safety properties using induction and a sat-solver. In *Proc. of International Conference on Formal Methods in Computer-Aided Design (FMCAD 2000), LNCS 1954*, pages 108–125, 2000.

[39] O. Shtrichman. Pruning techniques for the sat-based bounded model checking problem. In *Proc. of Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME 2001), LNCS 2144*, pages 58–70, 2001.

[40] T. K. Srikanth and S. Toueg. Simulating authenticated broadcast to derive simple fault tolerant algorithms. *Distrib. Computing*, 2(2):80–94, 1987.

[41] T. Tsuchiya, S. Nagano, R. B. Paidi, and T. Kikuno. Symbolic model checking for self-stabilizing algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 12(1):81–95, 2001.

[42] T. Tsuchiya, M. Nakamura, and T. Kikuno. Detecting Feature Interactions in Telecommunication Services with a SAT solver. In *Proc. of 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, 2002.

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