Feature Interaction Verification of Telecommunication Services and Home Network Services Using Model Checking

January 2009

Takafumi Matsuo

Feature Interaction Verification of Telecommunication Services and Home Network Services Using Model Checking

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2009

Takafumi Matsuo

Abstract

Everywhere in our daily life many computer systems, such as telecommunication systems and networked home appliances, are in use. Many new services for these systems are developed to meet various requirements of customers. However, the new services may have conflict with existing services. These conflicts are called feature interaction. Feature interaction can occur when several services are executed at the same time.

In many cases feature interactions cause undesirable behaviors of a system. Hence, to develop new services, it is very important to prevent occurrence of feature interactions. However, detecting feature interactions of concurrent systems is very difficult because such systems have many execution patterns. In addition, the concurrency causes the low repeatability of feature interaction.

Model checking has attracted recent attention as one of the powerful verification techniques to deal with such complex systems. Model checking allows an automatic and exhaustive verification of software and system designs modeled as state machines. The correctness criteria are specified in a temporal logic. When the design fails to meet the correctness criteria, model checking tools can usually produce a counterexample. Using this counterexample, one can easily detect the cause of the error.

This dissertation focuses on model checking and proposes methods for detecting feature interactions of services in two types of systems: telecommunication systems and home network systems.

First, this dissertation proposes a new unbounded model checking method for feature interaction verification for telecommunication systems. To deal with the concurrency of telecommunication systems, we propose to use a new scheme for encoding the behavior of the system and adapt the unbounded model checking algorithm to this encoding. To demonstrate the effectiveness of our approach, we conduct experiments where 21 pairs of telecommunication services are verified using several methods including ours. The results show that our approach exhibits significant speed-up over unbounded model checking using the traditional encoding.

Second, a framework for detecting feature interactions in home network systems is proposed. Our proposed method consists of two steps. In the first step, a model is developed to capture the behavior of the services. In the second step, the model is automatically analyzed to see if possible interactions exist. This automatic analysis can be effectively performed with model checking techniques. The usefulness of the proposed approach is demonstrated through a case study.

List of Major Publications

[1] Takafumi Matsuo, Tatsuhiro Tsuchiya, and Tohru Kikuno, “Feature Interaction Verification Using Unbounded Model Checking with Interpolation,” IEICE Transaction on Information and systems. (conditional acceptance)

[2] Takafumi Matsuo, Pattara Leelaprute, Tatsuhiro Tsuchiya, and Tohru Kikuno, “Verifying Feature interactions in Home Network Systems,” IPSJ Journal, vol. 49, no. 6, pp. 2129-2143, June 2008. (In Japanese)

[3] Takafumi Matsuo, Tatsuhiro Tsuchiya, and Tohru Kikuno, “Safety Verification on Networked Appliance Systems,” The Journal of Reliability Engineering Association of Japan, vol. 30, no. 3, pp. 243-251, May 2008. (In Japanese)

[4] Pattara Leelaprute, Takafumi Matsuo, Tatsuhiro Tsuchiya, and Tohru Kikuno, “Detecting Feature Interactions in Home Appliance Networks,” In Proc. of 9th Int’l Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2008), pp. 895-903, August 2008.

[5] Fuminori Makikawa, Takafumi Matsuo, Tatsuhiro Tsuchiya, and Tohru Kikuno,

"Constructing Overlay Networks with Low Link Costs and Short Paths," In Proc. 6th International Symposium on Network Computing and Applications (NCA 2007), pp. 299-302, July 2007.

[6] Takafumi Matsuo, "A Model of Home Network System for Detecting Feature Interactions by Applying Model Checking," In Supplemental Proceedings of DSN 2007, pp. 300-302, June 2007.

[7] T. Matsuo, P. Leelaprute, T. Tsuchiya, T. Kikuno, M. Nakamura, H. Igaki, and K. Matsumoto,"Automatically Verifying Integrated Services in Home Network Systems", In Proc. of 2006 International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC2006), Vol.2, pp.173-176, July 2006.

[8] Takafumi Matsuo, Tatsuhiro Tsuchiya, and Tohru Kikuno, "Verification of a Distributed Consensus Algorithm Against Safety Properties with Model Checking," In Supplemental Proceedings of DSN 2006, pp. 180-181, June 2006.

Acknowledgments

During the course of this work, I have been fortunate to have received assistance from many individuals. Especially I would like to thank my supervisor Professor Tohru Kikuno for his continuous support, encouragement, and guidance for this work.

I am also very grateful to the members of my dissertation review committee: Professor Takao Onoye and Professor Shinji Kusumoto for their invaluable comments and helpful criticism of this dissertation.

I would like to express my special thanks to the members of my advisory committee: Dr. Yoshiki Kinoshita, Dr. Masayuki Hirayama, and Associate Professor Kiyoharu Hamaguchi for their insightful comments and suggestions of this work.

I would like to express my special thanks to Associate Professor Tatsuhiro Tsuchiya for his continuous assistance and helpful advice.

I am also indebted to Dr. Pattara Leelaprute and Associate Professor Masahide Nakamura of Kobe University for their advice and suggestions.

Finally, I wish to thank many friends in the Graduate School of Information Science and Technology at Osaka University, who gave much help.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

Everywhere in our daily life many computer systems, such as telecommunication systems and networked home appliances, are in use. Many new services for these systems are developed to meet various requirements of customers. However, the new services may have conflict with existing services. These conflicts are called *feature interaction* [10]. Feature interaction can occur when several services are executed at the same time.

In many cases feature interactions cause undesirable behaviors of a system. Hence, to develop new services, it is very important to prevent occurrent of feature interactions. In practical development, however, ad hoc testing is usually conducted to prevent feature interactions. This leads to services which have no interaction-free guarantee.

Many approaches have been proposed to deal with feature interactions. The term “feature interaction problem” in telecommunication systems was proposed

by E.J. Cameron and N. Griffeth of Bellcore in the early 1980s. The first effort to provide a framework for the field of feature interaction had been made by Bowen et. al. [6] in 1989. Since then, much research focusing on feature interactions has been studied by researchers from academia, research centers and industries [2, 5, 8, 12, 15, 16, 26, 45]. Keck and Kuehn surveyed approaches for overcoming feature interactions [24].

Today, feature interaction is not unique to the field of the telecommunication systems. Feature interactions have become problematic in other fields, such as, Web services [48] and building control systems [37]. In the field of home network systems, several studies have been conducted [27, 39].

This dissertation focuses on feature interaction in telecommunication systems and in home network systems. Detection of feature interactions of these concurrent systems is very difficult because such systems have many execution patterns. In addition, the concurrency causes the low repeatability of feature interaction.

Model checking [13] has attracted recent attention as one of the powerful verification techniques to deal with such complexity. Model checking allows an automatic and exhaustive verification of software and system designs modeled as state machines. The correctness criteria are specified in a temporal logic. When the design fails to meet the correctness criteria, model checking tools can usually produce a counterexample. Using this counterexample, one can easily detect the cause of the error.

This dissertation proposes model checking-based methods for detecting feature interactions of services in two types of systems: telecommunication systems and home network systems by using model checking.

1.2 Main Results

1.2.1 Feature Interaction Verification in Telecommunication Systems

As for the first contribution, this dissertation proposes a new unbounded model checking method for feature interaction verification for telecommunication systems. The application of unbounded model checking to asynchronous systems, such as telecommunication systems, has rarely been practiced. This is because, with the conventional encoding the behavior of an asynchronous system can only be represented as a large propositional formula, thus resulting in large computational cost. To overcome this problem we propose to use a new scheme for encoding the behavior of the system and adapt the unbounded model checking algorithm to this encoding. By exploiting the concurrency of an asynchronous system, this encoding scheme allows a very concise formula to represent system's behavior.

To demonstrate the effectiveness of our approach, we conduct experiments where 21 pairs of telecommunication services are verified using several methods including ours. The results show that our approach exhibits significant speed-up over unbounded model checking using the traditional encoding.

1.2.2 Feature Interaction Verification in Home Network Systems

As for the second contribution, this dissertation proposes a framework for detecting feature interactions in home network systems. Our proposed method consists

of two steps. In the first step, a model is developed to capture the behavior of the services and the feature interactions in home network systems are classified based on their causes. In the second step, the model is automatically analyzed to see if possible interactions exist. This automatic analysis can be effectively performed with model checking techniques.

At the first step, we propose a model which consists of four parts: users, services, appliances and environment and classified feature interactions into five types based on their causes. At the second step, we propose a method for translating from our proposed model into Promela, which is the input language of the model checker SPIN. A property that represents the occurrence of each type of feature interactions is also proposed. By using Promela code and properties obtained by our approach, one can detect feature interactions automatically.

The usefulness of the proposed approach is demonstrated through a case study. The result shows that the proposed method can successfully detect any of five types of feature interactions.

1.3 Overview of Dissertation

This dissertation is organized as follows: Chapter 2 describes feature interaction and model checking method. Chapter 3 describes the first contribution, entitled “Feature Interaction Verification with Interpolant-based Unbounded Model Checking.” In this chapter, some examples of practical telecommunication services and feature interactions between those services are described. Next, our proposed encoding is shown and a method for detecting those interactions by using the proposed encoding is also presented. In Chapter 4, entitled “Feature Interac-

tions Verification in Home network system," the second contribution is described. First, this chapter shows examples of home network services and interactions in home network systems. Next, our proposed model and classification of feature interactions are presented. Finally, our proposed method for detecting feature interactions in home network systems are described. Chapter 5 summarizes this dissertation and discusses future work.

Chapter 2

Preliminary

2.1 Feature Interaction

Feature interaction refers to situations where the behavior of different services affect each other. Feature interactions may cause the undesirable behavior of servicesm and thus are considered a very serious problem in developing new services.

2.1.1 Feature Interaction in Telecommunication Systems

In telecommunication systems, many services are provided by modifying basic telecommunication services. For example, the Call Forwarding (CF) service allows the user to forward incoming calls to another address, and the Originating Call Screening (OCS) service restricts outgoing calls according to a screening list. When the new services have conflict with existing services or other new services, feature interactions occur.

We show an example of feature interaction in telecommunication systems. Consider a situation where user A has subscribed to the OCS service and specified

user B in the screening list. User C has activated the CF service to user B. In this situation, if A calls C, the call is forwarded to B by the CF service. As a result, the feature of the OCS service is ignored.

2.1.2 Feature Interaction in Home Network Systems

As home appliances are becoming increasingly interconnected, the use of home network systems is being expanded [38, 47, 41, 17]. Home network systems integrate different features of appliances to provide value-added services. For example, by integrating an air-conditioner, a ventilator and thermometers, one can implement an energy-saving HVAC (heating, ventilation and air-conditioning) service. Another example could be an air-cleaning service which automatically cleans the room air by controlling a ventilator and a smoke sensor.

Here, an example of feature interactions in home network systems is shown. Assume that the HVAC service is operating the air-conditioner to warm up the room temperature and that at the same time the air-cleaning service is using the ventilator to clean the room air. If the room temperature is higher than the outside temperature, then cool outside air is taken into the room by the ventilator, which results in the low efficiency of the HVAC service.

In home network systems, the “physical” environment is an important factor to deal with feature interactions. The change of the environment is not explicit as the change of the state of the appliances. For example, the temperature of room does not immediately reach the value of the temperature setting of an air-conditioner just after the air-conditioner is turned on. In dealing with the feature interaction problem for home network systems, such a property of the environment must be taken into consideration.

2.2 Model Checking

Model checking is a technique for verifying state transition systems. Model checking explores the state space to determine whether or not a given property holds in the system. This method allows an automatic and exhaustive verification of software and system designs. The correctness criteria are specified in a temporal logic. When the design fails to meet the correctness criteria, model checking tools can usually produce a counterexample. Using this counterexample, one can easily detect the cause of the error.

For realistic systems, however, the number of states of the system model can be very large, making the model checking problem intractable. This problem is called the state explosion problem. This problem is one of the most serious problems with model checking. To deal with this problem, many methods have been proposed.

2.2.1 Unbounded Model Checking

One of the approaches to the state explosion problem is *bounded model checking* [4, 14]. The main idea of bounded model checking is to look for counterexamples that are shorter than some fixed length k for a given property. This limitation allows one to reduce the model checking problem to the satisfiability (SAT) checking problem for a formula of some logic such that its satisfiability implies the existence of a counterexample. Thus if the formula turns out to be satisfiable, then it is possible to conclude that the violation of the property occurs in the system.

Although effective in detecting property violation, bounded model checking

cannot be directly used for proving the absence of violation. To cope with this disadvantage, McMillan proposed unbounded model checking [36], which combines bounded model checking and *interpolation*. In the field of hardware verification, unbounded model checking has been successful in verifying the properties of the circuits that cannot be verified by other model checking methods [23].

The key observation used in McMillan's method is that when bounded model checking fails to find a counterexample, in which case the formula is unsatisfiable, an over-approximation of the state set reachable in one step can be derived from the unsatisfiability proof produced by the SAT solver. Technically this over-approximation is obtained in the form of an *interpolant* of the tested formula, using an interpolation procedure. By repeatedly executing the interpolant procedure, an over-approximation of the reachable state set can be obtained. If this over-approximation contains no state violating a given property, then it is ensured that the system meets that property.

2.2.2 SPIN

The SPIN model checker [21] is a verification tool for concurrent systems. In this tool the partial order reduction [42] is used to reduce the state space to be checked. To use SPIN, the behavior of a system needs to be described in the Promela language, the input language of SPIN. Properties to be verified are represented as Linear-Time Temporal Logic (LTL) [43].

In a Promela program, a system is defined as a collection of processes which run asynchronously. These processes communicate via buffered channels and shared global variables. Each process consists of a sequence of local variable declarations, message channel declarations and statements.

An LTL formula represents properties about the execution traces of the Promela program, where a trace is sequence of states. The model checker determines whether or not all traces starting with the initial state satisfies a given LTL formula. LTL formulas are translated into processes called never-claims in Promela. The never-claim processes are equivalent to Büchi Automata [7]. Such processes represents undesirable behavior of the system.

Chapter 3

Feature Interaction Verification in Telecommunication Systems

3.1 Introduction

This chapter proposes a method for verifying feature interactions in telecommunication systems [34]. This method uses *unbounded model checking* with interpolant.

The application of unbounded model checking to asynchronous systems has rarely been practiced. Indeed we are not aware of any application to telecommunication systems. This can be explained by the fact that with the conventional encoding, the behavior of an asynchronous system can only be represented as a large formula, thus resulting in large computational cost.

In our proposed method, we use a new scheme for encoding the behavior of the system. By exploiting the concurrency of the telecommunication system, this encoding scheme allows a very concise representation of system's behavior.

By adapting unbounded model checking to this encoding, we obtain our model checking method. The effectiveness of our approach is demonstrated through experiments.

Previous attempts to improve the performance of unbounded model checking include, for example, [30, 1, 46]. In [30], the method of reusing interpolants is proposed to efficiently obtain an over-approximation of the reachable state set. In [1], hybridization of interpolation and abstraction refinement is studied. In [46], a new interpolation algorithm which is based on linear programming is proposed. These studies aim to improve the interpolation procedure but do not focus on the representation of the behavior of the system. The central idea behind our encoding can also be seen in [49, 40]. In [40] a similar encoding is proposed in the context of the verification of safe Petri nets. The encoding proposed in [49] is used to represent telecommunication systems, as is done in this chapter. However transition ordering, which will be explained in Section 3.3.2 is not applied in the encoding of [49]. More importantly, in contrast to this chapter where unbounded model checking is discussed, these early attempts only deal with bounded model checking.

The rest of this chapter is organized as follows: In Section 3.2, examples of services and feature interaction in telecommunication systems are shown. Section 3.3 shows a new scheme for encoding the behavior of the system. Next, a method for adapting unbounded model checking to this encoding is described. In Section 3.4, the effectiveness of our approach is demonstrated through experiments. Finally, Section 3.5 summarizes this chapter.

3.2 Telecommunication Services

3.2.1 Examples of Services

In this chapter we consider seven telecommunication services taken from ITU-U recommendation [22] and Bellcore's feature standard [3].

Call Waiting (CW): This service allows the subscriber to receive a second incoming call while he or she is already talking.

Call Forwarding (CF): This service allows the subscriber to have his or her incoming calls forwarded to another address.

Originating Call Screening (OCS): This service allows the subscriber to specify that outgoing calls be either restricted or allowed according to a screening list.

Terminating Call Screening (TCS): This service allows the subscriber to specify that incoming calls be either restricted or allowed according to a screening list.

Denied Origination (DO): This service allows the subscriber to disable any call originating from the terminal. Only terminating calls are permitted.

Denied Termination (DT): This service allows the subscriber to disable any call terminating at the terminal. Only originating calls are permitted.

Directed 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 off-hooking, x is directly calling y. It is not necessary for x to dial y.

3.2.2 Feature Interaction

Two types of feature interaction are considered. The freedom from these types of interaction can be viewed as safety properties. In order to detect these types of interactions, it suffices to check the reachability the initial state to undesirable states where feature interaction occurs.

Invariant Violation

It is usually the case that services require some specific properties to be satisfied at any time. For example, the OCS service requires 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*. However, combining multiple services can result in violation of this property. Consider a situation where user A has subscribed to OCS service and specified user B in the screening list while user C has activated CF service to B. In this situation, if A calls C, the call is forwarded to B by the CF service. As a result, the invariant property of OCS is violated.

Nondeterminism

Nondeterminism is one of the best known types of feature interactions [18, 25]. Nondeterminism refers to a situation where a single 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. For example, this type of interaction can occur between the CW service and the CF service. Suppose that user A subscribes both services. Now consider the situation where A is talking with user B while user C is in A's forwarding address list. If user D dials A, then

either the call from D to A may be received by A because of CW, or the call may be forwarded to C because of CF.

3.2.3 System Model

We use State Transition Rules (STR) [20] to describe services and to model the behavior of the system. A *service* is defined as a 6-tuple $\langle U, V, P, E, R, s_{init} \rangle$, where U is a finite set of service users, V is a finite set of variables, P is a set of predicates, E is a finite set of events, R is a finite set of rules, and s_{init} is the initial state. A predicate $p \in P$ is of the form $p(x_1, x_2, \dots)$ where $x_i \in V$. An event $e \in E$ is of the form $e(x_1, x_2, \dots)$ where $x_i \in V$. A rule $r \in R$ is of the form:

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

The pre-condition is a set of predicates or negations of predicates, or both, while the post-condition is a set of predicates.

Figure 3.1 shows an example of a service specification expressed in STR. This specification describes the Plain Old Telephone Service (POTS). Additional communication features can be described by modifying this specification (for example, by adding new rules).

A predicate (or an event or a rule) is *instantiated* by substituting a user $a \in U$ for each variable $x \in V$ occurring in the predicate (event or rule, respectively) such that no two variables are substituted by the same user. That is, given a predicate $p(x_1, x_2, \dots,) \in P$ and a substitution $\theta = \langle x_1 | a_1, x_2 | a_2, \dots \rangle$, $\forall i, j, i \neq j$: $a_i \neq a_j$, we have a predicate instance $p(a_1, a_2, \dots)$. An event instance or a rule instance is defined similarly. We let $\mathcal{P} = \{p_1, \dots, p_m\}$ denote the set of all predicate instances and m denote the number of the predicate instances (i.e. $m = |\mathcal{P}|$).

U=$\{A, B\}$

V=$\{x, y\}$

P=$\{idle(x), dialtone(x), busytone(x),$

$calling(x, y), path(x, y)\}$

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

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 3.1: Rule-based specification for POTS

Also we denote by $\mathcal{R} = \{t_1, t_2, \cdots, t_n\}$ the set of all rule instances and by n the number of the rule instances (i.e. $n = |\mathcal{R}|$).

A state is defined as a set of predicate instances and is regarded as representing the predicate instances that hold in that state. We denote by S the set of states, that is, $S = 2^{\mathcal{P}}$.

The execution semantics is as follows. For a rule instance $t \in \mathcal{R}$, let $Pre[t]$ denote the set of predicate instances in the pre-condition of t and $\hat{P}re[t]$ denote the set of predicate instances whose negations are in the pre-condition. Also let $Post[t]$ denote the set of the predicate instances in the post-condition of t and $e[t]$ denote the event instance of t. t is *enabled* for $e[t]$ in a state s iff all instances in $Pre[t]$ hold and no instance in $\hat{P}re[t]$ hold in s; that is, $Pre[t] \subseteq s$ and $\hat{P}re[t] \cap s = \emptyset$. Exactly one enabled rule instance is selected for execution at a time. The execution of an enabled rule t causes a state transition from s to the next state s', by deleting all instances in $Pre[t]$ from s and adding all instances in $Post[t]$; that is $s' = (s \setminus Pre[t]) \cup Post[t]$.

Example 1 Consider the specification of POTS in Figure 3.1. Let t be the rule instance of $r = pots1$ based on $\theta = \langle x|A\rangle$. Then $Pre[t] = \{idle(A)\}$, $\hat{P}re[t] = \emptyset$, and $Post[t] = \{dialtone(A)\}$. In state $s = \{idle(A), idle(B)\}$ t is enabled for event $offhook(A)$ — the event that subscriber A picks up the phone. If t is executed in s, then a transition to state $s' = \{dialtone(A), idle(B)\}$ occurs.

In general a state transition system is represented as (S, T, I) where S is the set of states, $T \subseteq S \times S$ is the transition relation, and $I \subseteq S$ is the set of initial states. Now, for each rule instance t, let $T_t \subseteq S \times S$ be the relation over states such that $(s, s') \in T_t$ iff t is enabled in s for some event instance and its execution

causes a state transition to s'. The state transition system (S, T, I) defined by an STR specification $SS = \langle U, V, P, E, R, s_{init} \rangle$ is such that $T = \bigcup_{t \in \mathcal{R}} T_t$ and $I = \{s_{init}\}$. We denote by $s \xrightarrow{t} s'$ iff $(s, s') \in T_t$.

3.3 Proposed Method

3.3.1 Symbolic Representation

We propose a propositional SAT-based unbounded model checking method. In order to use propositional SAT solvers for model checking, it is essential to encode the state space and the transition relation with Boolean variables.

Recall that $\mathcal{P} = \{p_1, \ldots, p_m\}$ is the set of predicate instances. A state can be represented as a Boolean m-vector such that it has a $true$ in the ith position iff p_i holds in that state. In the following of the chapter, we represent states with m Boolean variables $s = (b_1, \ldots, b_m)$; that is, a state is a truth assignment of these m variables.

Any set $S_0 \subseteq S$ of states can be represented as a Boolean function $f : \{true, false\}^m \rightarrow \{true, false\}$ such that:

$$f(s) = \begin{cases} true & s \in S_0 \\ false & otherwise \end{cases}$$

For example, the state set where the pre-condition of a rule instance t holds is represented as:

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

The relation over states is also represented as a Boolean function with $2m$ Boolean variables, since the relation is simply a set of state pairs. We therefore identify a set of states or of transitions with its corresponding Boolean function. For example, T_t is represented as:

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

where $s = (b_1, \cdots, b_m)$ and $s' = (b_1', \cdots b_m')$. The operator $\leftrightarrow$ means that two operands have same value.

Example 2 For simplicity, we use the name of a predicate instance to denote its corresponding Boolean variable. Let t be the instance of the rule $pots1$ in Figure 3.1 with substitution $\theta = \langle x|A \rangle$. Then we have $T_t(s,s') = idle(A) \wedge dialtone(A)' \wedge \neg idle(A)' \wedge (idle(B) \leftrightarrow idle(B)') \wedge (dialtone(B) \leftrightarrow dialtone(B)') \wedge (busytone(A) \leftrightarrow busytone(A)') \wedge (busytone(B) \leftrightarrow busytone(B)') \wedge (calling\text{-}(A,B) \leftrightarrow calling(A,B)') \wedge (calling(B,A) \leftrightarrow calling(B,A)') \wedge (path(A,B) \leftrightarrow path(A,B)') \wedge (path(B,A) \leftrightarrow path(B,A)')$.

The transition relation T is represented as:

$$T(s,s') = \bigvee_{t \in \mathcal{R}} T_t(s,s')$$

For simplicity, assume that we know that T is a *total* relation [13]. Then whether state set G is reachable from the initial state in k steps can be determined by checking the satisfiability of the following formula:

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

This is the basic formula used in SAT-based model checking.

3.3.2 Boolean Encoding of Telecommunication Systems

The obstacle to applying SAT-based model checking to asynchronous systems like telecommunication systems is that the transition relation T for such a system can only be represented as a large formula. This can be understood by seeing Example 2: To represent the transitions for each rule instance t, T_t must contain conjuncts $(b_i \leftrightarrow b_i')$ for all Boolean variables b_i that represent predicate instances not engaged in that rule. Our encoding overcomes this disadvantage.

The intuitive idea is as follows: We introduce a new semantics for system execution that maintains safety properties of the original model. In this semantics the n rule instances are totally ordered and a step is represented as a sequence of n "micro" steps. The ith micro step is either the state transition by the ith rule instance or a stuttering step. Because only two state transitions are possible at each micro step, this semantics can avoid a blow-up in the formula size, which is inherent to symbolic representation of asynchronous systems.

Let $Chng[t]$ denote the set of predicate instances that change their truth value as a result of the execution of rule instance t; that is, $Chng[t] = (Post[t] \setminus Pre[t]) \cup (Pre[t] \setminus Post[t])$. Our encoding can avoid generating a large subformula related to $\mathcal{P} \setminus Chng[t]$.

Now let $d_t(s, s')$ be defined as follows:

$$
\begin{aligned}
d_t(s,s') &= T_t(s,s') \vee \bigwedge_{p_i \in \mathcal{P}} (b_i \leftrightarrow b_i') \\
&= \Big(\big(\bigwedge_{p_i \in Pre[t]} b_i \wedge \bigwedge_{p_i \in \hat{P}re[t]} \neg b_i \wedge \bigwedge_{p_i \in Post[t] \setminus Pre[t]} b_i' \wedge \bigwedge_{p_i \in Pre[t] \setminus Post[t]} \neg b_i' \big) \\
&\quad \vee \bigwedge_{p_i \in Chng[t]} (b_i \leftrightarrow b_i') \Big) \\
&\quad \wedge \bigwedge_{p_i \in \mathcal{P} \setminus Chng[t]} (b_i \leftrightarrow b_i')
\end{aligned}
$$

Example 3 Let t be the rule instance of $pots1$ in Figure 3.1 with substitution $\theta = \langle x|A \rangle$. Then $Chng[t] = \{idle(A), dialtone(A)\}$ and thus we have $d_t(s,s') = \Big(\big(idle(A) \wedge dialtone(A)' \wedge \neg idle(A)' \big) \vee \big((idle(A) \leftrightarrow idle(A)') \wedge (dialtone(A) \leftrightarrow dialtone(A)') \big) \Big) \wedge (idle(B) \leftrightarrow idle(B)') \wedge (dialtone(B) \leftrightarrow dialtone(B)') \wedge (busytone(A) \leftrightarrow busytone(A)') \wedge (busytone(B) \leftrightarrow busytone(B)') \wedge (calling\text{-}(A,B) \leftrightarrow calling(A,B)') \wedge (calling(B,A) \leftrightarrow calling(B,A)') \wedge (path(A,B) \leftrightarrow path(A,B)') \wedge (path(B,A) \leftrightarrow path(B,A)')$.

By definition $d_t(s,s') = true$ iff $s \xrightarrow{t} s'$ or $s = s'$. Using this property, a step (or more) can be represented by a conjunction of $d_t(s,s')$ as follows:

$$D(s_0, \cdots, s_n) = \bigwedge_{1 \le i \le n} d_{t_i}(s_{i-1}, s_i)$$

$D(s_0, \cdots, s_n)$ evaluates to true iff any $1 \le i \le n$, $s_{i-1} \xrightarrow{t_i} s_i$ or $s_{i-1} = s_i$. This means that if this function 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 $s \xrightarrow{t_i} s'$, $D(s_0, \cdots, s_n)$ evaluates to true under an assignment such that $s = s_0 = \cdots = s_{i-1}$, $s_{i-1} \xrightarrow{t_i} s_i$, and $s_i = \cdots = s_n = s'$.

Consequently, the following formula BMC^k can be used for the verification.

$$BMC^k \quad = \quad I(s_0) \quad \wedge \quad \bigwedge_{0 \leq j < k} D(s_{j*n}, \cdots, s_{(j+1)*n}) \quad \wedge \quad G(s_{k*n})$$

If BMC^k is satisfiable, then some state in G is reachable from the initial state in at most $k * n$ steps. If BMC^k is unsatisfiable, then no state in G can be reached from the initial state in k steps.

A major benefit of using this formula is that it can be shortened to a considerable extent and thus in turn the run time of SAT solving can be reduced. The idea is as follows. Note that in BMC^k, term $(b_i \leftrightarrow b_i')$ for any $p_i \in \mathcal{P} \setminus Chng[t]$ occurs as a conjunct (See the definition of $d_t(s, s')$). Replacing b_i' with b_i, such a term can safely be removed from BMC^k without altering the satisfiability, because BMC^k is satisfiable only if b_i and b_i' have the same truth value.

The effect of this optimization is significant, since for practical telecommunication services, a rule execution affects only a small fraction of the predicate instances. Compared to the conventional formula shown in Section 3.3.1, a reduction of around 60 to 90 percent in the number of literal occurrences has typically been observed in the examples tested in Section 3.4.

Remark 1 For presentation purpose we explain our model checking method using the original BMC^k; but this optimization is always used in the implementation.

Transition Ordering

In practice, the state transitions represented by D critically depend on the order of rule instances. This can be intuitively explained as follows: Consider two rule

instances t_i and t_j and suppose that the execution of t_i cause the precondition of t_j to hold but not vice versa. In this case, if t_i occurs before t_j in D, then D can represent the successive execution of t_i and t_j. On the other hand, if t_i comes after t_j, then D can only express the execution of only one of the two rule instances.

We propose a heuristic algorithm for transition ordering, by extending the one in [40], which is proposed in the context of safe Petri nets. The basic idea is to select a rule instance t when each predicate instance in $Pre[t]$ occurs in s_{init} or in the post-condition of an already selected rule instance. Figure 3.2 shows the algorithm.

The FIFO queue $Done$ is used for storing rule instances that have already been ordered. The set $Checked$ of predicate instances is used to maintain those occurring in the initial state or in the post-condition of rule instances ordered already. The main part of the algorithm calls procedure CHECK with each predicate instance occurring in the initial state. In the procedure, first p is added to $Checked$. Then, for each rule instance t such that $p \in Pre[t]$, the following is done: If t has not yet been ordered and all predicate instances in $Pre[t]$ have been stored in $Checked$, then t is enqueued and the procedure is recursively called with each predicate instance in $Post[t]$. Since a rule instance t is ordered only after all predicate instances in $Pre[t]$ are stored in $Checked$, for any predicate instance p in $Pre[t]$, it occurs in the initial state or there must be at least one already ordered rule instance t' such that $p \in Post[t']$.

In case a given service specification is ill-formed, this algorithm may fail to order all rule instances. It is easy to show that in such a case, the rule instances not selected for ordering are always unenabled, and thus they can safely be omitted.

Example 4 Consider the service specification in Figure 3.1 and let $\theta 1 = \langle x|A, y|B\rangle$

```
1: set Checked := ∅ ; {Keeps predicate instances.}
2: queue Done := ∅ ; {Keeps ordered rule instances.}
3: for all p ∈ s_init do
4:    call CHECK(p);
5: end for
6:
7: Procedure: CHECK(p)
8:    add p to Checked;
9:    for all t such that p ∈ Pre[t] do
10:      if t ∉ Done and ∀p' ∈ Pre[t][p' ∈ Checked] then
11:         enqueue t to Done;
12:         for all p' such that p' ∈ Post[t] and p' ∉ Checked do
13:            call CHECK(p');
14:         end for
15:      end if
16:   end for
17: end Procedure
```

Figure 3.2: Algorithm for ordering transitions

and $\theta 2 = \langle x|B, y|A\rangle$. The algorithm orders a total of 18 rule instances as follows: $(t_1, \cdots, t_{18})$ = ($pots1\theta 1$, $pots1\theta 2$, $pots2\theta 1$, $pots2\theta 2$, $pots3\theta 1$, $pots3\theta 2$, $pots4\theta 1$, $pots4\theta 2$, $pots5\theta 1$, $pots5\theta 2$, $pots6\theta 1$, $pots6\theta 2$, $pots7\theta 1$, $pots7\theta 2$, $pots8\theta 1$, $pots8\theta 2$, $pots9\theta 1$, $pots9\theta 2$), where $r\theta$ denotes the instance of a rule r with a substitution θ. With this ordering, BMC^1 allows reachability checking for 12 states, including state $\{path(A, B), path(B, A)\}$ which requires $t_1 = pots1\theta 1$, $t_5 = pots3\theta 1$, and $t_{11} = pots6\theta 1$ to occur in this order to be reached. On the other hand, if the rule instances were reversely ordered, the efficiency would be much deteriorated. In this case BMC^1 can check the reachability of only four states: $\{idle(A), idle(B)\}$, $\{dialtone(A), idle(B)\}$, $\{idle(A), dialtone(B)\}$, and $\{dialtone(A), dialtone(B)\}$.

3.3.3 Unbounded Model Checking

State Exploration Using Interpolants

For two first-order logic formulas A and B, if $A \wedge B$ is unsatisfiable, then the interpolant P for A and B is a formula with the following properties:

- $A \rightarrow P$,
- $P \wedge B$ is unsatisfiable, and
- P refers only to the common variables of A and B.

Several interpolation methods have been proposed, including [36, 44, 46]. Using an interpolation method with a SAT solver, one can simultaneously perform satisfiability checking and, if the formula is unsatisfiable, interpolant generation.

We divide BMC^k into $PREF$ and $SUFF^k$ as follows:

$$\begin{aligned} PREF &= I(s_0) \wedge D(s_0, \cdots, s_n) \\ SUFF^k &= \bigwedge_{1 \leq i < k} D(s_{i*n}, \cdots, s_{(i+1)*n}) \wedge G(s_{k*n}) \end{aligned}$$

If $PREF \wedge SUFF^k$ is satisfiable, then the system can reach a state in G. On the other hand, if $PREF \wedge SUFF^k$ is unsatisfiable, then an interpolant of $PREF$ and $SUFF^k$ can be generated.

The interpolant refers only to the common variables of $PREF$ and $SUFF^k$. Hence, it is a formula over state s_n. We denote by $Interpolant(s_n)$ this interpolant and by $Interpolant(s_n)\langle s_n|s\rangle$ the formula obtained from $Interpolant(s_n)$ by replacing Boolean variables for s_n with those for s.

Since the interpolant is implied by $PREF$, it follows that the $Interpolant(s_n)$ $\langle s_n|s\rangle$ is true in the initial state and in every state reachable from the initial state in one step. In other words, $Interpolant(s_n)\langle s_n|s\rangle$ is an over-approximation of the state set reachable from the initial state within one step.

In effect this interpolant usually contains more reachable states than those reachable in one step, because D can represent, in addition to all single steps, up to n consecutive steps. This property contributes to effective state exploration of our method.

Overview of the Algorithm

The overview of the algorithm is shown in Figure 3.3. The input of the algorithm is a service specification $SS = \langle U, V, P, E, R, s_{init}\rangle$ and the state set G whose reachability is to be verified.

```
1: Construct M = ⟨I, D, G⟩ from SS and G;
2: while true do
3:    k := 2;
4:    FINITERUN(M, k);
5:    if aborted then
6:       k := k + 1;
7:    else if true is returned then
8:       return Reachable;
9:    else if false then
10:      return Unreachable;
11:   end if
12: end while
```

Figure 3.3: Unbounded model checking for given (SS,G)

First, we build the symbolic representations of the transition system $M = \langle I, D, G \rangle$ from the given service specification $SS = \langle U, V, P, E, R, s_{init} \rangle$.

The function FINITERUN is at the heart of the algorithm. It has two arguments M and k. The function returns true if it determines, by performing SAT solving for BMC^k, that a state in G is reachable and returns false if it determines that no state in G is reachable. In these cases, the algorithm can terminate simply by returning Reachable or Unreachable, according to the result of FINITERUN. FINITERUN aborts if it is impossible to determine whether or not G is reachable by using given k. In this case, the algorithm increases k and calls FINITERUN again.

The function FINITERUN

The function FINITERUN is shown in Figure 3.4. The basic design of this function follows that of [36] but is adapted to subtle but important differences in the encoding of system behaviors, elaborated in Section 3.3.2.

This function first checks the satisfiability of $PREF \wedge SUFF^k$. If $PREF \wedge SUFF^k$ is satisfiable, then at least one of the states in G is reachable from the initial state (line 9). Hence, this function returns true. On the other hand, if $PREF \wedge SUFF^k$ is unsatisfiable, then state exploration is performed by repeatedly computing interpolants.

In the function R is used to represent the set of explored states. Initially R only represents the initial state. In each iteration of the while loop, R is updated to the interpolant for $PREF$ and $SUFF^k$ (line 19) and $PREF$ is updated with I being replaced with R (line 5). At the end of the ith execution of the while loop, R represents an over-approximation of a set of states that are reachable within i steps.

As discussed in Section 3.3.3 in detail, the iteration of the while loop eventually terminates (or aborts) in either of two ways. The first case is where $PREF \wedge SUFF^k$ turns out to be satisfiable. Then the function aborts (line 11), since the satisfiability of $PREF \wedge SUFF^k$ only means the reachability of G from R which may contain unreachable states.

The second case is where R reaches a fixed point — the point from which R will never grow further. At this point R contains all reachable states and thus the unreachability of G is immediately concluded from the unsatisfiability of $PREF \wedge SUFF^k$. If this happens, the function terminates by returning false (line 17).

1: **Function:** FINITERUN($M = \langle I, D, G \rangle, k > 1$)
2: $R := I$;
3: **while true do**
4: $M' := \langle R, D, G \rangle$;
5: Generate $PREF, SUFF^k$ from M'
6: Run SAT on $PREF \wedge SUFF^k$;
7: **if** satisfiable **then**
8: **if** $R = I$ **then**
9: return **true**; {Can be reached only in the 1st iteration.}
10: **else**
11: abort;
12: **end if**
13: **else**
14: Generate interpolant($Interpolant(s_n)$) of $PREF, SUFF^k$;
15: $R' := Interpolant(s_n)\langle s_n/s_0 \rangle$;
16: **if** $R' = R$ **then**
17: return **false**;
18: **else**
19: $R := R'$;
20: **end if**
21: **end if**
22: **end while**
23: **end Function**

Figure 3.4: Function FINITERUN for given (M, k)

Correctness of the Algorithm

Lemma 1 FINITERUN(M, k) terminates for every (M, k).

Proof: First, suppose that G is reachable from the initial state. If BMC^k is satisfiable, then the function terminates by returning true. If BMC^k is unsatisfiable, then the function proceeds to the iterative generation of interpolants. In this iterative process, R gradually increases until it reaches a fixed point. Such a fixed point must exist since the state space is finite and contains all reachable states. Thus R always grows to the extent where BMC^k is satisfiable, in which case the function aborts (at line 11) since $R \neq I$.

Next, suppose G is unreachable from the initial state. Since BMC^k is unsatisfiable at the first time (at line 6), the function tries to compute the over-approximate set of states reachable from the initial state. During this computation, if BMC^k becomes satisfiable, then the function aborts since $R \neq I$. Otherwise, the process of computing the over-approximation of the reachable state set terminates because the state space is finite. In this case, $R = R'$ holds at line 16 and thus false is returned (at line 17).

Lemma 2 For every M, there exists k such that FINITERUN(M, k) terminates without aborting.

Proof: If G is reachable, then the result of the first run of SAT must be "satisfiable" when $k = |S| - 1$, in which case the function returns true and terminates.

Now suppose that G is unreachable and let k_G be the maximum length (the number of transitions) of the shortest path from any state in S to any state in G. Of course such a path only contains unreachable states. Let k be $k_G + 1$. Then $PREF \wedge SUFF^k$ is always unsatisfiable in any run of SAT. This can be explained by showing that R never contains unreachable states. In the first iteration,

this trivially holds since $R = I$. In any later iteration, R is an interpolant for $PREF$ and $SUFF^k$, and thus $R(s_n) \wedge SUFF^k$ is unsatisfiable. This implies that R contains no unreachable state, because if an unreachable state existed in R, then $R(s_n) \wedge SUFF^k$ would be satisfiable since G can be reached within $k_G = k - 1$ steps from that state. Since the number of states of the system is finite, R eventually reaches a fixed point, at which time the function returns false and terminates.

Lemma 3 When FINITERUN(M, k) terminates, it returns true if G is reachable and false if G is unreachable.

Proof: First suppose the function returns true. This means that BMC^k (at line 6) is satisfiable; hence G is reachable from initial state. Next suppose that the function returns false. In this case, R represents an over-approximation of the set of reachable states and $PREF \wedge SUFF^k$ is unsatisfiable with that R. Since any reachable state is contained in R, the fact that $PREF \wedge SUFF^k$ is unsatisfiable implies that no state in G is reachable.

Theorem 1 The algorithm shown in Figure 3.3 terminates. It returns Reachable if G is reachable from the initial state; it returns Unreachable, otherwise.

Proof: The proof straightforwardly follows from Lemmas 1, 2 and 3.

3.4 Experiment Results

To evaluate the effectiveness of our proposed method, we conducted experiments. We verified 21 pairs of telecommunication services described in Section 3.2.1 using the proposed unbounded model checking method, McMillan's unbounded model checking method [36] and the model checker SPIN [21]. We implemented

these two unbounded model checking methods using McMillan's FOCI tool for both SAT solving and interpolant generation. In the proposed unbounded model checking method, the new encoding and the proposed algorithm are used. On the other hand, McMillan's method uses conventional encoding with T and the interpolant procedure proposed in [36].

SPIN, a very well-known model checker, uses explicit state representation, in the sense that it does not employ Boolean state space encoding. We construct Promela models for telecommunication services as follows. We represent each predicate instance by a Boolean variable. We use a single Promela process to represent the behavior of the whole system. The Promela process has, in turn, a single big `do` statement, in which every rule instance is represented as a guarded command. At any point of time, a single guarded command whose guard is true is nondeterministically selected for execution.

The experiments were performed on a Linux (kernel 2.4) PC with a 2.8 GHz CPU and about 3 Gbyte memory.

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 DO, x never receives dialtone at any time. ($\neg DO(x) \vee \neg dialtone(x)$)

DT: If x subscribes DT, y is never calling x at any time. ($\neg DT(x) \vee \neg calling(y, x)$)

Consequently a total of 18 pairs that contain at least one of the four services are verified against these invariant properties. Because of the symmetry of users, we check the violation with a single variable substitution by users. For example, the invariant of OCS is verified by checking reachability to $G(s) = \neg(\neg OCS(A,B) \vee \neg calling(A,B))$, where $OCS(A,B)$ and $calling(A,B)$ are Boolean variables representing predicate instances $OCS(A,B)$ and $calling(A,B)$.

Nondeterminism occurs in states where two rules simultaneously become enabled for the same event. Due to user symmetry, it suffices to check all event instances obtained from any single variable substitution θ_0 of users. Thus we let:

$$G(s) = \bigvee_{e\theta_0 : e \in E} \bigvee_{\substack{t_1, t_2 \in \mathcal{R}(t_1 \neq t_2): \\ e[t_1]=e[t_2]=e\theta_0}} E_{t_1}(s) \wedge E_{t_2}(s)$$

Here $e\theta_0$ is an event instance obtained by the substitution θ_0 and t_1 and t_2 are two different rule instances that have the same event instance $e\theta_0$. For example, consider the specification shown in Figure 3.1 and let $\theta_0 = \langle x|A, y|B\rangle$. Three event instances, namely, $onhook(A)$, $offhook(A)$ and $dial(A,B)$, are shared by more than one rule instance. Specifically, $onhook(A)$ triggers $pots2\theta_0$, $pots5\theta_0$, $pots7\theta_0$ and $pots8\theta_0$, $offhook(A)$ triggers $pots1\theta_0$ and $pots6\theta_0$, and $dial(A,B)$ triggers $pots3\theta_0$ and $pots4\theta_0$. Hence we have $G(s) = \Big(\big(E_{pots2\theta_0}(s) \wedge E_{pots5\theta_0}(s)\big) \vee \big(E_{pots2\theta_0}(s) \wedge E_{pots7\theta_0}(s)\big) \vee \big(E_{pots2\theta_0}(s) \wedge E_{pots8\theta_0}(s)\big) \vee \big(E_{pots5\theta_0}(s) \wedge E_{pots7\theta_0}(s)\big) \vee \big(E_{pots5\theta_0}(s) \wedge E_{pots8\theta_0}(s)\big) \vee \big(E_{pots7\theta_0}(s) \wedge E_{pots8\theta_0}(s)\big)\Big) \vee \Big(E_{pots1\theta_0}(s) \wedge E_{pots6\theta_0}(s)\Big) \vee \Big(E_{pots3\theta_0}(s) \wedge E_{pots4\theta_0}(s)\Big)$.

Table 3.1 shows the results of the verification of the violation of invariant properties, while Table 3.2 shows the results of the verification of nondeterminism. The two leftmost columns represent the combination of services tested and whether feature interaction occurs in that combination.

Table 3.1: Verification result of violation of invariant

		Proposed method		McMillan's method		SPIN
Service	Interaction	time(s)	(k,r)	time(s)	(k,r)	time(s)
CW + DO		NA (2754.0)	(4,3)	NA (15311.3)	(11,3)	1.5
CW + DT	√	0.8	(2,0)	6595.3	(10,0)	1.4
CW + OCS	√	0.7	(2,0)	3819.6	(10,0)	1.6
CW + TCS	√	0.7	(2,0)	8130.8	(10,0)	1.6
CF + DO		NA (1346.5)	(4,7)	NA (4973.5)	(7,4)	1.5
CF + DT	√	0.6	(2,0)	19.6	(6,0)	1.2
CF + OCS	√	0.5	(2,0)	49.9	(6,0)	1.3
CF + TCS	√	0.5	(2,0)	37.7	(6,0)	1.2
DC + DO		6.6	(3,3)	NA (4266.9)	(8,5)	0.9
DC + DT		0.4	(2,2)	0.7	(2,2)	0.9
DC + OCS	√	0.3	(2,0)	1.3	(3,0)	0.9
DC + TCS	√	0.3	(2,0)	1.3	(3,0)	1.0
DO + DT		1149.4	(4,5)	NA (8892.3)	(8,4)	0.8
DO + OCS		1023.4	(4,6)	NA (6037.8)	(7,4)	0.9
DO + TCS		257.1	(3,6)	NA (1249.2)	(7,4)	0.9
DT + OCS		16.1	(3,5)	NA (29077.8)	(7,4)	1.0
DT + TCS		14.0	(3,4)	NA (13662.3)	(8,4)	1.0
OCS + TCS [B]	√	0.3	(2,0)	1.5	(3,0)	0.9

Table 3.2: Verification result of nondeterminism

		Proposed method		McMillan's method		SPIN
Service	Interaction	time(s)	(k,r)	time(s)	(k,r)	time(s)
CW + CF	√	3.9	(2,0)	NA (1365.9)	(6,2)	2.1
CW + DC		NA (12326.5)	(4,2)	NA (4776.5)	(7,1)	1.9
CW + DO		NA (6512.2)	(5,1)	NA (4453.4)	(8,1)	1.6
CW + DT	√	3.2	(2,0)	1004.3	(8,0)	1.6
CW + OCS	√	2.0	(2,0)	3049.8	(8,0)	1.8
CW + TCS	√	2.1	(2,0)	3910.2	(8,0)	1.7
CF + DC		NA (1033.9)	(2,4)	NA (7664.3)	(7,2)	2.3
CF + DO		NA (1346.5)	(4,7)	NA (8177.8)	(8,2)	1.5
CF + DT	√	0.7	(2,0)	31.8	(5,0)	1.3
CF + OCS	√	0.5	(2,0)	39.5	(5,0)	1.3
CF + TCS	√	0.5	(2,0)	53.8	(5,0)	1.3
DC + DO	√	0.3	(2,0)	0.4	(2,0)	1.3
DC + DT		25.7	(3,3)	NA (35501.1)	(6,4)	0.9
DC + OCS		74.5	(3,4)	NA (3749.4)	(5,3)	1.2
DC + TCS		45.6	(3,3)	NA (3485.2)	(4,5)	1.2
DO + DT		4.8	(2,4)	NA (7057.7)	(3,8)	1.0
DO + OCS		16.3	(2,5)	NA (2922.1)	(3,7)	0.9
DO + TCS		19.6	(2,5)	NA (1700.3)	(3,6)	0.9
DT + OCS	√	0.3	(2,0)	0.7	(2,0)	0.9
DT + TCS	√	0.3	(2,0)	0.3	(2,0)	0.9
OCS + TCS	√	0.3	(2,0)	0.5	(2,0)	1.0

For all the three methods, the execution time needed for verification is presented. The execution time is the total time that elapsed between when a service specification was input and when the verification was completed. NA means that we could not complete verification since an error was caused by memory overflow when the program was generating an interpolant. (The number inside the parentheses shows the elapsed time till the error occurred.)

For the proposed method and McMillan's method, (k, r) shows the value of k of the finally executed instance of FINITERUN and the number of times of computing an interpolant in that execution of FINITERUN.

For all cases where the verification was completed, our proposed method outperformed the McMillan's ordinary unbounded model checking in execution time. In particular, when a violation required a relatively large number of transitions to occur, the proposed method could conclude the existence of the violation using a much smaller value k. Such cases include: CW + DT, CW + OCS, CW + TCS, CF + DT, CF + OCS, CF + TCS in Table 3.1 and CW + CF, CW + DT, CW + OCS, CW + TCS, CF + DT, CF + OCS, CF + TCS in Table 3.2. This clearly shows that when k is fixed, our encoding scheme allows a larger state space to be explored.

SPIN consistently exhibited good performance; but our proposed method outperformed SPIN for many cases, including: CW + DT, CW + OCS, CW + TCS, CF + DT, CF + OCS, CF + TCS, DC + DT, DC + OCS, DC + TCS, OCS + TCS in Table 3.1 and CF + DT, CF + OCS, CF + TCS, DC + DO, DT + OCS, DT + TCS, OCS + TCS in Table 3.2. When our method showed lower performance or even aborted, a large k was (or would be) needed for the algorithm to terminate. This fact is explained by the fact the time needed for generating an interpolant rapidly increases with the size of the input formula.

One might think that the result is somewhat discouraging; but we think there is still plenty of room for improving our method. We will discuss this point in the final chapter.

3.5 Summary

In this chapter, we proposed a verification method for checking whether or not feature interaction occurs in telecommunication services. We used a new encoding scheme that effectively represents the behaviors of asynchronous systems such as telecommunication systems. Based on this encoding, we developed an unbounded model checking method. To show the effectiveness of our method, we conducted experiments. To the best of our knowledge this was the first time to adapt unbounded model checking to the interleaving concurrency of asynchronous systems.

Chapter 4

Feature Interaction Verification in Home Network Systems

4.1 Introduction

In this chapter, we propose a method for detecting feature interactions by means of the SPIN model checker [32, 28]. The main contributions of this chapter are as follows: First, we propose a method for describing a home network system and its users in Promela [31, 33]. Our proposed approach focuses only on the high-level behavior of services, and thus can be used independently of underlying network protocols, such as [47, 19, 17]. Second, we classify feature interactions based on their causes. We also devise an LTL formula for each type of feature interaction.

This chapter is structured as follows. Section 4.2 introduces a running example of a home network system. Section 4.3 presents our proposed method for detecting feature interactions between services in home network systems. First, a method for describing home network systems and its users is presented. Then, the

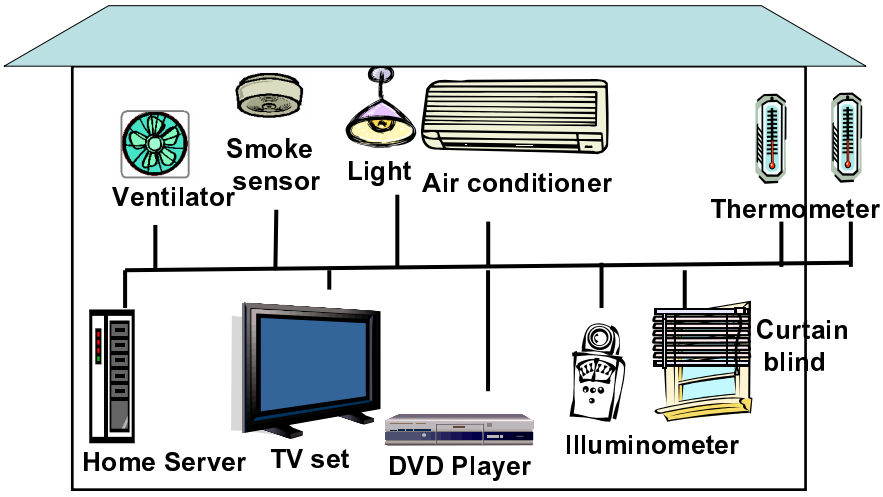

Figure 4.1: An example of home network system

classification of feature interactions and the LTL formula describing each classified type are presented. Section 4.4 shows the results of an experiment to demonstrate the usefulness of the proposed approach. Section 4.5 concludes this chapter.

4.2 Preliminaries

4.2.1 Home Network Systems

An example of a home network system is shown in Figure 4.1. The home network system consists of an air-conditioner, a ventilator, a smoke sensor, two thermometers (inside and outside a room), a DVD player, a TV set, lights, a curtain blind, an illuminometer and a home server.

4.2.2 Services in Home Network Systems

By integrating features of different appliances, convenient services can be implemented. These value-added services are one of the main advantages of home network systems. Below we present such services in our running example. Some of the services are taken from [27].

HVAC service (Heating): The HVAC service integrates the features of the air-conditioner, the two thermometers, and the ventilator. This service achieves energy-efficient air-conditioning of a room. If the room is cooler than the temperature set point, the HVAC service operates the air-conditioner in the heating mode. To efficiently warm the room up, the HVAC service turns the ventilator on to provide warmer outside air if the room temperature is cooler than the outside temperature. In this case the ventilator will be kept operating until the room temperature reaches the outside temperature. If the room temperature is warmer than the temperature set point, the HVAC service operates the air-conditioner in the fan mode.

Air-cleaning service: The air-cleaning service uses the smoke sensor and the ventilator to automatically clean the air in the room. When smoke is detected, this service automatically turns on the ventilator. The ventilator is kept operating until the sensor senses no smoke. When the air is cleaned, the ventilator will be turned off.

Home theater service: The home theater service uses the TV set, the DVD player, the illuminometer, the lights and the curtain blinds. When activated, this service turns on the TV set and the DVD player. At the same time, the curtain blinds in the room are drawn down, and the lights are adjusted to the optimal level based on the current brightness of the illuminometer.

Energy saving service: This service aims to conserve energy consumption by turning off unnecessary appliances. For example, when the power of the TV set is off, it is useless to keep the power of the DVD player turned on. This service will turn off the DVD player in such a situation.

4.2.3 Feature Interactions of Services

In this section, we show several examples of interactions between the services shown in Section 4.2.2.

Example 5 A feature interaction occurs between the HVAC service and the air-cleaning service. Consider the following situation: The room temperature is 15℃, the outside temperature is 8℃, and there is smoke in the room. The temperature set point of the HVAC service is 21℃. Now suppose that the HVAC service is operating the air-conditioner in the heating mode. In this case, the HVAC service tries to turn off the ventilator to prevent cool outside air from flowing into the room. On the other hand, the air-cleaning service tries to turn on the ventilator to clean the room air.

Example 6 This interaction occurs between the home theater service and the energy saving service. Consider the following scenario: The power of the TV set is OFF at the beginning. The energy saving service checks the power of the TV set. The service comes to know that the TV set is OFF and thus tries to turn off the DVD player. At the same time, the home theater service is activated and turns on the TV set. As a result, the DVD player is turned off, while the TV set is turned on.

Example 7 Suppose that the HVAC service is operating the air-conditioner in the heating mode to warm up the room temperature and that the air-cleaning service is using the ventilator to clean the room air. If the room temperature is higher than the outside temperature, cool outside air is taken by the ventilator. This lowers the efficiency of the HVAC service.

4.3 Detection of Feature Interactions with SPIN

To use SPIN, the system to be verified needs to be described in the Promela language and properties to be verified are represented as LTL formulas. In Section 4.3.1, we show a method for describing home network systems and its users. Next, in Section 4.3.2, we show a classification of feature interactions based on their causes. We also devise LTL formulas to detect these feature interactions.

4.3.1 Describing Home Network Systems and Users in Promela

This section shows a method for describing home network systems and its users in Promela. A home network system is modeled as three components: the environment, appliances and services. The environment consists of several elements, such as the room temperature and smoke. The state of these elements is changed by the effects from appliances. The appliances are operated by services. Users execute these services.

Figure 4.2 shows a model of home network systems and its users. In Figure 4.2, there are two users, two services (the HVAC service and the air-cleaning service), three appliances (an air-conditioner, a ventilator and a smoke sensor), and three elements of the environment (room temperature, smoke, and brightness).

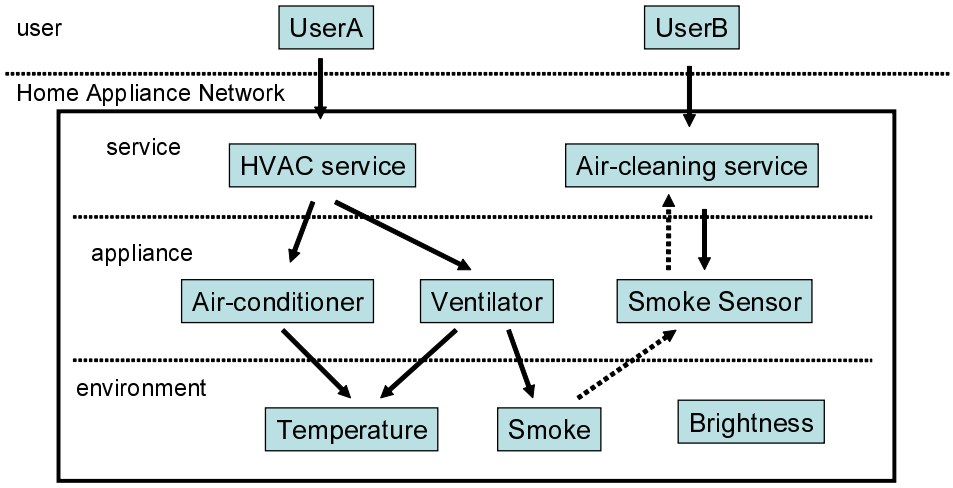

Figure 4.2: System model for home network systems

In this figure, the arrows represent relations between the four components. UserA executes the HVAC service. The HVAC service operates the air-conditioner and the ventilator. The air-conditioner has an effect on the room temperature, and the ventilator has effects on the room temperature and the smoke.

In this section, we present a method for describing these four components: the environment, appliances, services and users.

In Promela programs, the states of these components are represented by variables. The types of these variables are defined, for example, as follows:

```
#define tTemp int
#define tPower int /* ON or OFF */
#define OFF 0
#define ON 1
```

`tTemp` is the type of the variables that represent temperatures. `tPower` is the

type of the variables that represent the power of appliances. Variables of `tPower` type can take `OFF` or `ON`, which is internally represented as an integer 0 or 1.

The Environment

The elements of the environment are defined as global variables. A room temperature `temp_in` and an outside temperature `temp_out` can be defined as follows:

```
tTemp temp_in; tTemp temp_out;
```

To model an unpredictable environment we let these variables take arbitrary values when they are read by appliances. For example, the room temperature changes even when the air-conditioner is not working. This modeling technique can be found in, for example, [11].

We represent the effects of the appliances on the environment by Boolean-valued formulas over the variables that represent the state of appliances and the state of the environment. As shown later, these formulas allow us to detect conflicting effects. For example, the effect `temp_in_up`, which indicates the presence of some appliance that is warming the room up, is defined as follows:

```
#define temp_in_up
    ((AC_power == ON && AC_Mode == Heater)
     ||(ventilator_power == ON && temp_in < temp_out))
```

Here `AC_power`, `AC_mode` and `ventilator_power` respectively represent the power of the air-conditioner, the mode of the air-conditioner, and the power of the ventilator. This Boolean formula evaluates to true when the air-conditioner is working in the heating mode or when the outside temperature is warmer than the room and the ventilator is working.

Appliances

Appliances are described with variables and macros. The variables represent the state of the appliances. The macros represent the methods of the appliances, such as setting the power to ON and setting the mode to a particular mode.

The state of the appliance is defined as global variables as was done with the environment. For example, the variables `power`, `AC_temp` and `AC_Mode`, which represent the power, the temperature set point and the mode of the air conditioner, are defined as follows:

```
tPower AC_power=OFF; tTemp AC_Temp=25;
tMode AC_Mode=FAN;
```

The methods of the appliances are invoked by services. The methods can have arguments. The behavior of each method consists of reading the variables of the environment, writing/reading the variables of its own appliance and returning a value to the caller service.

Each method has a pre-condition and a post-condition. The method can be executed when the pre-condition is true. The post-condition need to become true immediately after the method is executed.

Each method is defined as a macro of Promela, as shown in Figure 4.3.

`pre_condition`, `post_condition` are Boolean expressions which represent a pre-condition and a post-condition respectively. In Promela, if a Boolean expression is used as a statement, it blocks system execution until the Boolean expression evaluates to true. Hence, when the pre-condition is true, the method executes the statements described at line 3. If the post-condition is also true, then `return_value` will be sent back to the caller service.

```
1 #define Appliance_method(argument){\
2  (pre_condition);\
3  ... /* The beavior of the method */
4  (post_condition);\
5  rvalue = return_value;}
```

Figure 4.3: Description of a method of an appliance

Services

Each service is modeled by two Promela processes, one of which represents the behavior of the service and the other of which represents the communication between the service and users.

The process representing the behavior controls appliances by executing their methods. A method invocation is performed by executing the macro corresponding to the method. Special local variable `r_value` is used to store the return value from a method.

Figure 4.4 shows the Promela code of the process that describes the behavior of the HVAC service. At lines 2, 3, local variables are defined. At lines 4 – 13, the behavior of the HVAC service is described. At line 4, this service waits until the variable `HVAC_State` is set to `START`. Line 5 is a do-statement which is an iteration statement of Promela. At lines 6 – 10, several macros are executed.

The state of a service is represented by a set of global variables. The process that deals with communication updates the state in response to the reception of a message from a user. Figure 4.5 shows the process for the HVAC service.

For each `Service`, global variable `Service_State` is declared to control the start and the stop of the service. This variable takes two values: `STOP`,

```
1  proctype HVAC(){
2   int rvalue;
3   tTemp Ti_temp,To_temp; /* Local variables*/
4  (HVAC_state == START);
5  do ::!(HVAC_state == STOP)->
6   Thermometer_in_SetPower(ON);
7   Thermometer_out_SetPower(ON);
8   AC_SetPower(ON);
9   Thermometer_in_Measure(); Ti_temp = r_value;
10  Thermometer_out_Measure(); To_temp = r_value;
11  do ::(Ti_temp < user_temp)->
12   AC_SetMode(HEATER);
13    :
14  }
```

Figure 4.4: The behavior part of the HVAC service

`START`. The service waits until a user sets this variable to `START`. When this variable is set to `START`, the service performs its execution. When the variable is set to `STOP` by a user, the service stops and waits until a user sets this variable to `START` again. In addition to `Service_State` other global variables may be used. For example, the HVAC service has variable `HVAC_SetTemp` to represent the temperature set point.

For each such global variable, a message channel is declared. `MC_HVAC_State` and `MC_HVAC_SetTemp` are the message channels for `HVAC_State` and `HVAC_SetTemp`. To set the variable to a particular value, users send a message in the message channel. For example, `MC_HVAC_state?HVAC_state`

```
1  proctype HVAC_cont(){
2   do
3   ::MC_HVAC_State?HVAC_State
4   ::MC_HVAC_SetTemp?HVAC_SetTemp
5   od;
6   }
```

Figure 4.5: The communication part of the HVAC Service

takes a message in the message channe `MC_HVAC_State` and stores it in variable `HVAC_State`.

Users

Users control services by sending messages to message channels. For example, in the case of the HVAC service, a user may send the value of the temperature set point, as well as a signal for start and stop, as shown in Figure 4.6. In this figure, `UserA` sets the variable `SetTemp`, a temperature set point, to 21℃ and sets the variable `HVAC_State` to `START`. After the HVAC service starts, the user sets `HVAC_State` to `STOP` to stop the HVAC service.

```
1  proctype UserA(){
2   MC_HVAC_SetTemp!21;
3   MC_HVAC_State!START;
4   MC_HVAC_State!STOP;
5   }
```

Figure 4.6: The execution of HVAC service by user A

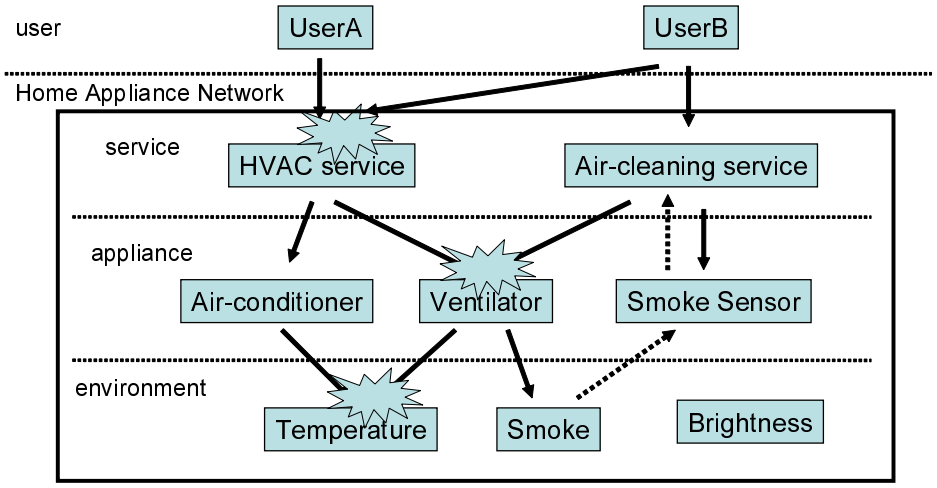

Figure 4.7: Feature interactions in home network systems

4.3.2 Representing Correctness Claims as LTL Formulas

In this section, we classify feature interactions based on their causes. To detect feature interactions using SPIN, we need to represent the absence of each type of feature interactions as an LTL formula. Two kinds of temporal operators are used in this chapter: always and eventually. The operator "always" is represented as `[]`. A formula `[] P` evaluates to true if `P` is always true in all system executions. The operator "eventually" is represented as `<>`. A formula `<> P` evaluates to true if `P` eventually becomes true in all system executions.

In general, feature interaction occurs when conflicting accesses are attempted to the same resource. Since there are four types of components (i.e., users, services, appliances and the environment), we have a simple classification as follows:

Interaction with services: Two users send conflicting commands to the same service.

Interaction with appliances: Two services are attempting conflicting operations on the same appliance.

Interaction with the environment: Two appliances have conflicting effects on the same element of the environment.

Figure 4.7 shows the examples of these interactions. In this figure, UserA and UserB can send conflicting commands to the HVAC service. The HVAC service and the air-cleaning service can operate the ventilator with conflicting purposes. The air-conditioner and the ventilator have conflicting effects on the room temperature.

Interactions with Services

Interactions occur with services when two users send conflicting commands to the same service. This type of interaction can be detected by checking if an incoming message conflicts with the previous message.

We consider a message m from a user to a service to be conflicting if the following three conditions are met: the service has already received another message m'; m' was issued by a different user; and the command of m is different from that of m'.

To detect conflicting messages, we modify send/receive statements of user and service processes in two respects. First, the identity of sender users is attached to every message. Second, additional Promela code is inserted immediately after each receive statement of a service. For example, the receive statement of line 3 in Figure 4.5 will be modified as follows:

```
MC_HVAC_State?user,HVAC_State;
```

```
if
::(MC_HVAC_State?[_,_] &&
  !MC_HVAC_State?[eval(user),_] &&
  !MC_HVAC_State?[_,eval(HVAC_State)])
   -> HVAC_error = 1;
::else -> skip;
fi;
```

`MC?[a1,a2]` evaluates to true iff in channel `MC` a message (`a1,a2`) exists. Underscore (_) is used as a wildcard. `eval()` is a function that returns the current value of the variable given as an argument.

The variable `HVAC_error` is used to record the occurrence of interactions. We let each `Service` have variable `Service_error`. If there is a conflicting message in the message channel, `Service_error` is set to 1. As a result interactions with each `Service` can be detected by checking the following LTL formula:

```
!<> (Service_error == 1)
```

Interactions with Appliances

Application-level interactions occur when several services try to execute conflicting operations on the same appliance. This type of interaction occurs in two situations: A1) two services try to change the state of the appliance to different states and A2) one service reads the state of an appliance when another service is changing the state.

We define variable `appliance_error` for each appliance to represent whether or not a feature interaction has occurred (0: not occurred, 1: A1, 2: A2). `appliance_error` is updated when a feature interaction occurred.

Detection of A1: Interaction A1 occurs when two services try to set the same variable of an appliance to different values. To detect this interaction, we can use a similar way as interaction S1. We translate a variable assignment statement into a pair of a send statement and a receive statement, mimicking value assignment as message passing. When the send statement is executed, interaction A1 can be detected by checking whether or not a conflicting message has already been sent. For example, Figure 4.8 shows method `AC_SetMode(mode)` of the air-conditioner.

```
1 #define AC_SetMode(mode){\
2  AC_Power == ON;\
3  if\
4  ::(MC_AC_Mode?[_] && !(MC_AC_Mode?[mode]))\
5     -> AC_error = 1;\
6  ::else -> skip;\
7  fi;\
8  MC_AC_Mode!mode;\
9  MC_AC_Mode?AC_Mode;}
```

Figure 4.8: Method SetMode of the air-conditioner

In Promela, if a message channel is full, the send statement waits until the message channel becomes non-full. For a method to receive the message sent by itself, the buffer size of message channel must be 1. This guarantees that, for

example, when the receive statement is executed at line 9 in Figure 4.8, the only message in the message channel is the message sent at line 8. Hence, the message that can be received at line 9 is only the message sent at line 8.

Detection of A2: Interaction A2 occurs in the situation where one service tries to read a variable when another service is changing the value of the variable. As a result of this interaction, the value obtained by the former service becomes different from the actual value of the variable. This interaction can be detected by checking, whenever a method reads a variable, if the channel associated with that variable contains a new value different from the current one. For example, the macro `TV_CheckPower`, which is used to check the power of the TV set by services, is described as follows:

```
#define TV_CheckPower(){\
true;\
if\
::(MC_TV_Power?[_] && \
  !(MC_TV_Power?[eval(TV_Power)]))\
   -> TV_error = 2;\
::else -> skip;\
fi;\
r_value = TV_Power;}
```

For each appliance, interactions on it can be detected by checking if the value of `Appliance_error` is 1 or 2. If checking interaction A1, for example, one can use the following LTL formula:

```
!<> AC_error == 1
```

Interactions with the Environment

Interactions with the environment occur when two appliances have conflicting effects on the same element of the environment. Hence this interaction occurs in the following two situations: E1) there are two different kinds of effects occurring on the same element of the environment simultaneously, and E2) some appliance reads the state of an environment element while some effect on that element is existing.

Detection of E1: This interaction can be detected by checking that continuous conflicting effects on the same environment element never occurs. This property is represented by an LTL formula as follows:

```
LTL:!<> [](e_eff1 && e_eff2)
```

where `e_eff`i is the formula that represents that some appliance has an effect `eff`i on the element `e`. This LTL formula asserts that the two different effects on the same element never occurs simultaneously. When the number of the types of effects is more than 2, by checking all pairs of effects, one can detect this type of interactions. For example, when the number of types is 3, the LTL formula can be described as follows:

```
LTL:!<>( [](e_eff1 && e_eff2) || [](e_eff1 && e_eff3)
       || [](e_eff2 && e_eff3) )
```

Detection of E2: This interaction can be detected by checking if some effect is existing on an element of the environment when a service reads the state of the same environment element. For each environment element `e`, variable `e_read` is used to detect such a situation. We add the following statements to all methods

that read the value of the element e at the point just before a statement that reads the value of the element.

```
e_read = 1; e_read = 0;
```

The value of e_read becomes 1 only if some method has just read the state of e. As a result this type of interaction can be detected by using the following LTL formula:

```
LTL:!<> (e_read && (e_eff1 || .. || e_effn))
```

(e_eff1 || .. || e_effn) represents that no effect is occurring on e. Hence, this LTL formula asserts that effects on the environment element e never exist while the state of e is being read.

4.4 Experiment

We conducted an experiment, in which we attempted to detect interactions caused by any pair of the four services of our running example (see Section 4.2.2). In this experiment, we assumed that there are two users and each user executes a single service. The experiment was conducted on a WindowsXP PC with a 900MHz PentiumIII and 512MB memory. SPIN was used with partial order reduction enabled.

In our running example, there are ten appliances as shown in Figure 4.1. Each appliance has one or two variables and two to six methods. The lines of code of the HVAC service, of the air-cleaning service and of the home theater service are all approximately 50 lines. The energy saving service is described in around ten lines.

Detection of service-level interactions was conducted by enforcing the two users to run the same service. Unlike the other three services, the HVAC service requires the user to specify the temperature set point. In this experiment, this value is set to 21 ℃ or 25 ℃.

For each of the two interaction types, A1 and A2, we check whether or not the interaction occurs for the ten appliances. Hence, we run a verification 20 times for each pair of services.

To detect interactions with the environment, we run a verification eight times, because there are two types, E1 and E2, of interactions and four environment elements.

4.4.1 Verification Results

The verification results are summarized in Table 4.1: This table shows the interactions detected between two services. S1, A1, A2, E1 and E2 indicate the type of interactions (services (S1), appliances (A1, A2), the environment (E1, E2)). Each symbol is followed by the component with which the interaction occurs. For example, when one user executes the HVAC service and the other user executes the air-cleaning service, a type A1 interaction with ventilator can occur. In the same case, type E1 and type E2 interactions with the room temperature and a type E2 interaction with smoke can occur.

Some performance figures are shown in Table 4.2 and Table 4.3 for the pair of the HVAV service and the air-cleaning service. Table 4.2 shows the results for interactions with appliances, while Table 4.3 shows those for interactions with the environment. These tables show the verification results (`true` indicates that an interaction was found), the execution time, and the number of states explored for

Table 4.1: Interactions detected between service examples in home network system

HVAC service	Air-cleaning service	Home theater service	Energy saving service	
S1(HVAC service) E2(room temperature)	A1(ventilator) E1(room temperature) E2(room temperature) E2(smoke)	E2(room temperature) E2(brightness)	E2(room temperature)	HVAC service
	S1(Air-cleaning service) E2(smoke)	E2(smoke) E2(brightness)	E2(smoke)	Air-cleaning service
		S1(Home theater service) E2(brightness)	A1(DVD player) A2(TV set) E2(brightness)	Home theater service
			S1(Energy saving service)	Energy saving service

Table 4.2: Interaction with appliances between the HVAC service and the air-cleaning service

appliance	type	result	time(s)	state
Air-conditioner	A1	true	1.57×10^1	9.9×10^5
	A2	true	1.45×10^1	9.9×10^5
Thermometer	A1	true	1.46×10^1	9.9×10^5
(inside)	A2	true	1.45×10^1	9.9×10^5
Thermometer	A1	true	1.55×10^1	9.9×10^5
(outside)	A2	true	1.48×10^1	9.9×10^5
smoke sensor	A1	true	1.52×10^1	9.9×10^5
	A2	true	1.45×10^1	9.9×10^5
ventilator	A1	false	2.65×10^{-1}	1.1×10^2
	A2	true	1.49×10^1	9.9×10^5

each appliance or environment element. The results for the appliances that are not used by the two services are omitted. As shown in these tables, the time required for verification is fairly reasonable.

Using counterexamples provided by SPIN, we detected scenarios leading to interactions. Below we show several examples of such scenarios.

Type A1 interaction with the ventilator between the HVAC service and the air-cleaning service: The room temperature is warmer than the outside temperature, and there is smoke in the room. The HVAC service and the air-cleaning service are both running. The HVAC service calls method `SetPower(OFF)` of the ventilator to turn it off, to prevent cool outside air from flowing into the room. On the other hand, the air-cleaning service executes method `SetPower(ON)`

Table 4.3: Interaction with the environment between the HVAC service and the air-cleaning service

element	type	result	time(s)	state
room temperature	E1	false	2.50×10^{-1}	8.7×10^{2}
	E2	false	2.34×10^{-1}	2.0×10^{1}
outside temperature	E1	true	1.47×10^{1}	9.9×10^{5}
	E2	true	1.51×10^{1}	9.9×10^{5}
smoke	E1	true	1.45×10^{1}	9.9×10^{5}
	E2	false	2.50×10^{-1}	2.6×10^{1}
brightness	E1	true	1.48×10^{1}	9.9×10^{5}
	E2	true	1.44×10^{1}	9.9×10^{5}

of the ventilator to turn it on to clean the room air. As a result, the conflicting operations of the ventilator are executed at the same time.

Type E1 interaction with the room temperature between the HVAC service and the air-cleaning service: This scenario is the same as the third example in Section 4.2.3.

Type E2 interaction with the room temperature between the HVAC service and the air-cleaning service: Suppose that the room temperature is cooler than the outside temperature and that the air-cleaning service is operating the ventilator to clean the room air. In this situation, the ventilator warms the room up. Now the HVAC service executes the method that measures the current room temperature. Since the ventilator is having an effect on the room temperature, the HVAC service can erroneously recognize the room temperature as if it were lower than the actual value, resulting in execution of unnecessary heating.

Type A1 interaction with the DVD player between the home theater service and the energy saving service: The power of the TV set is OFF at the beginning. The energy saving service checks the power of the TV set. The service comes to know that the TV set is OFF and thus tries to execute `SetPower(OFF)` to turn off the DVD player. At this time, the home theater service is activated and tries to execute `SetPower(ON)` to turn the DVD player on. As a result, the conflicting operations to DVD player are executed.

Interaction A2 with the TV set between the home theater service and the energy saving service: This scenario is the same as the second example in Section 4.2.3.

4.4.2 Discussion

Unlike telecommunication systems, modeling home network systems requires us to consider the "physical" environment which might involve, for example, temperature or brightness. A method for detecting feature interactions in a telecommunication system by using the SPIN model checker is proposed in [9], but it can not be applied to home network systems. In the context of interaction detection for intelligent building control systems, Metzger and Webel proposed an approach to deal with such physical elements of the environment [37]. The idea of their approach is to detect different services that access the same environment element. Unlike ours, their approach does not consider how the services affect the environment. As a result it easily yields false negatives.

In our proposed framework, we define the types of effects on the environment to overcome this problem. For example, consider the situation where both the HVAC service and the air-cleaning service operate a ventilator. Because the

ventilator has effect on the “Smoke” element of the environment these two services access the same element of the environment. Existing methods such as [37] consider these situations as feature interactions. However, this situation is not undesirable because the purpose of each service is not interfered. In our proposed framework, these situations were correctly considered desirable. This can be seen in the experiment in Section 4.4. The verification result shows that type E1 interaction with smoke between the HVAC service and the air-cleaning service does not occur.

Research that addressed the feature interaction problem of home network systems includes [27, 39]. In [27], a runtime detection method and a priority-based resolution are proposed. Our approach works at a higher abstract level than [27] in the sense that interactions detected by our approach might be resolved by prioritizing services. Thus, even when such a runtime resolution exists, the results obtained through our approach can be used to identify the situations where the mechanism comes into play, resulting in a better understanding of the system behavior.

In [39], a static method for detecting feature interactions is proposed. However, this method is a conservative approximation method and thus can detect false feature interactions which will never occur in actual runs.

The authors of [29] propose a method for verifying the behavior of services with the SMV model checker [35]. This method can also be used for feature interaction detection, it was not the main objective of [29], though. The work presented in this chapter improves [29] in several ways: First, we classified interactions and devised the LTL correctness claim for each category. These LTL formulas allow systematic interaction detection, while in [29] correctness claims were constructed

in an ad hoc manner. Another improvement came from our adopting the Promela language. Unlike the SMV input language, Promela is similar to conventional procedural programming languages. This makes it much easier to describe the specification of appliances and services.

Our proposed framework focuses only on the high-level behavior of services, and thus can be used independently of underlying network protocols. For example, we show a way of applying our framework to ECHONET specifications.

Appliances are represented as objects which consist of state variables in ECHONET. Appliances are operated by writing/reading the variables. The variables in ECHONET specifications can be directly represented as those in our model. Writing/reading operations on the variables in ECHONET specifications can be translated into methods in our model. Because the effects on the environment are not defined in ECHONET specifications, we must define the effects on the environment for each state of appliances. As a result, applying our proposed framework to ECHONET specifications proceeds as follows: First, state variables of appliances of ECHONET specifications are translated into variables and methods of appliances in our model. Second the effects on the environment are defined for appliances. Finally, the services are described in the format of our model.

As a case study, we describe a ventilator and a service which achieves low power consumption of houses in the ECHONET standard available from [17] using our proposed model. This is shown in the appendix of this chapter.

4.5 Summary

In this chapter, we proposed a method for detecting feature interactions in home network systems. The proposed method uses SPIN, an LTL model checker. We classified interactions into several types and devised LTL formulas that represent the absence of these interactions. To demonstrate the usefulness of the proposed approach, we conducted an experiment. We checked whether or not feature interactions occur in our running example and successfully detected several interactions. By using counterexamples produced by SPIN, we also succeeded in obtaining scenarios leading to the interactions.

4.6 Appendix

Description of Ventilator

We show that the specification of a ventilator described in the ECHONET standard can be represented using our proposed model.

The ventilator has eight variables: `OperationStatus`, `RoomRelativeHumid`, `VentilatingStatus`, `MeasureHumid`, `VentilatingWindLevel`, `HeatExchangerStatus`, `CO2Concentration`, and `SmokeDetectionStatus`.

Six variables, `OperationStatus`, `RoomRelativeHumid`, `VentilatingStatus`, `VentilatingWindLevel`, `HeatExchangerStatus`, and `CO2Concentration`, and two operations, Get, by which the variable is read, and Set, by which the variable is written, are defined. Two variables, `MeasureHumid` and `SmokeDetectionStatus`: have only one operation Get. To rep-

resent these operations as methods, we define the pre-condition and the post-condition of each method as follows: The pre-conditions for all methods are true. The post-condition of Set is defined such that it becomes true if the value of the variable has been correctly updated. The post-condition of Get is defined as true. Here we only show the pre-conditions and the post-conditions for Set and Get of `OperationStatus`, because these for the other variables are identical except the name of the variable.

Since the effects on the environment are not defined in ECHONET, we define three types of effects on the environment. When the state of the ventilator is ON, the smoke is removed. When the state of the ventilator is ON and the room temperature is warmer than the outside, the effect "down" on the element "temp_in" of the environment occurs. On the other hand, when the state of the ventilator is ON and the room temperature is cooler than the outside, the effect "up" on the element "temp_in" of the environment occurs.

- Ventilation = (F_V, M_V, EW_V). F_V is a set of variables, M_V is a set of methods, and EW_V(eff) is a formula which represents the state of the ventilator and the state of the environment when the ventilator has an effect on the element eff of the environment.

- F_V = (OperationStatus, RoomRelativeHumid, VentilatingStatus, MeasuredHumid, VentilatingWindLevel, HeatExchangerStatus, CO2Concentration, SmokeDetectionStatus)

- M_V = (Get_OperationStatus(), Set_OperationStatus(status), Get_RoomRelativeHumid(), Set_RoomRelativeHumid(humid), Get_VentilatingStatus(), Set_VentilatingStatus(status), Get_MeasuredHumid(), Get_VentilatingWind-

Level(), Set_VentilatingWindLevel(level), Get_HeatExchangerStatus(), Set_HeatExchangerStatus(status), Get_CO2Concentration(), Set_CO2Concentration(concentration), Get_SmokeDetectionStatus(status))

- Pre(Get_OperationStatus()) = [true]
- Post(Get_OperationStatus()) = [true]
- Pre(Set_OperationStatus(status)) = [true]
- Post(Set_OperationStatus(status)) = [OperationStatus =status]

- EW_V(eff):

  eff=Smoke_removal: [OperationStatus = ON]

  eff=Temp_in_up: [OperationStatus = ON $\wedge$ Temp_in$<$Temp_out]

  eff=Temp_in_down: [OperationStatus = ON $\wedge$ Temp_in $>$ Temp_ out]

  otherwise: [false]

The Promela description can be obtained by translating the above description. We show the Get and Set methods for `OperationStatus`.

```
/* Definition of types */
#define tStatus int
#define OFF 0
#define ON 1
#define tAuto
#define Auto 2
#define NonAuto 3
#define tSmoke int
```

```
#define Found 4
#Define NotFound 5

/* Definition of Variable */
tStatus Ventilation_OperationStatus;
int Ventilation_RoomRelativeHumid;
tAuto Ventilation_VentilatingStatus;
int Ventilation_MeasuredHumid;
int Ventilation_VentilatingWindLevel;
tStatus Ventilation_HeatExchangerStatus;
int Ventilation_CO2Concentration;
tSmoke Ventilation_SmokeDetectionStatus;
/* variable for detecting interaction */
int Ventilation_error=0;

/* Operations for OperationStatus */
#define Ventilation_Get_OperationStatus(){\
 true;\
 if::(MC_Ventilation_OperationStatus?[_] &&
  !(MC_Ventilation_OperationStatus?
     [eval(Ventilation_operationStatus)]))\
   -> Ventilation_error = 2;\
 ::else-> skip;fi;\
 r_value = Ventilation_OperationStatus;
}
#define Ventilation_Set_OperationStatus(status){\
 true\
```

```
 if::(MC_Ventilation_OperationStatus?[_] &&
     !(MC_Ventilation_OperationStatus?[mode]))\
   ->Ventilation_error = 1;\
   ::else -> skip; fi;
 MC_Ventilation_OperationStatus!status;\
 MC_Ventilation_OperationStatus?Ventilation_OperationStatus;\
}
```

Description of the Energy Management Service

Here, EMS (Energy Management Service) in ECHONET is described in Promela. This service checks the total power consumption of all appliances, and if the power consumption exceeds a certain value (Is) set by users, the service suspends the appliances based on the priority until the power consumption becomes lower than the value of Is. If the power consumption is lower than the value Ie and there are appliances suspended by the service, the service restarts the appliances. To use this service, users need to set the values of Is and Ie. The behavior of this service is as follows:

1. This service checks the power consumption of all appliances. If the total power consumption exceeds Is, step 2 is executed. If the power consumption is lower than Ie, the next step is step 3. Otherwise, it repeats step 1.

2. This service suspends the appliance that has the lowest priority, and returns to step 1. If there are no appliances that can be suspended, this service gives the alarm and terminates.

3. If there are suspended appliances, this service restarts one of the appliances.

After that, this service returns to step 1.

As in the document of the ECHONET standard, we consider a home network system which consists of an air-conditioner, a ventilator, a freezer, a microwave, a heater, and a washer. In this case, EMS can be described as follows. We assume that the priorities and the power consumption of all appliances are given. About one hundred lines of Promela code for the EMS service is obtained.

```
int EMS_State;
int priority[6]; /* priority for each appliance*/
int consumption[6];/* power consumption*/

SERVICE EMS(int Is, int Ie){
 VAR
  int It;
  bool cut[6]; # suspended appliance
               # suspended: 1
  int maxpriority, minpriority;
    #max: appliance: not suspended and max priority
    #min: appliance: suspended and min priority
  tStatus Status_tmp;

CONTENT
 # initialize for cut
 cut[0]=0;cut[1]=0;cut[2]=0;
 cut[3]=0;cut[4]=0;cut[5]=0;

 while(EMS_STATE=START){
```

```
  #maxpriority, minpriority
  It = 0; maxpriority = 0; minpriority = 255;
#power consumption of air-conditioner
  Status_tmp :=
    Air_conditioner.Get_OperationStatus();
  if(Status_tmp == ON){
   It := It + consumption[0];
   if(cut[0] = 0 && priority[0] < minpriority){
     minpriority := priority[0];}}
  if(cut[0] = 1 && priority[0] > maxpriority){
   maxpriority := priority[0];}
#power consumption of ventilator
 Status_tmp := Ventilater.Get_OperationStatus();
 if(Status_tmp = ON){
   It := It + consumption[1];
   if(cut[1] = 0 && priority[1] < minpriority){
     minpriority := priority[1];}}
 if(cut[1] = 1 && priority[1] > maxpriority){
  maxpriority := priority[1];}
#power consumption of freezer
 Status_tmp := Freezer.Get_OperationStatus();
 if(Status_tmp = ON){
  It := It + consumption[2];
  if(cut[2] = 0 && priority[2] < minpriority){
    minpriority := priority[2];}}
 if(cut[2] = 1 && priority[2] > maxpriority){
  maxpriority = priority[2];}
```

```
#power consumption of microwave
 Status_tmp := Microwave.Get_OperationStatus();
 if(Status_tmp = ON){
  It := It + consumption[3];
  if(cut[3] = 0 && priority[3] < minpriority){
    minpriority := priority[3]; }}
 if(cut[3] = 1 && priority[3] > maxpriority){
  maxpriority = priority[3];}
#power consumption of heater
 Status_tmp := Heater.Get_OperationStatus();
 if(Status_tmp = ON){
  It := It + consumption[4];
  if(cut[4] = 0 && priority[4] < minpriority){
    minpriority := priority[4];}}
 if(cut[4] = 1 && priority[4] > maxpriority){
  maxpriority := priority[4];}
#power consumption of washer
 Status_tmp := Washer.Get_OperationStatus();
 if(Status_tmp = ON){
  It := It + consumption[5];
  if(cut[5] = 0 && priority[5] < minpriority){
     minpriority := priority[5];}}
 if(cut[5] = 1 && priority[5] > maxpriority){
  maxpriority := priority[5];}

 if(It>Is){/* suspend appliances */
  if(minpriority = priority[0]){
```

```
   Air_conditioner.Set_OperationStatus(OFF);
   cut[0] := 1;}
  else if(minpriority = priority[1]){
   Ventilater.Set_OperationStatus(OFF);
   cut[1] := 1;}
  else if(minpriority = priority[2]){
   Freezer.Set_OperationStatus(OFF);
   cut[2] := 1;}
  else if(minpriority = priority[3]){
   Microwave.Set_OperationStatus(OFF);
   cut[3] := 1;}
  else if(minpriority = priority[4]){
   Heater.Set_OperationStatus(OFF);
   cut[4] := 1;}
  else if(minpriority = priority[5]){
   Washer.Set_OperationStatus(OFF);
   cut[5] := 1;}
  else {Alarm.Set_OperationStatus(ON);}}
 ::(It<Ie) -> /* restart appliances*/
  if(maxpriority = priority[0]){
   Air_conditioner.Set_OperationStatus(ON);
   cut[0] := 0;}
  else if(maxpriority = priority[1]){
   Ventilater.Set_OperationStatus(ON);
   cut[1] := 0;}
  else if(maxpriority = priority[2]){
   Freezer.Set_OperationStatus(ON);
```

```
   cut[2] := 0;}
  else if(maxpriority = priority[3]){
   Microwave.Set_OperationStatus(ON);
   cut[3] := 0;}
  else if(maxpriority = priority[4]){
   Heater.Set_OperationStatus(ON);
   cut[4] := 0;}
  else if(maxpriority = priority[5]){
   Washer.Set_OperationStatus(ON);
   cut[5] := 0;}
 }
}
```

Chapter 5

Conclusion

5.1 Achievements

In this dissertation, methods for verifying feature interactions in telecommunication systems and home network systems were described.

The first achievement is the development of a verification method for verifying feature interactions in telecommunication services with unbounded model checking. First, a new encoding scheme that effectively represents the behaviors of telecommunication systems was proposed. Next, a method for adapting unbounded model checking to this encoding was described.

To show the effectiveness of our method, we conducted experiments where 21 pairs of telecommunication services were verified using several methods including ours. The results showed that our approach exhibited significant speed-up over unbounded model checking using the traditional encoding. Our method exhibited better or comparable performance comparing to the SPIN model checker in many cases. In some cases, however, our method could not complete verification, while

SPIN solved the verification problem in a few seconds.

The second achievement is the development of a framework for detecting feature interactions in home network systems. Our proposed approach for verification of feature interactions consists of two part. First, a model for home network system is proposed, and the feature interactions in home network systems is classified based on their causes. Next, an automatic translation method from proposed model into Promela, which is the input language of the SPIN model checker is proposed. we also proposed LTL formulas which represent the absence of each type of feature interactions. By using our proposed framework, one can automatically detect feature interactions in home network systems.

To demonstrate the usefulness of the proposed approach, we conducted an experiment. We checked whether or not feature interactions occur in our running example and successfully detected several interactions. By using counterexamples produced by SPIN, we also succeeded in obtaining scenarios leading to the interactions.

5.2 Future Research

Some issues are left for future research. In the research on telecommunication systems, our implementation is still in its prototype stage. Our current implementation uses FOCI for interpolant generation. FOCI supports not only pure propositional logic but also uninterpreted functions or linear arithmetic. By developing a new, faster interpolation procedure tailored to propositional logic, we may be able to enhance the performance of our method. This expectation can also be justified by the facts that the research on interpolation is still in its early stage,

and that the performance of SAT solving has been improved by several orders of magnitude in this decade.

In the research on home network systems, a tool should be developed in order to support users to describe and validate home network services based on the proposed framework. In this dissertation we show how users can verify home network services; but doing this requires them to have some knowledge about the framework. Such a tool would greatly facilitate ordinary users to use the automatic verification method.

The issue of resolving the detected feature interactions in home network systems still remains. Many approaches for resolving feature interactions exist. For example, when feature interaction occurs, the system may ask the users to determine which service execution is suspended, or the users may assign priorities to services to automatically abort a service with a lower priority. Finding the best resolution method for each type of feature interactions is needed.

Bibliography

[1] N. Amla and K. L. McMillan. Combining abstraction refinement and sat-based model checking. In *Proc. the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2007)*, Vol. 4424, pages 405–419, Braga, Portugal, March 2007.

[2] D. Amyot and L. Logrippo, editors. *Feature Interaction in Telecommunications and Software Systems VII*. 2003.

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

[4] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *Proc. the 5th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS 1999)*, pages 193–207, London, UK, 1999.

[5] L. G. Bouma and H. Velthuijsen, editors. *Feature Interaction in Telecommunications Systems*. 1994.

[6] T. F. Bowen, F. S. Dworack, C. H. Chow, N. Griffeth, G. Herman, and Y. J. Lin. The feature interaction problem in telecommunications systems. In *Proc. the 7th International Conference on Software Engineering for Telecommunication Switching Systems*, pages 59–62, London, July 1989.

[7] J. R. Büchi. On a decision method in restricted second order aritmetic. In *In Proceedings of the International Congress on Logic, Methodology and Philosophy of Science*, pages 1–11. Stanford University Press, 1960.

[8] M. Calder and E. Magill, editors. *Feature Interaction in Telecommunications and Software Systems VI*. 2000.

[9] M. Calder and A. Miller. Feature interaction detection by pairwise analysis of ltl properties - a case study. *Formal Methods in System Design*, Vol. 28, No. 3, pages 213–261, May 2006.

[10] E. J. Cameron and H. Velthuijsen. Feature interactions in telecommunications systems. *IEEE Communication Magazine*, Vol. 31, No. 8, pages 18–23, 1993.

[11] W. Chan, R. J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J. D. Reese. Model checking large software specifications. *IEEE Trans. Softw. Eng.*, Vol. 24, No. 7, 1998.

[12] K. E. Cheng and T. Ohta, editors. *Feature Interaction in Telecommunications III*. 1995.

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

[14] F. Copty, L. Fix, R. Fraer, E. Giunchiglia, G. Kamhi, A. Tacchella, and M. Y. Vardi. Benefits of bounded model checking at an industrial setting. In *Proc. the 13th International Conference on Computer Aided Verification (CAV 2001)*, pages 436–453, London, UK, 2001.

[15] P. Dini, R. Boutaba, and L. Logrippo, editors. *Feature Interaction in Telecommunications Networks IV*. 1997.

[16] L. du Bousquet and J.-L. Richer, editors. *Feature Interaction in Telecommunications and Software Systems IX*. 2007.

[17] ECHONET Consortium. `http://www.echonet.gr.jp/`.

[18] A. Gammelgaard and J. E. Kristensen. Interaction detection, a logical approach. In *Proc. the 2nd Workshop on Feature Interaction in Telecommunication Systems*, pages 178–196, Amsterdam, The Netherlands, May 1994.

[19] Havi. `http://www.havi.org/`.

[20] Y. Hirakawa and T. Takenaka. Telecommunication service description using state transition rules. In *Proc. the 6th international workshop on Software specification and design (IWSSD 1991)*, pages 140–147, Como, Italy, October 1991.

[21] G. J. Holzmann. The model checker SPIN. *IEEE Trans. Software Eng.*, Vol. 23, No. 5, pages 279–295, 1997.

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

[23] H.-J. Kang and I.-C. Park. SAT-based unbounded symbolic model checking. *IEEE Transaction on Computer-aided Design of Intergrated Circuit and Systems*, Vol. 24, No. 2, pages 129–140, 2005.

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

[25] A. Khoumsi. Detection and resolution of interactions between services of telephone networks. In *Proc. the 4th Workshop on Feature Interaction in Telecommunication Systems*, pages 78–92, Montréal, Canada, June 1997.

[26] K. Kimber and L. G. Bouma, editors. *Feature Interaction in Telecommunications and Software Systems V*. 1998.

[27] M. Kolberg, E. H. Magill, and M. Wilson. Compatibility issues between services supporting networked appliances. *IEEE Communications Magazine*, Vol. 41, No. 11, pages 136–147, 2003.

[28] P. Leelaprute, T. Matsuo, T. Tsuchiya, and T. Kikuno. Detecting feature interactions in home appliance networks. In *Proc. the 9th Int'l Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2008)*, pages 895–903, August 2008.

[29] P. Leelaprute, M. Nakamura, T. Tsuchiya, K. Matsumoto, and T. Kikuno. Describing and verifying integrated services of home network systems. In *The 10th Asia-Pacific Software Engineering Conference (APSEC2005)*, pages 549–558, December 2005.

[30] J. Marques-Silva. Interpolant learning and reuse in sat-based model checking. *Electronic Notes in Theoretical Computer Science*, Vol. 174, No. 3, pages 31–43, 2007.

[31] T. Matsuo. A model of home network system for detecting feature interactions by applying model checking. In *Supplemental Volume of the 2007 International Conference on Dependable Systems and Networks (DSN-2007)*, pages 300–302, June 2007.

[32] T. Matsuo, P. Leelaprute, T. Tsuchiya, and T. Kikuno. Verifying feature interactions in home network systems. *IPSJ Journal (In Japanese)*, Vol. 49, No. 6, pages 2129–2143, June 2008.

[33] T. Matsuo, P. Leelaprute, T. Tsuchiya, T. Kikuno, M. Nakamura, H. Igaki, and K. Matsumoto. Automatically verifying integrated services in home network systems. In *Proc. International Technical Conference on Circuits/Systems, Computers and Communications (ITC-CSCC2006)*, Vol. 2, pages 173–176, July 2006.

[34] T. Matsuo, T. Tsuchiya, and T. Kikuno. Feature interaction verification using unbounded model checking with interpolation. *IEICE Transaction on Information and systems (conditional acceptance)*.

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

[36] K. L. McMillan. Interpolation and sat-based model checking. In *Proc. the 15th International Conference on Computer Aided Verification (CAV2003)*, Vol. 2725, pages 1 – 13, Boulder, CO, USA, July 2003.

[37] A. Metzger and C. Webel. Feature interaction detection in building control systems by means of a formal product model. *Feature Interactions in Telecommunications and Software Systems VII*, pages 105–122, June 2003.

[38] S. Moyer, D. Marples, and S. Tsang. A protocol for wide area, secure networked appliances communication. *IEEE Communications Magazine*, Vol. 38, No. 10, pages 52–59, October 2001.

[39] M. Nakamura, H. Igaki, and K. Matsumoto. Feature interactions in integrated services of networked home appliance. In *Proc. of Int'l. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05)*, pages 236–251, June 2005.

[40] S. Ogata, T. Tsuchiya, and T. Kikuno. Sat-based verification of safe petri nets. In *Proc. the 2nd International Symposium on Automated Technology for Verification and Analysis (ATVA 2004)*, pages 72–92, November 2004.

[41] OSGi Appliance. The OSGi service platform. `http://osgi.org`.

[42] D. Peled. Combining partial order reductions with on-the-fly model-checking. In *Proc. the 6th International Conference on Computer Aided Verification (CAV94)*, pages 377–390, London, UK, 1994. Springer-Verlag.

[43] A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th IEEE symposium on foundation of computer science*, pages 46–57. IEEE Computer Society Press, 1977.

[44] P. Pudlák. Lower bounds for resolution and cutting plane proofs and monotone computations. *Symbolic Logic*, Vol. 62, No. 2, pages 981–998, September 1997.

[45] S. Reiff-marganiec and M. D. Ryan, editors. *Feature Interaction in Telecommunications and Software Systems VIII*. 2005.

[46] A. Rybalchenko and V. Sofronie-Stokkermans. Constraint solving for interpolation. In *Proc. the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2007)*, pages 346–362, Nice, France, January 2007.

[47] UPnP Forum. `http://www.upnp.org/`.

[48] M. Weiss. Feature interactions in web services. In *Feature Interaction in Telecommunications and Software Systems VII*, pages 149–158, July 2003.

[49] T. Yokogawa, T. Tsuchiya, M. Nakamura, and T. Kikuno. Feature interaction detection by bounded model checking. *IEICE Transactions on Information and Systems*, Vol. E86-D, No. 12, pages 2579–2587, December 2003.