IPSJ Journal Vol. 49 No. 6 1751–1762 (June 2008)

*Regular Paper*

# Validating Safety for the Integrated Services of the Home Network System Using JML

Ben Yan,[†1] Masahide Nakamura,[†2]
Lydie du Bousquet[†3] and Ken-ichi Matsumoto[†1]

The *home network system* (HNS, for short) enables the flexible integration of networked home appliances, which achieves value-added *integrated services*. Assuring safety within such integrated services is a crucial issue to guarantee a high quality of life in smart home. In this paper, we present a novel framework for the safety of the HNS integrated services. We first propose a way to define safety in the context of the integrated services, which is characterized by *local safety*, *global safety*, and *environment safety*. We then propose a method that can validate the above three kinds of safety for given HNS implementations. Exploiting the concept of *Design by Contract* (DbC, for short), the proposed method represents every safety property as a *contract* between a provider and a consumer of an HNS object. The contracts are embedded within the implementations, and then are validated through elaborate testing. We implement the method using *Java Modeling Language* (JML, for short) and JUnit with a test-case generation tool TOBIAS. Using the proposed framework, one can define and validate the safety of HNS integrated services, systematically and efficiently.

## 1. Introduction

The *home network system* (HNS, for short) is a system consisting of multiple networked household appliances and sensors. It is one of the promising applications of emerging ubiquitous technologies. The great advantage of the HNS lies in the flexible *integration* of different home appliances through the network. The integration achieves value-added *integrated services* [8]. For example, integrating a TV, a DVD player, lights, sound systems and curtains implements a *DVD Theater Service*, which allows a user to watch movies in a theater-like atmosphere within a single operation.

In developing and providing the HNS integrated services, the service provider must guarantee that the service is *safe* for inhabitants, house properties and their surrounding environment. Assuring safety is a crucial issue to guarantee a high quality of life in smart home. With the conventional (non-networked) home appliances, safety is ensured manually by the human user. That is, every user is supposed to follow the *safety instructions* typically described in the user's manual.

With the HNS integrated services, however, we have to consider the safety much more carefully. First, the networked appliances are operated *automatically* by software agents, but not by the human user. Second, the integration of multiple appliances yields global dependencies between the appliances. Moreover, the residential safety rules of every home, which are independent of appliances and services, should also be concerned. Most of these issues must be coped with carefully in the software implementation. Unfortunately however, there exists no solid framework to handle the safety of the HNS integrated services, as far as we know.

In this paper, we propose a novel framework for the safety, consisting of two contributions. The first contribution is to propose a way to *define* the safety of the HNS integrated services. Considering the nature of the HNS and integrated services, we define three kinds of safety: (1) *local safety* is the safety to be ensured by the safety instructions of individual appliances, (2) *global safety* is specified over multiple appliances as required properties of an integrated service, and (3) *environment safety* is prescribed as residential rules in the home and surrounding environment.

Our second contribution is to propose a method that *validates* the above three kinds of safety for given HNS implementations. For this, the proposed method uses the technique of *Design by Contract* (DbC, for short) [7], extensively. In general, the HNS involves multiple stakeholders (e.g., appliance vendors, service providers, house builders, end users, etc.). It is essential to find out who is responsible in each instance for the safety issue. We consider every safety property as a *contract* between a provider and a consumer of an HNS object.

In the proposed method, the contracts for local (global, or environment) safety

†1 Nara Institute of Science and Technology (NAIST)
†2 Kobe University
†3 Joseph Fourier University (Grenoble I)

are embedded within the implementations of the *appliance* (*service*, or *home*, respectively) objects. Following this, the contracts are validated through elaborate testing. In this paper, especially for the HNS written in Java, we implement the method with *JML* (*Java Modeling Language*) [10] and JUnit. In order to cover all possible scenarios where the integrated service is activated, we also introduce a tool TOBIAS for the combinatorial test-case generation.

To evaluate the feasibility, we also conduct case studies for practical integrated services. It is demonstrated that the proposed method can automatically detect logical faults that violate safety properties, with in a necessarily short time.

Using the proposed framework, one can define and validate the safety of HNS integrated services, systematically and efficiently. We believe that the proposed method would be a powerful means not only for validating given services, but also for providing solid safety guidelines to stakeholders of the HNS.

## 2. Preliminaries

### 2.1 Home Network System (HNS)

An HNS consists of one or more *networked appliances* connected to a LAN at home. In general, each appliance has a set of *application program interfaces* (APIs), by which the users or external software agents can control the appliance via the network. An HNS typically has a *home server*, which manages all the appliances in the HNS. Services and applications are installed on the home server. An *HNS integrated service* operates multiple different appliances together, and achieves a sophisticated and value-added service. An integrated service is implemented as a software application that invokes APIs of the appliances.

### 2.2 Example of Integrated Services

Here we introduce three example scenarios of HNS integrated services.

**[SS1: DVD Theater Service]** Integrating a TV, a DVD player, a sound system, a light and a curtain, this service automatically sets up the living room in a theater configuration. Upon a user's request, the TV is turned on with the DVD input, the curtains are closed, the sound system is configured for 5.1ch mode, the light darkens, and finally the DVD player plays back the contents.

**[SS2: Relax Service]** Integrating a DVD player, a sound system, a light, and an air-conditioner, this service helps a user relax in the living room. When the user starts the service, the DVD player is turned on in music mode, a 5.1ch speaker is selected with an appropriate sound level, the brightness of the light is adjusted, the air-conditioner is configured with a comfortable temperature.

**[SS3: Cooking Preparation Service]** Integrating a gas-valve, a ventilator, a kitchen light, and an electric kettle. This service automatically sets up the kitchen configuration in preparation for cooking. When requested, the kitchen light is turned on, the gas-valve is opened, the ventilator is turned on, and the kettle is turned on with a boiling mode to prepare hot water for cooking.

### 2.3 Object-Oriented Model for HNS

To understand the HNS clearly, here we introduce an object-oriented model of HNS shown in **Fig. 1**, which has been proposed in our previous work [5),9)]. As represented with the UML class diagram, the model consists of three kinds of objects (classes): `Appliance`, `Service`, and `Home`. These classes have relationships such that (R1) a `Home` has multiple `Appliance`s, (R2) a `Home` has multiple `Service`s, and (R3) a `Service` uses multiple `Appliance`s, which reasonably characterize the structure of an HNS.

**(A) Appliance Object**

An appliance object models a networked appliance. The model involves a super class `Appliance`, which aggregates attributes and methods commonly contained in all kinds of electric appliances. It also has a `Specification`, which stores static specification information such as power voltage, rated current, size, allowable temperature and humidity. Typical methods involve the power switches (`on()`, `off()`), getting the current power consumptions (`getCurrentConsumption()`).

On the other hand, operations specific to each kind of appliance are specified in the concrete appliance classes. Such methods include `TV.setChannel()`, `DVD.play()` and `Kettle.openLid()`. Every appliance also has a method that returns the current state of the appliance (e.g. `TV.getTvStatus()`), so that the state can be referred by external objects.

**(B) Service Object**

A service object models an integrated service, which *uses* several appliance objects. There is a super class `Service` which has common interfaces like `getStatus()`. The concrete service scenarios are implemented in sub-classes that inherit `Service`. Specifically, each service contains a set of appliance ob-

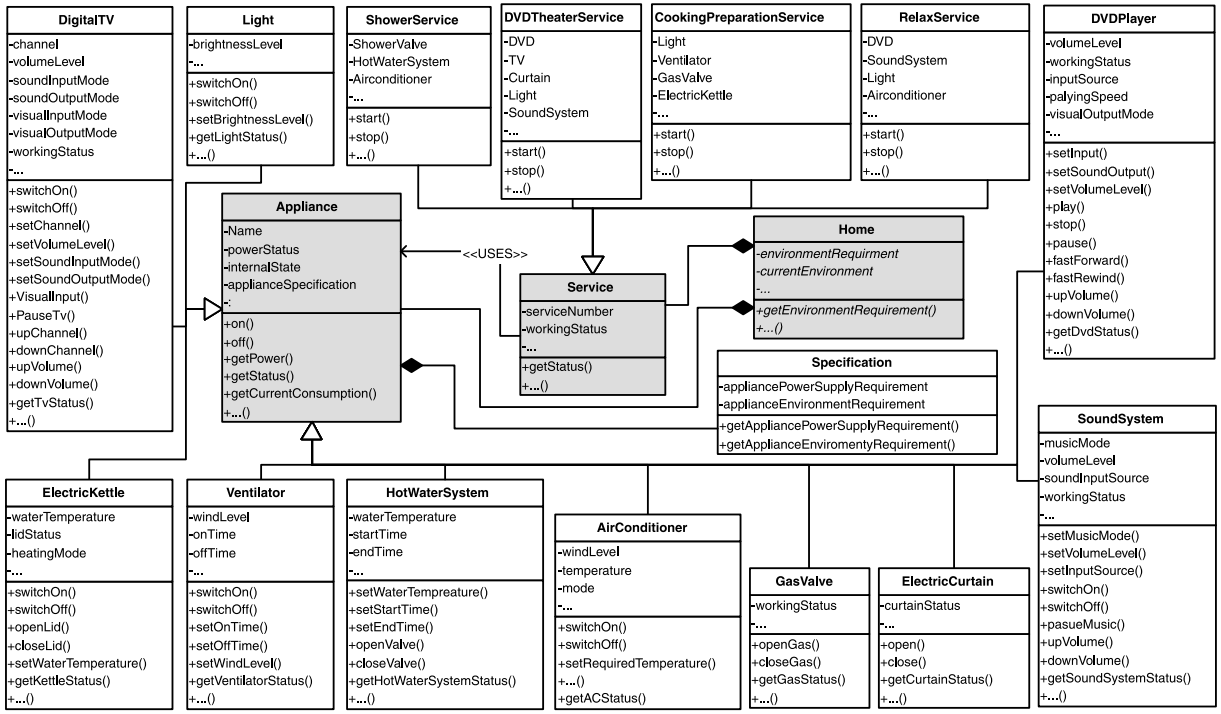


**Fig. 1** Object-oriented model of HNS.

jects, and invokes methods of the appliance objects according to a certain logic. **Figure 2** shows a Java implementation of the `DVDTheaterService`. It can be seen in `start()` that the scenario SS1 (see Section 2.2) is implemented.

**(C) Home Object**

A home object, represented as a singleton object `Home`, models the house that involves *environmental attributes*. The attributes include energy consumption, sound level, brightness, temperature and humidity. We assume that values of these attributes can be computed from the current states of appliances and services. For instance, the current temperature is obtained by `Home.currentEnvironment.getTemperature()`.

The current electricity consumption is computed from specifications and states of appliances that are currently on. Note that a user may want to operate some appliances directly and *not* through the integrated services. To simulate this, we assume that `Home` has methods that can directly invoke any appliance methods.

## 3. Formalizing the Safety of the HNS Integrated Services

### 3.1 Safety in a Broad Sense

Our first concern is to see what *safety* is. In a broad sense, the safety of an integrated services could be defined as follows:

**Definition 1 (safety in broad sense)** An HNS integrated service $s$ is *safe* iff $s$ is free from any condition that can cause [injury or death to home users and neighbors], or [damage to or loss of home equipment and the surrounding

```
public class DVDThaterService {
    DigitalTV tv; DVDPlayer dvd; SoundSystem sound;
    Light light; ElectricCurtain curtain; //Appliances used.
   public class DVDTheater(DigitalTV tv, DVDPlayer dvd,
     SoundSystem sound, Light light, ElectricCurtain curtain){
    this.tv = tv; this.dvd = dvd; this.sound = sound;
    this.light = light; this.curtain = curtain; //Constructor
   }
   public void start() { // Initiate DVDTheater
    tv.on();                        /* Turn on TV */
    tv.visualInput('DVD');
    dvd.on();                       /* Turn on DVD player */
    dvd.setSoundOutput('5.1');
    sound.on();                     /*Turn on Sound System*/
    sound.setInput('DVD');
    sound.setVolumeLevel(25);
    curtain.close();                /*Close curtain*/
    light.setBrightnessLevel(1);    /*Minimize brightness*/
    dvd.play();                     /*Play DVD*/
   }
   :
}
```

**Fig. 2** Java Implementation of DVD Theater Service.

environment].

Our long-term goal is to establish a solid framework that can guarantee the safety in Definition 1. In general however, it is quite difficult to achieve 100% safety. Hence, safety is often evaluated by means of *risk*. To assure safety to a considerable extent, a set of conditions or guidelines minimizing the risk (called, *safety properties*) are usually considered [2]. Considering the nature of the HNS, we propose to classify the safety properties into the following three types.

### 3.2 Local Safety Property

For every electric appliance, the vendor of the appliance prescribes a set of *safety instructions* for the proper and safe use of the appliance. Conventionally, these instructions have been designated for human users. In the HNS however, the instructions must be guaranteed within the software that uses the appliance. For instance, the following property is taken from a safety instruction of an electric kettle.

*L1: Do not open the lid while the water is boiling, or there is a risk of scald.*

Any integrated service using the kettle (e.g., SS3 in Section 2.2) must be implemented so that the service never opens the lid while the kettle is in boiling mode. Usually the safety instructions of an appliance can be regarded as a set of safety properties that are *locally* specified within that appliance only. Thus we define them as *local safety properties* as follows:

**Definition 2 (local safety property)** A safety property $lp$ is called *local safety property*, iff $lp$ is defined over a single appliance $d$. Let $LocalProp(d) = \{lp_1, lp_2, \ldots, lp_m\}$ be the set of all local safety properties with respect to the appliance $d$. For a given integrated service $s$, let $App(s) = \{d_1, d_2, \ldots, d_n\}$ be the set of networked appliances used by $s$. Then, we define $LocalProp(s) = \cup_{d_i \in App(s)} LocalProp(d_i)$ which is the set of local safety properties with respect to the service $s$.

### 3.3 Global Safety Property

Since an integrated service operates multiple appliances, it is also necessary to consider *global dependencies* among different appliances. For instance, the following safety property is for the SS3 to avoid carbon monoxide poisoning.

*G1: While the gas valve is opened, the ventilator must be turned on.*

Note that the property *G1* requires a global dependency between the gas valve and the ventilator. These kinds of safety properties are often *service-specific*, and are not covered by the local safety properties of individual appliances. So we define them as *global safety properties*.

**Definition 3 (global safety property)** A safety property $gp$ is called *global safety property* iff $gp$ is defined over multiple appliances $d_1, d_2, \ldots, d_n$ that are used by an integrated service $s$. $GlobalProp(s) = \{gp_1, gp_2, \ldots, gp_k\}$ denotes the set of all global safety properties for $s$.

### 3.4 Environment Safety Property

In general, each house has a set of *residential rules* for inhabitants to create a safe living environment. For instance, most houses have a capacity for electricity. In addition, a community might have a rule for noise control at night:

*E1: The total amount of current used simultaneously must not exceed 30 A.*

*E2: Do not make a loud noise after 9 p.m.*

These residential rules might vary from house to house, but they are usually

*independent* [⋆1] of appliances or services in the house. The integrated services of course have to conform to the rules to be safe in the environment.

**Definition 4 (environment safety property)** A safety property *ep* is called an *environment safety property* iff *ep* is defined as an environmental or residential constraint, which is independent of any appliances or services. $EnvProp = \{ep_1, ep_2, \ldots, ep_l\}$ denotes the set of all environment properties.

### 3.5 Who Gives Safety Properties?

Since an HNS involves several stakeholders, it is important to clarify who is responsible for specifying the safety properties. In this paper, we assume that $LocalProp(s)$ are given by *appliance vendors*, since the properties are derived based on the safety instructions of the appliances. Next, $GlobalProp(s)$ is supposed to be given by the *service provider* of $s$, since the provider is responsible for the integration of appliances. Finally, we assume that $EnvProp$ should be given by the *house builder*, considering the house specification and the surrounding community rules.

### 3.6 Safety of HNS Integrated Services

Based on the discussion above, we define three kinds of safety as follows.

**Definition 5 (safety of integrated service)** For a given integrated service $s$, and a set $P$ of safety properties, we write $s \vdash P$ iff $s$ satisfies all properties contained in $P$. Then, we define the safety of $s$ as follows.

- $s$ is *locally safe* iff $s \vdash LocalProp(s)$.
- $s$ is *globally safe* iff $s \vdash GlobalProp(s)$.
- $s$ is *environmentally safe* iff $s \vdash EnvProp$.
- $s$ is *safe* iff $s$ is locally, globally and environmentally safe.

## 4. Exploiting Design by Contract (DbC) for Safety Validation of HNS

### 4.1 Safety Validation Problem

Once the safety is defined, our next concern is how to *validate* it. Specifically, the problem is formulated as follows:

⋆1 Here "independent" means that the definition of each environment property does not require the direct reference of appliances or services. In reality, each environment property becomes true or false, indirectly depending on the status of appliances and services.

**Definition 6 (safety validation problem)** Let $h$ be a given implementation of HNS and $s$ be an integrated service. Let $LocalProp(s)$, $GlobalProp(s)$, and $EnvProp$ be given. The *safety validation problem* is to check if $s$ is safe.

We assume that the given $h$ and $s$ are implemented based on the object-oriented model presented in Section 2.3.

### 4.2 Key Idea: Introducing DbC

As seen before, an HNS consists of many heterogeneous objects, and the safety properties defined over the objects are given by multiple stakeholders. So, when a safety violation occurs in an object, say *obj*, it is not always easy to prove which stakeholder is to be blamed for the accident. To do this rigorously, the consumer and the provider of *obj* must agree with mutual responsibilities before *obj* is used. This motivated us to describe every safety property as a *contract* to be fulfilled between the consumer and the provider of any HNS object. To implement this, we borrow a software design strategy called *Design by Contract* (DbC, for short) [7].

For a given program, the DbC describes properties, conditions and invariants as a set of contracts between calling and callee objects. The contracts are verified during the runtime of the program under testing. During the execution, if a contract is broken, an exception is thrown or an error is reported. Thus, if we could successfully represent the safety properties as DbC contracts among the HNS objects, then the safety validation problem can be reduced to the testing of the HNS implementations.

### 4.3 Guidelines for Describing Safety Properties as DbC Contracts

There are three types of contracts in the DbC.

**[Pre-Condition:]** A pre-condition of a method $m$ is a condition that must be satisfied *before* executing $m$.

**[Post-Condition:]** A post-condition of a method $m$ is a condition that must be satisfied *after* executing $m$.

**[Class Invariant:]** A class invariant of a class $c$ is a condition that must be guaranteed (i.e., kept unchanged) no matter which methods in $c$ are executed.

#### 4.3.1 Choosing the Type of Contract

We first consider which type of the DbC contracts is appropriate for representing a given safety property. By definition, a pre-condition characterizes a

*premise* of an API $m$. Therefore, we represent any safety property that must be satisfied by the *consumer side* of $m$ as a pre-condition. On the other hand, a post-condition characterizes a *conclusion* of $m$. We describe any safety property to be guaranteed by the *provider side* of $m$ in a post-condition. For a safety property that must hold globally without depending on any specific APIs, we use the class invariant to represent it.

#### 4.3.2 Choosing an Object for Contract

Suppose that a safety property $p$ is represented as a DbC contract $c_p$. We then have to choose an appropriate object (class) in Fig. 1 where $c_p$ is embedded. For this, we propose the following criteria based on the property type:

- If $p \in LocalProp(s)$, then embed $c_p$ in `Appliance` or its sub-classes.
- If $p \in GlobalProp(s)$, then embed $c_p$ in `Service` or its sub-classes.
- If $p \in EnvProp$, then embed $c_p$ in `Home` or its sub-classes.

The above criteria is quite reasonable, considering the definition of each type of safety properties and the role of each class in the object-oriented model.

### 4.4 Examples

Based on the guideline above, let us describe some contracts for the safety properties presented in Section 3.

First, let us take the safety property *L1*. Since *L1* is a local property related to an operation "open the lid" of `ElectricKettle`, *L1* can be represented as the following contract:

```
Target Method:  ElectricKettle.openLid()
Pre-condition:  heatingMode != 'boiling'
Post-condition: lidStatus == 'open' && heatingMode != 'boiling'
```

The pre-condition is stating a consumer-side responsibility that any service that invokes `openLid()` must ensure the kettle is not in boiling mode before executing it. In addition, the post-condition prescribes a provider-side responsibility that when the method is completed, the lid is surely opened and the status never changes to boiling mode.

The next example is the property *G1*. Since *G1* is a global property, the contract should be embedded in the service class, `CookingPreparationService`. The property should be satisfied globally no matter what method of the service is executed. So, we represent *G1* as the following class invariant.

```
Target Class: CookingPreparationService
Invariant:   GasValve.workingStatus=='open'
          -> Ventilator.powerStatus=='ON'
```

The above contract prescribes a condition that at any time when the gas valve is opened, the ventilator must be turned on. Note that it is not appropriate to embed the contract in the appliance class, e.g., `GasValve`. Since `GasValve` does not know what other appliances exist in the HNS, it is difficult for `GasValve` to obtain the status of the `Ventilator` directly.

The last example is the property *E1*. Since it is an environment property, the contract is embedded in `Home` class. Since the property does not depend on any specific APIs, we represent *E1* as a class invariant.

```
Target Class: Home
Invariant:    Home.currentEnvironment.getTotalConsumption() <= 30
```

The method `getTotalConsumption()` is supposed to return the current total consumption of electricity, which is computed from the appliances that are being turned on.

## 5. Implementing Safety Validation

Based on the discussion above, we implement a method of safety validation, especially for the Java implementations of the HNS.

**Figure 3** depicts the overview of the proposed method. The method mainly consists of the following three steps.

**Step1:** Describe the DbC contracts in JML.
**Step2:** Generate test cases.
**Step3:** Run the test.

### 5.1 Step1: Describing DbC Contracts in JML

The proposed method extensively uses the JML (*Java Modeling Language*) [3),10)] to implement the DbC-based safety validation. The JML is a specification language that can be used to describe the DbC contracts in the form of Java comments, called *JML annotations*.

In Step 1, for each safety property $p$ given, we represent DbC contract $c_p$ in the JML annotation, and embed $c_p$ to the Java source code according to the guideline in Section 4.3.

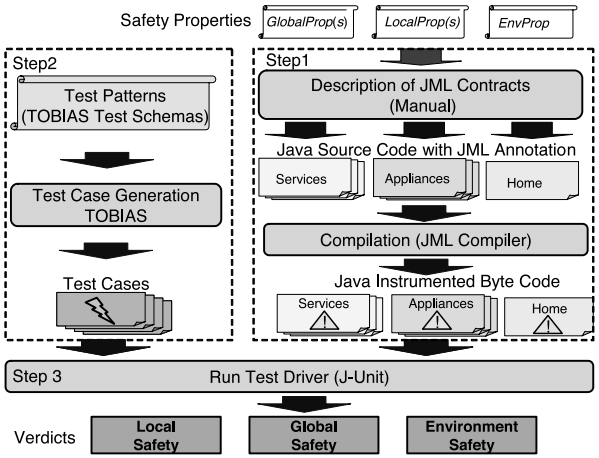


**Fig. 3** Proposed safety validation method.

```
public class ElectricKettle extends Appliance{
 private /*@spec_public@*/ String heatingMode;//BOILING or IDLE
 private /*@spec_public@*/ String lidStatus;  //OPEN or CLOSE
…
// --- Contract L1: ---
// Do not open the lid while the kettle is in the boiling mode.
/*@ assignable lidStatus;
  @ requires   heatingMode.equals("OFF");
  @ ensures lidStatus.equals("OPEN")&&!heatingMode.equals("BOILING");
  @*/
 public void openLid(){
   //Implementaion ……
…
}
```

**Fig. 4** `ElectricKettle.java` with JML annotations.

**Figure 4** shows the JML annotation describing contracts for the property *L1* of `ElectricKettle` (see Section 4.4). The contracts are described in comment lines just above method `openLid()`. The line starting with `requires` (or `ensures`) represents the pre-condition (or post-condition, respectively). The word `spec_public` is for exporting the attribute to be used in the JML annotation. Just for convenience, we describe `lidStatus` and `heatingMode` as simple string variables.

As shown in Fig. 3, the source code with the JML annotations is then compiled by the *JML compiler* into *instrumented bytecode*, implementing assertion-based checking routines of the DbC contracts.

Note that one might want to encode DbC contracts directly in the source code using the Java assertions. However, the great advantage of using JML rather than Java assertions is that we can completely *separate* the contracts from the implementation. Thus, the safety properties can be specified in the code as comment lines, without modifying the original code.

### 5.2 Step 2: Generating Test Cases

The next step is to construct *test cases* used for safety validation. In real life, integrated services can be activated under various situations (i.e., *states*) in the HNS. For instance, the `CookingPreparationService` may be activated when all the appliances are off. Or, it may be activated when the ventilator is already on and the lid of the kettle is initially opened. `CoolingPreparationService` must be implemented so that *the service behaves safely, in whatever state it is activated*. Therefore, for a given integrated service $s$, we generate test cases activating $s$ under all possible states.

To generate such test cases systematically and efficiently, we use a combinatorial test generation tool, called TOBIAS [1),4)]. TOBIAS allows us to define *abstract test patterns*, in which similar operations and parameter values are managed by sets, called *groups*. The groups are combined algebraically based on regular expressions, to construct more sophisticated *test schemas*. The test schemas are then translated by TOBIAS into a large set of executable test cases for JUnit [11)], where all elements in each group are *unfolded*.

Now we present our key idea. Suppose that $s$ is a given integrated service and that we want to validate a method $s.m()$. Then, we construct the following TOBIAS test schema $T$:

$$T \ ::= \ Init \ ; \quad AppOp^n \ ; \quad s.m()$$

In the schema, ; represents a sequential operator. *Init* represents a group containing initialization operations (typically constructors of HNS objects, or settings of environment parameters). *AppOp* is a group containing any appliance operations (methods). $AppOp^n$ means the $n$-time product of *AppOp*, which characterizes all possible sequences $m_1; m_2; \ldots; m_n$ ($m_i \in AppOp$). Thus, the part "$Init; AppOp^n$" is a *preamble* to generate possible states before $s.m()$ is

```
Group HomeInit ::= {begin seq Home home := new Home() end seq }

Group KettleOperation ::= {begin seq home.OffKettle() end seq,
 begin seq home.SwitchOnKettle()  end seq ,
 begin seq home.SwitchOffKettle() end seq ,
 begin seq home.CloseLidKettle()  end seq ,
 begin seq home.OpenLidKettle()   end seq ,
 begin seq home.SetWaterTemperatureKettle(98) end seq };

Group threeKettleOperations ::= KettleOperation^3..3

Group testAllKettleOperations::= { begin seq
  HomeInit ;threeKettleOperations
end seq }
```

(a) Test Schemas for ElectricKettle

```
Group testCookingWRTKettleOp ::= { begin seq HomeInit;
  threeKettleOperations; home.CookingPreparationService.start()
end seq }

Group testEachServiceWRTKettleOp ::= { begin seq HomeInit;
  threeKettleOperations; OneActivationForEachService
end seq }

Group HomeInitEnv ::= {
 begin seq Home home := new Home(tempValues,12,10, 8,0 ) end seq ,
 begin seq Home home := new Home(21,brightValues,10,8,0 ) end seq,
 begin seq Home home := new Home(22,1,volumeValues,9,15 ) end seq,
 begin seq Home home := new Home(17,20,5,timeValues,13 ) end seq,
 begin seq Home home := new Home(19,2,15,11,powerValues) end seq
};
Group testEachServiceWRTevironmentSettings ::= {begin seq
 HomeInitEnv; OneActivationForEachService
end seq }
```

(b) Test Schemas for CookingPreparationService and other services

**Fig. 5** TOBIAS test schemas.

activated. Note that the preamble can also be used as test schemas of individual appliances.

**Figure 5** (a) shows an example of TOBIAS schemas, which define the preamble using operations of `ElectricKettle`. In the figure, `HomeInit` creates a `Home` object. `KettleOperation` contains 6 methods for operating the kettle in `home` from outside (see Section 2.3 (C)). `threeKettleOperaions` is 3-time product of `KettleOperation`. Concatenating `HomeInit` and `threeKettleOperations` yields a preamble of the proposed method. Note that the preamble can be used to test `ElectricKettle` itself, we name the schema `testAllKettleOperations`, from which 216 ($= 6 \times 6 \times 6$) test cases will be unfolded.

Figure 5 (b) shows TOBIAS schemas for testing integrated services. The first schema tests the activation of `CookingPreparationService` by using `threeKettleOperations` as its preamble. The second one deals with one activation of any integrated service (definition of the group is omitted due to the space). The third schema initializes `Home` with varying environment parameters, under which safety properties concerning the environment can be validated thoroughly.

The test schemas are then translated by TOBIAS into JUnit test classes. For instance, from the group `testEachServiceWRTKettleOp`, the total 1512 JUnit tests ⋆1 have been automatically generated within just 0.641 sec. (in a mid-class PC, Pentium-M 1 GHz, 760 MB). Thus, using TOBIAS we can manage a large number of test cases in a very compact form, which significantly reduces the cost of test case generation.

### 5.3 Step 3: Running Test

In this step, we conduct the test using the *JUnit* testing framework for Java. According to the test cases obtained in Step 2, JUnit automatically runs the instrumented bytecode under test. During the execution, if any contract is broken, then a JML exception is thrown to JUnit. Then, we can get a report about which safety property is violated. Thus we can solve the safety validation problem in Section 4.1.

## 6. Case Study

To evaluate the proposed method, we have conducted safety validation for a practical HNS and integrated services.

### 6.1 Preparations

For the experiment, we have prepared Java implementations of 11 appliances and 7 integrated services, as follows:

**Appliances:** `DVDPlayer`, `AirConditioner`, `ElectricKettle`, `Ventilator`, `Light`, `Door`, `SoundSystem`, `TV`, `GasValve`, `Blind`, `Curtain`.

**Integrated Services:** `CookingPreparation`, `Relax`, `DVDTheater`, `Shower`, `Blind`, `Sleep`, `ComingHome`.

Although the appliance classes behave as virtual appliances without hardware

⋆1 `OneActivationForEachService` contained 7 integrated services.

**Table 1** Validated safety properties in the case study (only a part).

| Safety Type | Safety Property |
|---|---|
| *LocalProp* | *L1*: Do not open the lid while the water is boiling or there is a risk of scald (for ElectricKettle).<br>*L2*: Make sure that the lid is closed before boiling water (for ElectricKettle).<br>*L3*: Do not connect it with outlets except exchange 100 V (for all appliances).<br>*L4*: Do not connect it with an outlet of rating 15A or less (for all appliances). |
| *GlobalProp* | *G1*: While the gas valve is opened, the ventilator must be turned on (for cookingPreparationService).<br>*G2*: Do not change the temperature while the shower is open (for shower service).<br>*G3*: When the service turns on the shower valve, the water temperature of the gas-boiler must be between 35 and 45 degrees (for shower service). |
| *EnvProp* | *E1*: The total amount of current used simultaneously must not exceed 30 A.<br>*E2*: Do not make a loud noise or sound after 9 p.m.<br>*E3*: Unlock doors and windows in the event of fire. |

**Table 2** Results of safety validation.

| TOBIAS Test Schemas | # of Total TC | # of Failed TC | Time Elapsed | Safety Violation |
|---|---|---|---|---|
| `testAllKettleOperations` | 216 | 78 | 1.5 sec. | *L1*, *L2* |
| `testCookingWRTKettleOp` | 216 | 216 | 2.265 sec. | *G1* |
| `testEachServiceWRTKettleop` | 1,512 | 0 | 25.892 sec. | none |
| `testEachServiceWRTevironmentSettings` | 231 | 18 | 2.719 sec. | *E1* |

devices, their source code has been partially taken from the service layer of a real HNS [8] under operation in our laboratory.

In the source code, we then inserted the total 209 JML annotations (17 pre-conditions, 150 post-conditions, and 42 invariants). **Table 1** shows a part of validated safety properties. The experiment has been performed in a PC with Pentium-M 1 GHz, 760 MB RAM, Windows XP Pro, J2SDK 1.4.2, JUnit 3.8.1, JML tools release 5.3 and TOBIAS Eclipse plug-in.

### 6.2 Experiment

The safety validation experiment has been conducted based on *incremental testing*. That is, taking one TOBIAS test schema at a time, we ran the generated test cases. If the proposed method detected any safety violation, we fixed the related faults in the source code, and then tested the revised version again. If all the test cases were passed, we took the next test schema to validate. Thus, the HNS implementations have been incrementally updated to a safer version for each run of testing.

### 6.3 Results

**Table 2** summarizes the validation statistics of each test schema. The table contains the total number of test cases generated from the schema, the number of test cases failed, the time taken for testing, and the safety properties violated during the test. Due to limited space, the table shows only some interesting results. For each safety violation detected, we explain the cause of the violation as follows.

**(A) Violation of *L1* and *L2***

The test cases from `testAllKettleOperations` revealed violations of local safety properties *L1* and *L2* within the original implementation of `ElectricKettle` class. First, the violation of *L1* was due to the omission of checking the heating mode in `openLid()` method. Therefore, a sequence `kettle.switchOn(); kettle.openLid()` leads to an unsafe situation where the lid is opened during the boiling mode.

The violation of *L2* was caused by a logical error in method `kettle.switchOn()`, which sometimes bypassed the checking of the lid status.

Hence in some sequences, the kettle entered boiling mode without checking if the lid was surely closed. We fixed the errors in `ElectricKettle` class before proceeding to the next test schema.

**(B) Violation of *G1***

In the validation of `testCookingWRTKettleOp`, the proposed method detected the violation of the global safety property *G1* within the `CookingPreparationService` class. The code inspection revealed that the invocation of `ventilator.setVentilatorLevel()` was omitted in the service. Hence, the ventilator did not start the fan although the power of the ventilator was on.

**(C) Violation of *E1***

We have found that the environment safety property *E1* was violated in some test cases from `TestEachServiceWRTevironmentSettings`. When the total power consumption was close to maximum, if `DVDTheaterService` (or `RelaxService`) was activated, the consumption exceeded 30 A. To assure environment safety, the home should have a mechanism that estimates the total consumption before every integrated service is activated.

## 7. Discussion

### 7.1 Summary of Contribution

We have proposed a comprehensive framework that can define and validate the safety of HNS integrated services. We believe that our safety definitions are reasonable and the idea of introducing DbC for safety validation fits well the nature of HNS. Thanks to JML, we have developed a safety validation method that can be directly applied to implementations written in Java. By using powerful tools such as JUnit and TOBIAS, a major portion of the validation can be automated. As demonstrated in the case study, the time taken for each test was very short. Thus, the proposed method is quite promising for many other practical settings.

### 7.2 Limitations

One limitation is that sophisticated TOBIAS schemas may yield the *combinatorial explosion* of test cases. As a result, TOBIAS generates so many test cases that the Java VM cannot accept them for execution. For such complex schemas, we need a technique to prune irrelevant test cases.

Another limitation is that we have not yet considered the case where two or more integrated services are executed simultaneously. When multiple services are executed, we need to consider functional conflicts among the services. This is called the *feature interaction problem* [9]. We leave these limitations for our future work.

### 7.3 Related Work

Despite their importance, the safety issues have not been well studied yet in the ubiquitous computing area. As far as we know there exists only a small amount of research work related to our contribution.

Yang, et al. [13] proposed a programming model that identifies safe and unsafe contexts in a smart home. Using standard ontology, the model builds a *context graph* enumerating all possible states, where each state is marked as desirable, transitional, or impermissible. Since the graph is constructed to quite a higher level of abstraction, it cannot be applied directly to the safety validation problem at the implementation level.

Pattara, et al. [5] proposed a method that verifies the functional properties of HNS integrated services based on *model checking*. The method gives an automatic and complete proof if given properties hold against an abstract HNS model defined in a finite state space. However, properties proven in the formal HNS model are not necessarily preserved in the HNS implementation, which is the limitation of the formal verification.

Traditionally, the safety issues have been addressed in *safety critical systems* [6], such as aerospace systems and nuclear plants. In such a safety critical system, all components in the system tend to be tightly coupled with each other under a fixed environment, in order to provide proprietary services. Thus, there is only local safety. This is quite different from the HNS (or even general ubiquitous applications), where the combinations of the components vary considerably for different purposes. So, we consider that our notions of global safety and environment safety are specific to ubiquitous applications.

The original idea of characterizing safety in the HNS has been published in our conference paper [12]. We have made some significant improvements in this version: (1) the refinement of the safety definitions, (2) the addition of the imple-

mentation details with the JML, and (3) the evaluation with a case study (i.e., the new part is after Section 4.3 to the end). We believe that these new results have substantial value.

### 7.4 Future Work

We plan to investigate a more systematic way of translating any safety property into DbC contracts. We also want to generate efficient test schemas that cover error-prone scenarios. Safety validation considering the feature interaction problem is also a challenging issue for our future research.

**Acknowledgments** This research was partially supported by the Comprehensive Development of e-Society Foundation Software program of the Ministry of Education, Culture, Sports, Science and Technology, the Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No.18700062) and Scientific Research (B) (No.17300007), and by JSPS and MAE under the Japan-France Integrated Action Program (SAKURA).

## References

1) du Bousquet, L., Ledru, Y., Maury, O. and Bontron, P.: A case study in JML-based software validation, *Proc. 19th IEEE International Conferences on Automated Software Engineering* (*ASE'04*), Linz, pp.294–297, IEEE Computer Society Press (Sep. 2004).
2) International Electrotechnical Commission, Household and similar electrical appliances — Safety, IEC 60335-1 (Sep. 2006).
3) Leavens, G.T. and Cheon, Y.: Design by Contract with JML, available from www.jmlspecs.org (May 2006).
4) Ledru, Y., du Bousquet, L. Maury, O. and Bontron, P.: Filtering TOBIAS combinatorial test suites, *Proc. International Conferences on Fundamental Approaches to Software Engineering* (*ETAPS/FASE'04*), LNCS 2984, Springer-Verlag (Mar. 2004).
5) Leelaprute, P., Nakamura, M., Tsuchiya, T., Matsumoto, K. and Kikuno, T.: Describing and Verifying Integrated Services of Home Network Systems, *Proc. 12th Asia-Pacific Software Engineering Conferences* (*APSEC 2005*), pp.549–558 (Dec. 2005).
6) Leveson, N.: Safeware: System Safety and Computers, Addison-Wesley (1995).
7) Meyer, B.: Applying Design by Contract, *IEEE Computer*, Vol.25, No.10, pp.40–51 (Oct. 1992).
8) Nakamura, M., Tanaka, A., Igaki, H., Tamada, H. and Matsumoto, K.: Adapting Legacy Home Appliances to Home Network Systems Using Web Services, *Proc. International Conferences on Web Services* (*ICWS 2006*), pp.849–858 (Sep. 2006).
9) Nakamura, M., Igaki, H. and Matsumoto, K.: Feature Interactions in Integrated Services of Networked Home Appliances — An Object-Oriented Approach, *Proc. International Conferences on Feature Interactions in Telecommunication Networks and Distributed Systems* (*ICFI'05*), pp.236–251 (July 2005).
10) The Java Modeling Language (JML), available from www.eecs.ucf.edu/~leavens/JML/.
11) JUnit, Testing Resources for Extreme Programming, available from www.junit.org/.
12) Yan, B., Nakamura, M., du Bousquet, L. and Matsumoto, K.: Characterizing Safety of Integrated Services in Home Network System, *Proc. 5th International Conferences on Smart homes and health Telematics* (*ICOST2007*), pp.130–140 (June 2007).
13) Yang, H.-I., King, J., Helal, S. and Jansen, E.: A Context-Driven Programming Model for Pervasive Spaces, *Proc. 5th International Conferences on Smart homes and health Telematics* (*ICOST2007*), pp.31–43 (June 2007).

(Received September 2, 2007)
(Accepted December 4, 2007)
(Original version of this article can be found in the Journal of Information Processing Vol.16, pp.38–49.)

**Ben Yan** received the B.E. degree from Henan University of Science and Technology, China, in 1999, and the M.E. degree from Department of information Science, Okayama University of Science, Japan, in 2006. He is currently a Ph.D. student of the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. His research interests include the V&V of home network systems, and requirements engineering for safety critical systems. He is a student member of IEICE and IPSJ.

**Masahide Nakamura** received the B.E., M.E., and Ph.D. degrees in Information and Computer Sciences from Osaka University, Japan, in 1994, 1996, 1999, respectively. From 1999 to 2000, he has been a post-doctoral fellow in SITE at the University of Ottawa, Canada. He joined the Cybermedia Center at Osaka University from 2000 to 2002. From 2002 to 2007, he worked for the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. He is currently an associate professor in the Graduate School of Engineering at Kobe University. His research interests include the service-oriented architecture, Web services, the feature interaction problem, V&V techniques and software security. He is a member of IEEE, the ACM and IEICE.

**Lydie du Bousquet** received a "Magistere d'Informatique" and a "Diplome d'Etudes approfondies" from Claude Bernard University, Lyon and Ecole Normale Superieure de Lyon in 1996. She was awarded a Ph.D. in Computer Science from Joseph Fourier University, Grenoble, France, in 1999. After one year in a postdoctoral position at IRISA, Rennes, France, she became an associated professor at Joseph Fourier University in 2000. Her main research interests focus on the validation of safety critical systems (in home networks systems) with testing approaches.

**Ken-ichi Matsumoto** received the B.E., M.E., and Ph.D. degrees in Information and Computer sciences from Osaka University, Japan, in 1985, 1987, 1990, respectively. Dr. Matsumoto is currently a professor in the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. His research interests include software metrics and measurement framework. He is a senior member of the IEEE, and a member of the ACM, IPSJ and JSSST.