NAIST-IS-DD0661029

Doctoral Dissertation

Characterizing, Deriving and Validating Safety Properties of Integrated Services in Home Network System

Ben Yan

December 6, 2008

Department of Information Systems
Graduate School of Information Science
Nara Institute of Science and Technology

A Doctoral Dissertation
submitted to Graduate School of Information Science,
Nara Institute of Science and Technology
in partial fulfillment of the requirements for the degree of
Doctor of ENGINEERING

Ben Yan

Characterizing, Deriving and Validating Safety Properties of Integrated Services in Home Network System*

Ben Yan

Abstract

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*. 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 has been ensured manually by the human user. That is, every user is supposed to follow the *safety instructions* typically described in the user's manual.

* Doctoral Dissertation, Department of Information Systems, Graduate School of Information Science, Nara Institute of Science and Technology, NAIST-IS-DD0661029, December 6, 2008.

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 instead of human users. 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 dissertation, we propose a novel framework for the safety, consisting of the following three 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.

The second contribution is to propose a requirement-engineering approach that can systematically derive the verifiable safety properties for the HNS based on the safety definition above. Specifically, we propose a *hazard analysis model* for the HNS, consisting of four levels of abstractions: (1) hazard context, (2) hazardous state, (3) object attribute and (4) object method. For a given HNS model and the hazard context, we then conduct a goal-oriented analysis to specify logical relations between the adjacent abstraction levels. The analysis yields cause-and-effect chains from the abstract hazard contexts to the concrete attributes and operations of HNS objects (appliances, services, environment). Finally, the safety properties and their responsible operations are derived from the complete model, which give the strong rationale of the safety of the HNS.

The third contribution is to propose a method that *validates* the three kinds of safety property which was extracted from the hazard analysis model above. For this, the proposed method uses the technique of *Design by Contract* (DbC, for short) 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 are embedded within the implementations of the *appliance* (*service*, or *home*, respectively) objects. Following this, the contracts are validated through elaborate testing. In this dissertation, especially for the HNS written in Java, we implement the method with *JML* (*Java Modeling Language*) 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.

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.

Keywords:

Home Network System, Object-Oriented Model, Integrated Service, Safety Property, Local Safety Property, Global Safety Property, Environment Safety, Require ment-engineering, Hazard Analysis Model, Hazard Template, Design by Contract, Java Modeling Language, TOBIAS

List of Major Publications

Journal Papers

- B. Yan, M. Nakamura, L. du Bousquet, and K. Matsumoto.: Validating Safety for Integrated Services of Home Network System Using JML, Journal of Information Processing Society of Japan (IPSJ Journal), Vol.49, No.6, pp.1751-1762, Jun.2008.

International Conference Papers

- B. Yan, M. Nakamura, L. du Bousquet, and K. Matsumoto.: Characterizing Safety of Integrated Services in Home Network System, Proceedings of the 5th International Conferences on Smart homes and health Telematics (ICOST2007), LNCS 4541, pp.130-140, Jun.2007.
- B. Yan, M. Nakamura, L. du Bousquet, and K. Matsumoto.: Considering Safety and Feature Interactions for Integrated Services of Home Network System, Proceedings of Doctorial Symposium on Components, Services and Features Composition or Interaction (CSF-CI) of the 9th International Conferences on Feature Interactions in Software and Communication Systems (ICFI2007), pp.213-216, Sep.2007.

- B. Yan, M. Nakamura, and K. Matsumoto.: Deriving Safety Properties for Home Network System Based on Goal-Oriented Hazard Analysis Model, Proceedings of the 3rd International Symposium on Smart Home (SH08), Dec.2008.

Domestic Conference Papers

- B. Yan, M. Nakamura, L. du Bousquet, and K. Matsumoto.: Considering Safety in Integrated Services in Home Network System, IEICE technical report, Vol.106, No.358(20061109) pp.49-54, NS2006-97, Nov.2006. (in Japanese)
- B. Yan, M. Nakamura, L. du Bousquet, and K. Matsumoto.: Describing and Verifying Safety in Integrated Services of Home Network System by JML Considering Safety, IEICE technical report, Vol.106, No.577(20070301) pp.7-12, NS2006-159, Mar.2007. (in Japanese)

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1. Background

The recent ubiquitous/pervasive technologies allow general household appliances to be connected within the network at home. The *home network system* (HNS, for short) is comprised of such networked appliances to provide various services and applications for home users [31]. The great advantage of HNS lies in *integrating* (or orchestrating) features of multiple appliances, which yields more value-added and powerful services. We call such services *HNS integrated services.*

For example, we can integrate 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 just within a single operation.

We also can integrate a gas system, a ventilator, a kitchen light and an electric kettle for building *Cooking Preparation Service.* This service can automatically set up the kitchen configuration in preparation for cooking. Such as, turn on the kitchen light, the ventilator, open the gas system and set the kettle into boiling mode etc..

In developing and providing a HNS integrated service, the service provider must guarantee that the service is *safe* for inhabitants, house properties and their surrounding environment. As for the HNS integrated services, however, we have to consider the safety much more carefully. First, the networked appliances are operated automatically by software agents instead of human user. Second, the integration of multiple appliances yields global dependencies among the appliances. Moreover, the residential safety rules of every home, which are independent of appliances and services, should be also concerned. Most of these issues must be coped with carefully in the software implementation. Thus, a single fault in the service application can cause serious accidents to the user.

Despite their importance, the safety issues have not been well studied yet in the ubiquitous computing area, including smart home. As far as we know, the safety issues have been mainly studied in critical system, such as transportation, spacecraft, aerospace systems and nuclear plants etc. [21] [22] [23] [24]. Moreover, the general solutions of the critical system do not always fit the safety issues of the home network system, because of the following reasons:

First, compared to the ubiquitous applications including HNS, the safety critical systems are quite monolithic, where requirements and system configurations are not frequently changed. On the other hand, for the home network system, the user can add a new appliance into the system, or even create a new integrated service depending on the requirements. This is quite different from the above critical system. Thus, we need alternative analysis models suitable for the object-oriented model.

Second, 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. This is quite different from the HNS, where the combinations of the components vary flexibly for different purposes. So, we have to consider the global relationships among appliances, and also the impacts to the environmental issues, which are specific to ubiquitous applications.

At last, most of safety requirement solutions for the critical system focus on the requirement engineering stage. The main purpose of such solutions is to give a method for analyzing "how to design a safety system" before implementing the system, or "why the accident happened" after the accident. Most of them do not deal with the safety issues on the implementation level. We consider that it is a weak point for conducting the safety management for the HNS domain, where the applications are developed in an agile way.

2. Goals of Research

In general, the safety issues are usually considered at *online stage* [9, 30, 35] or *offline stage*. *Online stage* always proposes a method for *watching and preventing the system into the hazard state* or considering *what measures should be took for decreasing the damage after the accident happened*. *Offline stage* always proposes a method for supporting *how to design a safe specification for a system*. In order to achieve the safety within the HNS integrated services systematically at offline stage, we set three goals for considering the safety issues of HNS form requirement design stage to test stage.

- **GOAL1: Formalizing the Safety of the HNS Integrated Services**

  The first goal 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.

- **GOAL2: Proposing a Safety Requirement Engineering Method for HNS**

  The second goal is to propose a requirement-engineering approach that can systematically derive the verifiable safety properties for the HNS. Specifically, we propose a HNS *hazard analysis model* (HNS-HAM, for short) for the HNS. The hazard analysis model consists of four levels: (1) *hazard context level*, (2) *hazardous state level*, (3) *object attribute level* and (4) *object method level*. For a given HNS model and the hazard context, we then conduct a goal-oriented analysis to specify logical relations between the adjacent abstraction levels. The analysis yields cause-and-effect chains from the abstract hazard contexts to the concrete attributes and operations of HNS objects (appliances, services, environment). Finally, the safety properties and their responsible operations are derived. Moreover, in order to enhance the reusability of the *HNS-HAM*, we will propose a notion using hazard template for applying to construct the *HNS-HAM*.

- **GOAL3: Proposing a Safety Validation Framework for Validating the Safety Properties**

  Our third goal is to propose a method that *validates* the three kinds of safety property which was extracted from the *HNS-HAM* above. For this, the proposed method uses the technique of *Design by Contract* (DbC, for short) [29], 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 are embedded within the implementations of the *appliance* (*service*, or *home*, respectively) objects. Following this, the contracts are validated through elaborate testing. In this dissertation, especially for the HNS written in Java, we implement the method with *Java Modeling Language* (JML, for short) [16] 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.

We believe that the proposed total framework can help the HNS developers significantly in designing and implementing safe HNS solutions.

3. Overview of the Dissertation

This dissertation is organized as follows:

In Chapter 2, we will introduce the *Home Network System* based on the *Object-Oriented Model* design. We also will introduce some practical integrated services scenario examples for understanding what is the integrated service of HNS. At last, in order understand well, we will introduce a language for describing the *design specification* of the HNS without considering the safety issues.

In Chapter 3, we will first discuss the safety issues of *Home Network System*. After this, we will formalize the safety of the HNS as three kinds of safety properties. At last, a formal definition of the safety validation of the HNS will be introduced.

In Chapter 4, we will propose a requirement-engineering approach which can systematically derive the safety properties for the HNS. For this approach, it includes constructing the *Hazard Analysis Model* for specific hazard context. By constructing this model, the safety properties and their responsible operations can be derived.

In Chapter 5, we will introduce a method for validating the three kinds of safety properties which was derived from the *Hazard Analysis Model* above. To evaluate the proposed method, we will conduct the case study for safety validation within a practical HNS and integrated services.

Finally, we will conclude this dissertation with a summary and make a foresight for our research with the future work in Chapter 6.

Chapter 2

Preliminaries

1. Home Network System

A *Home Network System* (HNS, for short) 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. To process the API calls, each appliance generally has embedded devices including a processor and storage. Figure 2.1 shows an example of HNS,

As seen in Figure 2.1, which consists of various networked appliances and a home server. An HNS typically has a *home server*, which manages all the appliances in the HNS. The home server typically plays a role of gateway to the external network. It also works as an application server, where services and applications are installed on the home server.

An *HNS integrated service* is implemented as a software application that invokes APIs of the appliances. By operating multiple different appliances together, the *integrated service* achieves a sophisticated and value-added service. Communications among the appliances are performed based on the underlying network

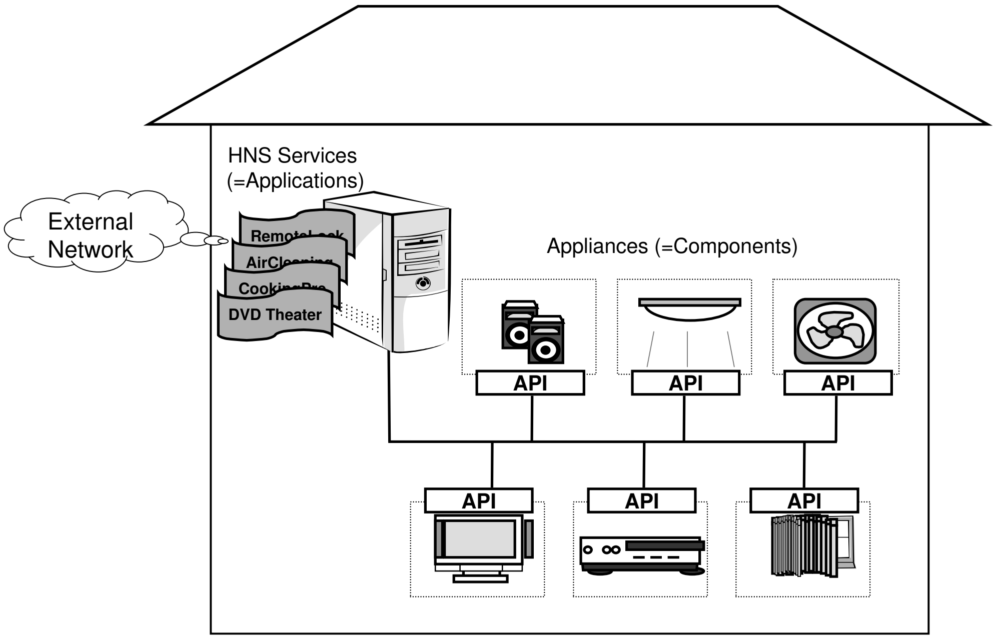


Figure 2.1. Home network system

protocol. Currently, many standard protocols are being standardized for the networked appliances. Major protocols include DLNA [28] for digital audio/video appliances and ECHONET [41] for white goods (e.g., refrigerator, air conditioners, and laundry machines) [13].

In this dissertation, we assume that the device control interface are provided in the form of *APIs*, Thus, the appliance can be operated by integrated services.

2. Examples of Integrated Services

Here we introduce four 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: Sound Healing Service**] Integrating a DVD player, a sound system, a light, and an air-conditioner, this service helps a user be relaxed 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 system, 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.

[**SS4: Bath Preparation Service**] Integrating a hot water system, a bathroom

```
Public   DVDTheaterService {
            DigitalTV     tv = new DigitalTV();
            DVDPlayer  dvd = new DVDPlayer();
            SoundSystem  sound =new SoundSystem();
            Light   light = new Light();
            Curtain  curtain = new  Curtain();

            tv.on();                                          /* Turn on TV */
            tv.setVisualInput('DVD');

            dvd.on();                                         /* Turn on the DVD Player */
            dvd.setSoundOutput('5.1');

            sound.on();                                       /* Turn on the Sound System */
            sound.setInputSource('DVD');
            sound.setVolumeLevel(25);

            curtain.closeCurtain();                           /*  Close curtain  */

            light.setBrightnessLevel(1);                      /*  Minimize brightness  */

            tv.playTv();                                      /*  Play TV   */
            dvd.playDvd();                                    /*  Play DVD  */
}
```

Figure 2.2. Java implementation of DVD theater service

light and a bathroom air-conditioner. This service could automatically set up the bath configuration in preparation for bath. While the service is implemented, the service will fill the bathtub with hot water at setting temperature, the air-conditioner will be turn on with the comfortable temperature and the bathroom light will be turned on.

Figure 2.2 shows a Java-like pseudo code which implements the scenario SS1 of DVD Theater service. In the figure, `X.Y()` means the invocation of API `Y()` of appliance `X`.

3. Object-Oriented Model for HNS

To understand the HNS clearly, here we introduce an object-oriented model of HNS [19, 32] shown in Figure 2.3.

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.

The basis of the model was built as a simplification of the implementation. Each component representing the appliances and the services was modified so that the calls to the remote control APIs were transformed into simple printing message calls (see Figure 2.4).

3.1 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

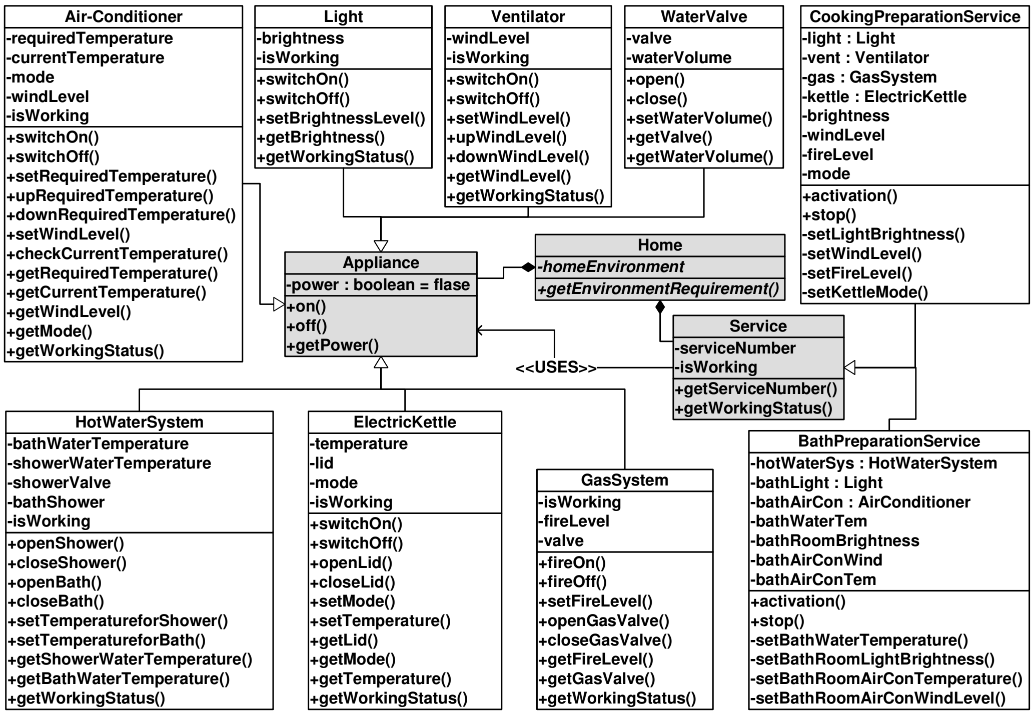


Figure 2.3. Object-oriented model of HNS

```
public class ElectricKettle extends Appliance{
...
  private /*@ spec_public @*/ String mode;         //{IDLE, BOILING, WARMING} = IDLE

  public void setMode (String wm) {                //setting working mode for electric kettle
      System.out.println (“ setMode,  kettle ”);
      this.mode = wm;
  }
...
}
```

Figure 2.4. Implementation of setModel() for electric kettle

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 status (`getPower()`). On the other hand, operations specific to each kind of appliance are specified in the concrete appliance classes. Such methods include `Light.setBrighnessLevel()`, `Air-Condition.setRequiredTemperature()` and `Kettle.openLid()`. Every appliance also has a method that returns the current state of the appliance (e.g. `Kettle.getWorkingStatus()`), so that the state can be referred by external objects.

3.2 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 `getWorkin gStatus()`. The concrete service scenarios are implemented in sub-classes that inherit `Service`. Specifically, each service contains a set of appliance objects, and invokes methods of the appliance objects according to a certain logic. Figure 2.2 shows a Java implementation of the `DVDTheaterService`. It can be seen in `activation()` that the scenario SS1 (see Section 2) is implemented.

3.3 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.homeEnvironment.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.

4. Describing HNS Specification

To capture the given HNS model more clearly, we here introduce a language for describing the *design specification* of the HNS [19] [32].

4.1 Appliance

As mentioned in Section 3.1, every appliance is characterized as an object consisting of *attributes* and *methods*. Figure 2.5 shows the general structure of appliance specification. Each attribute is defined by associated type and initial value. In our language, integer (with allowable values), boolean, or enumerate can be used for the type. For instance, `power` can take two values `ON` or `OFF`, which is initialized to `OFF`.

Each method in the specification is simply characterized by a pair of logical formula over the attributes, namely, *pre-condition* and *post-condition*. The pre-condition of a method is a guard condition that must be satisfied *before* the method is executed. On the other hand, the post-condition is a resultant condition that must be satisfied *after* the method is executed. In our model, standard comparative and logical operators [1] (== (equal), ! = (not equal), > (more than), >= (more than or equal to), < (less than), <= (less than or equal to), && (and), || (or), ! (not), => (imply)) can be used.

[1] The notation follows the one in the C (or Java) language

```
APPLIANCE Appliance_name EXTENDS Appliance

    ATTRIBUTES

       TYPE attribute_name1 : {value1 ... value2} = Init_val1;
       TYPE attribute_name2 : {value1 ... value2} = Init_val2;
       ...

    METHODS

       method_name1() {
           PRE:   Pre-Condition;
           POST:  Post-Condition;
       }

       ...

    INVARIANTS

       Invariant Condition;
```

Figure 2.5. General specification of appliance

```
APPLIANCE ElectricKettle EXTENDS Appliance

 ATTRIBUTES
   isWorking   : {true, false} = false;
   mode        : {IDLE, BOILING, WARMING} = IDLE;
   lid         : {OPEN, CLOSE} = CLOSE;
   temperature : {30,40,50,60,70,80,90,100} = 40;

 METHODS
   switchOn()  {   PRE:   power == ON && isWorking == false;      POST:  isWorking == true;   }

   switchOff() {   PRE:   power == ON && isWorking == true;       POST:  isWorking == false;  }

   openLid()   {   PRE:   power == ON && lid == CLOSE;            POST:  lid == OPEN;         }

   closeLid()  {   PRE:   power == ON && lid == OPEN;             POST:  lid == CLOSE;        }

   setMode(md) {   PRE:   power == ON;                            POST:  mode == md;          }

   setTemperature(tem){   PRE:   power == ON;                     POST:  temperature == tem;  }

   getLid()    {   PRE: power == ON;                              POST: ReturnValve == lid;   }

   getMode()   {   PRE: power == ON;                              POST: ReturnValve == mode;  }

   getTemperature()  { PRE: power == ON;                  POST: ReturnValve == temperature;   }

   getWorkingStatus(){ PRE: power == ON;                  POST: ReturnValve == isWorking;     }

 INVARIANTS
   true;
```

Figure 2.6. Specification of electric kettle

For instance, Figure 2.6 represents a specification of an electric kettle. This specification says that the object `ElectricKettle` has four attributes and six methods. Let us take `openLid()` method in Figure 2.6. To execute `openLid()`, the lid must be closed and the power must be on. After executing the method, the lid will be opened. The method `setMode()` takes a formal parameter `md`, intended that the working mode will be updated to the given `md` in the post-condition.

Our specification language also can specify *invariants*. The invariant is a condition that must be satisfied all the time no matter which method is executed. In Figure 2.6, no specific invariant is given.

4.2 Service

The specification for HNS service can be specified in almost the same way.

Figure 2.7 shows a general structure of service specification. In addition to the case of individual appliance, a service has a set of appliances used by the service.

For instance, Figure 2.8 shows a specification of *Cooking Preparation Service*, which warms up the kitchen appliances. The specification says that this service uses four appliances — a ventilator, a gas system, a light, and an electric kettle. The attribute `attr` of each appliance `app` is denoted by `app.attr`. As specified in the post condition of `activation()`, when the service is activated, the light is turned on, gas valve is opened, ventilator is turned on, and kettle is turned to the boiling mode. We assume that for every appliance (or every service) in the HNS, a specification is given.

The detailed specification for other appliances are described in the Appendix.

```
SERVICE Service_name EXTENDS Service

    APPLIANCES
       APPLIANCE appliance_name1;
       APPLIANCE appliance_name2;
       ...

    ATTRIBUTES
       TYPE attribute_name1 : {value1 ... value2} = Init_val1;
       TYPE attribute_name2 : {value1 ... value2} = Init_val2;
       ...

    METHODS
       method_name1() {
           PRE:   Pre-Condition;
           POST:  Post-Condition;
       }
       ...

    INVARIANTS
       Invariant Condition;
```

Figure 2.7. General specification of service

```
SERVICE CookingPreparationService EXTENDS Service
  APPLIANCES
     vent      : Ventilator;
     gas       : GasSystem;
     light     : Light;
     kettle    : ElectricKettle;
  ATTRIBUTES
     brightness         : {SOFT, MID, STRONG} = SOFT;
     windLevel          : {0, 1, 2, 3} = 1;
     fireLevel          : {ZERO, SOFT, MID, STRONG} = ZERO;
     mode               : {IDLE, BOILING, WARMING} = IDEL;
  METHODS
     activation() {             //activate the cooking preparation
        PRE:   service.isWorking==false;
        POST:  light.power==ON && light.brightness==brightness && light.isWorking==true &&
               gas.power==ON && gas.valve==OPEN && gas.fireLevel==fireLevel && gas.isWorking==true &&
               vent.power==ON && vent.windLevel==windLevel && vent.isWorking==true &&
               kettle.power==ON && kettle.mode==workingMode && kettle.isWorking==true &&
               service.isWorking==true;
     }

     stop() {                   //shutdown the preparation
        PRE:   service.isWorking==true;
        POST:  light.isWorking==false && light.power==OFF &&
               gas.valve==CLOSE &&  gas.isWorking==false &&  gas.power==OFF &&
               vent.isWorking==false && vent.power==OFF &&
               kettle.isWorking==false && kettle.power==OFF && service.isWorking==false;
     }

     setLightBrightness(lb){  PRE:   service.isWorking==false;         POST:  brightness==lb;  }

     setWindLevel(wl)      {  PRE:   service.isWorking==fasle;         POST:  windLevel==wl;   }

     setFireLevel(fl)      {  PRE:   service.isWorking==false;         POST:  fireLevel==fl;   }

     setKettleMode(wm)     {  PRE:   service.isWorking==false;         POST:  mode==wm;        }
  INVARIANTS
     true;
```

Figure 2.8. Specification of cooking preparation service

5. Summary

In this chapter, in order to understand the HNS well, we proposed an object-oriented model an specification language for the HNS. The model mainly consists of three kinds of objects (classes): Appliance, Service, and Home. These objects form the following relationships: [R1: a Home has multiple Appliances], [R2: a Home has multiple Services], and [R3: a Service uses multiple Appliances]. These relationships match well the intuition of the HNS and integrated services. For each object, it has the internal state (power, isWorking, etc.) and the operational interfaces (i.e., APIs). Hence, it is reasonable to model each object with consisting of attributes and methods.

In order to conduct the safety requirement engineering for HNS (see Chapter 4) and capture the given HNS model more clearly. we have introduced a language for describing the *design specification* of the HNS. In this language, we defined a pair of `PRE` and `POST` conditions for each method of the object to specify the function requirement, `INVARIANTS` conditions to describe the invariant specification for each object of the HNS model. Note, that the specification defined in this chapter does not consider the safety requirements because the safety requirements have been not been dealt yet.

By using the object-oriented model and the proposed specification language, we believe it would be enough to model the HNS and specify function requirement clearly.

6. Related Work

For the related work of [32], they have developed an object-oriented model for networked appliances. In addition to the appliance object, here we newly introduce a service object for the integrated service and a home object for the home environment.

For the related work of [19], they proposed a language for representing integrated services. The proposed language consists of two parts: (a) system description for the HNS and (b) service description for integrated service. In this language, the `PRE` and `POST` conditions are only defined in the Appliance object. Moreover, there is not `INVARIANTS` condition definition for any object. It does not fit our HNS object-oriented model very much and it is very difficult for us to describe the safety requirement specification for each object of HNS by using our proposed method in Chapter 4.

Because of this, we modified and simplified specification language by giving `PRE`, `POST` and `INVARIANTS` conditions for each object and made it suitable to conduct the function and the safety requirements of HNS.

Chapter 3

Characterizing Safety of HNS Integrated Services

1. Introduction of HNS Safety Issues

In Chapter 2, we have introduced the HNS by using object-oriented model and specification language. The great advantage of HNS integrated service lies in the flexible integration of different home appliances through the network. In developing and providing a HNS integrated service, 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 high quality of life in smart home.

In general, for the conventional (non-networked) home appliances, the safety has been assured manually by the human user. That is, every user is supposed to follow the safety instructions typically described in the user's manual. However, for the HNS integrated services, we have to consider the safety issue much more carefully than before because the following reasons.

- **Difference in operators**

  The first reason is that the operators of the appliances and the service are different. For the conventional home appliances, the operators are the human users. But for HNS, the service is typically implemented as a software application, the networked appliances are operated automatically by software agents instead of human user, so we can not ensure that the system (software) can obey the safety instructions.

- **Integration of appliances and services**

  The second reason is that the integrated service is realized on the flexible integration of multiple appliances or services. In general, the user can only operate a single appliance simultaneously. But for one integrated service, it usually operates multiple appliances at a time, which yields global dependencies among different appliances. Moreover, because multiple integrated services can be executed simultaneously, the unexpected functional conflicts may occur among the services. Thus, a single fault in the service application can also cause serious accidents to the user.

- **Responsibilities among stakeholders**

  The third reason is that it is necessary to make it clear who is responsible for guaranteeing the safety of HNS. As we know, an HNS consists of many heterogeneous appliances, and the service orchestrates different multiple appliances simultaneously. For every appliance and service of HNS, the manufacturer of the appliance and the developer of service should prescribe some safety instructions for proper and safe use of them under the provided environment. So we can consider that the safety instructions should be defined over the heterogeneous objects by multiple stakeholders which includes the appliance manufacturers, the service developers and the environment providers. For this, it is necessary to clarify who is responsible for the safety instruction if the system can not satisfy the safety requirements (an accident has happened).

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. For this, in this chapter, we will propose a way to formalize the safety of the HNS integrated services and formulate the safety validation problem by considering the nature of the HNS and integrated services.

2. Formalizing the Safety of HNS

First, we will see what *safety* is. In a broad sense, the safety of an integrated services can 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 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 [8, 11, 12]. Considering the nature of the HNS, we propose to classify the safety properties into the following three types.

2.1 Local Safety Property

For every electric appliance, the manufacturer of the appliance prescribes a set of *safety instructions* for proper and safe use of the appliance. Conventionally, these instructions have been designated for human users. However, in the HNS integrated service, the instructions must be guaranteed within the software. For instance, the following shows a safety instruction for 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 of Chapter 2) must be implemented so that the service never opens the lid while the kettle is in the boiling mode. Other safety instructions include the installation issues. That is, every appliance must be installed in a proper environment described by its *specification*, including power voltage, rated current, power consumption, allowable temperature and humidity, etc.

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 $LocalProp(d) = \{lp_1, lp_2, ..., lp_m\}$ be the set of all local safety properties with respect to the appliance d. For a given integrated service s,

let $App(s) = \{d_1, d_2, ..., d_n\}$ 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.

2.2 Global Safety Property

Since an integrated service orchestrates different multiple appliances simultaneously, it is necessary to consider *global properties* over the multiple appliances. For instance, SS4 (bath preparation service) in Section 2 of Chapter 2 should guarantee the following safety property to prevent the user from getting scald or heart attack.

> G1: When the service opens the bath valve, the water temperature of the hot water system must be between 35 and 45 degree.

The next example shows a safety property for SS3 (cooking preparation service), which avoids carbon monoxide poisoning.

> G2: While the gas valve is opened, the ventilator must be turned on.

Note that each of the properties is *globally* specified over multiple appliances. These *global safety properties* are usually service-specific, and are not covered by the local safety properties of individual appliances. Therefore, we suppose that the global safety properties are carefully *specified by the provider of the integrated service.* 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 $d_1, d_2, ..., d_n$ that are used by an integrated service s. $GlobalProp(s) = \{gp_1, gp_2, ..., gp_k\}$ denotes the set of all global safety properties for s.

2.3 Environment Safety Property

In general, each house has a set of residential rules for inhabitants and neighbors to make a safe living. Since the integrated services give various impacts against the surrounding *environment* (including the room, the building, the neighbors, etc.), the services must be safe against the environment by conforming to the residential rules. For instance, most house has a capacity of electricity, which yields the following safety property.

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

Also for emergency, the following safety property should be concerned.

> E2: Do not lock doors and windows in case of fire.

The following property may be derived from community rules.

> E3: Do not make loud noise or sound after 9 p.m.

We assume that these safety properties are derived from the residential rules, including the house manual, the emergency procedure, community rules and policies, etc. We call such properties *environment safety properties.*

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. $EnvProp = \{ep_1, ep_2, ..., ep_l\}$ denotes the set of all environment properties.

[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.

2.4 Safety Definition 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 $s \vdash \mathit{LocalProp}(s)$.
- s is *globally safe* iff $s \vdash \mathit{GlobalProp}(s)$.
- s is *environmentally safe* iff $s \vdash \mathit{EnvProp}$.
- s is *safe* iff s is locally, globally and environmentally safe.

3. Safety Validation Problem

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

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

[**Input:**] A HNS h, an integrated service s, $\mathit{LocalProp}(s)$, $\mathit{GlobalProp}(s)$, and $\mathit{EnvProp}()$.

[**Output:**] A verdict whether s is safe or not within h.

We assume that the given h and s are implemented based on the object-oriented model presented in Section 3 of Chapter 2.

4. Summary

In order to understand why the safety issues of HNS have to be considered more carefully than before, we compared the differences of HNS with general appliance operation in the beginning of this chapter. Based on the discussion about these differences, we can understand only a single fault in the service application can cause serious accidents to the user. So considering these safety issues carefully in the software implementation is a very important point for guaranteeing the safety of HNS.

In order to consider the safety issues systematically, we also proposed a systematic approach to formalize the concept of safety in the context of HNS. First, we gave a definition of "safety in broad sense". After this, we also formalized the safety properties as $LocalSafetyProperty$, $GlobalSafetyProperty$ and $EnvironmentSafetyProperty$ by considering the nature of the HNS. At last, we formulated the safety validation problem based on the three kinds of safety definition of HNS integrated service.

For this systematic approach of HNS safety issue, we believe that it can not only provide a strong rationale to consider the safety issues, but also can assist the HNS service vendors to guarantee the service is safe for inhabitants, house properties and their surrounding environment for developing and providing a HNS integrated service,

5. Related Work

Traditionally, the safety issues have been addressed in *safety critical systems* [21–25], such as, aerospace systems and nuclear plants. Despite their importance, there are yet little research work in the ubiquitous computing area, including smart home. Compared to the ubiquitous applications, the safety critical systems are quite monolithic, where requirements and system configurations are not frequently changed. Thus, we needed alternative analysis models suitable for the object-oriented model. On the other hand, 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 flexibly for different purposes. So, we consider that our notions of global safety and environment safety are specific to ubiquitous applications.

We believe that the original idea of characterizing safety in the HNS has substantial values for developing and analyzing safe integrated services systematically.

Chapter 4

Deriving Safety Properties of the HNS integrated service

1. Introduction

In Chapter 3, we have characterized the safety of the HNS by three types of properties: (1) *local safety properties* are safety instructions of each individual appliance, (2) *global safety properties* are specified over multiple appliances to operate the HNS service safely, and (3) *environment safety properties* are residential rules in home and surrounding environments, independent of appliances and services.

Then a question arises: "How can we give the concrete safety properties for the given HNS (model)?" Of course, one can specify any safety properties in an ad-hoc and intuitive manner. However, how can we say that the properties are good or bad?

Here we should note that it is important to have *correct* and *complete* safety properties. The quality of the safety properties has a significant impact on the subsequent safety validation process. If a safety property is not correctly specified, then the validation process yields a wrong verdict. If a critical safety property is overlooked, then the validation process ignores serious accidents in the HNS. If a safety property is quite irrelevant to the given HNS model, we should not put it into the expensive validation process.

Unfortunately however, it is very difficult and challenging to have such a *good* set of safety properties in general.

This chapter presents a requirement-engineering approach that can systematically derive the verifiable safety properties for the HNS on the offline stage. Specifically, we propose a new hazard analysis model, called *HNS-HAM (HNS Hazard Analysis Model)*, which investigates potential *hazards* within the given HNS model. The safety properties are then derived so that the properties define conditions that prevents the hazards from occurring.

The proposed *HNS-HAM* consists of four levels of abstractions: (1) *hazard context* level, (2) *hazardous state* level, (3) *object attribute* level, and (4) *object method* level, as shown in Figure 4.1. For a given HNS model and the hazard context, we construct the *HNS-HAM* by a goal-oriented analysis [38], which specifies logical relations between the adjacent abstraction levels.

Intuitively, the hazard context is a type of hazard to be focused in the hazard analysis, which is independent of the specific HNS model. The hazard context is decomposed into several hazardous states, where the context is realized. Then, the hazardous states are associated by attributes and methods of appliance objects. The construction of *HNS-HAM* yields cause-and-effect chains from the abstract hazard contexts to the concrete attributes and operations of HNS objects, specified in the given HNS specification.

Finally, the safety properties and their responsible operations are derived from the complete model, which give the strong rationale of the safety of the HNS. We also discuss a technique using *hazard template* to enhance the reusability of the *HNS-HAM*.

2. Goal and Approach

The main problem in this chapter is considering how to derive safety properties for given HNS specifications (in Figures 2.6 and 2.8), systematically. Moreover, the derived safety properties should be reflected in the (original) specifications, so that the safety properties are explicitly considered at the design level.

Among the local, the global and the environment safety properties, we do not consider the environment property here. By definition, every environment property heavily depends on the environmental factors, which cannot be captured by the HNS model and specification. After all, the problem is formulated as follows:

Input:

- (I) *ASpec*: a set of appliance specifications, and
- (II) *SSpec*: a set of service specifications.

Output:

- (a) *LProp*: a set of local safety properties,
- (b) *Safe-ASpec*: a set of (safe) appliance specifications, where *Safe-ASpec* is a revision of *ASpec* with considering *LProp*
- (c) *GProp*: a set of global safety properties,
- (d) *Safe-SSpec*: a set of (safe) service specifications, where *Safe-SSpec* is a revision of *SSpec* with considering *GProp*.

To achieve the goal, we conduct a *hazard analysis*, which investigates potentially dangerous situations under the given HNS. To perform the analysis efficiently, we propose a *HNS hazard analysis model* (HNS-HAM, for short). Using the *HNS-HAM*, we then derive the safety properties in a goal-oriented way.

3. Hazard Analysis Model for HNS

3.1 Overview of Hazard Analysis Model

We propose a unique hazard analysis model, called *HNS-HAM*, consisting of four levels of abstractions. Figure 4.1 depicts the overview of the proposed model. The model starts with abstract types of hazards (we call *hazard contexts*), which are independent of specific HNS configuration. The hazard contexts (Level 1) are refined to *hazardous states* (Level 2), and then mapped to concrete *attributes* (Level 3) and *methods* (Level 4) of the HNS objects. The adjacent levels are linked by *logic relations*. The *HNS-HAM* specifies *cause-and-effect chains* [23] from the abstract hazard contexts to the concrete attributes and operations of the HNS model. We explain the details of each level as follows.

(A) *Level1: Hazard Context Level*

The top level of the *HNS-HAM* defines abstract types of hazards to be considered in the hazard analysis, which we call *hazard contexts*. Each hazard context must be independent of the specific HNS instances. Typical hazard contexts for the HNS include burn, scald, explosion, gas poisoning, flood, deficiency of oxygen, noise, etc. The purpose of Level 1 is to determine the scope of the hazard analysis.

(B) *Level2: Hazardous State Level*

For each specific hazard context *hc* defined in Level 1, this level defines possible states in the given HNS where the hazard *hc* is realized. We call such dangerous states *hazardous states*. In general, a hazard occurs due to several related

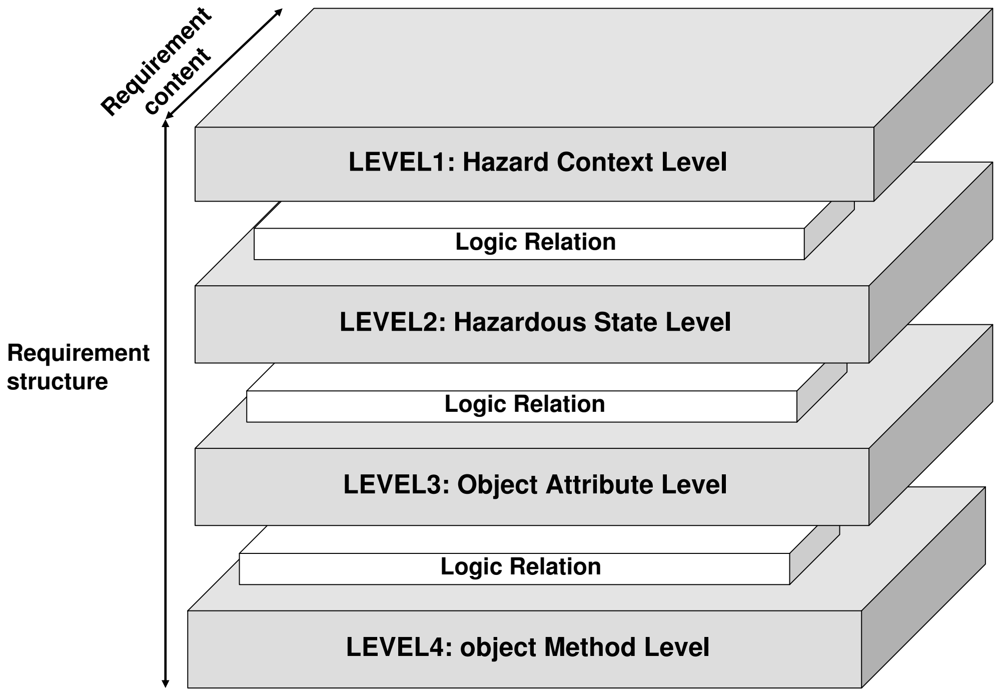

Requirement content
Requirement structure
LEVEL1: Hazard Context Level
Logic Relation
LEVEL2: Hazardous State Level
Logic Relation
LEVEL3: Object Attribute Level
Logic Relation
LEVEL4: object Method Level

Figure 4.1. Structure of HNS-HAM

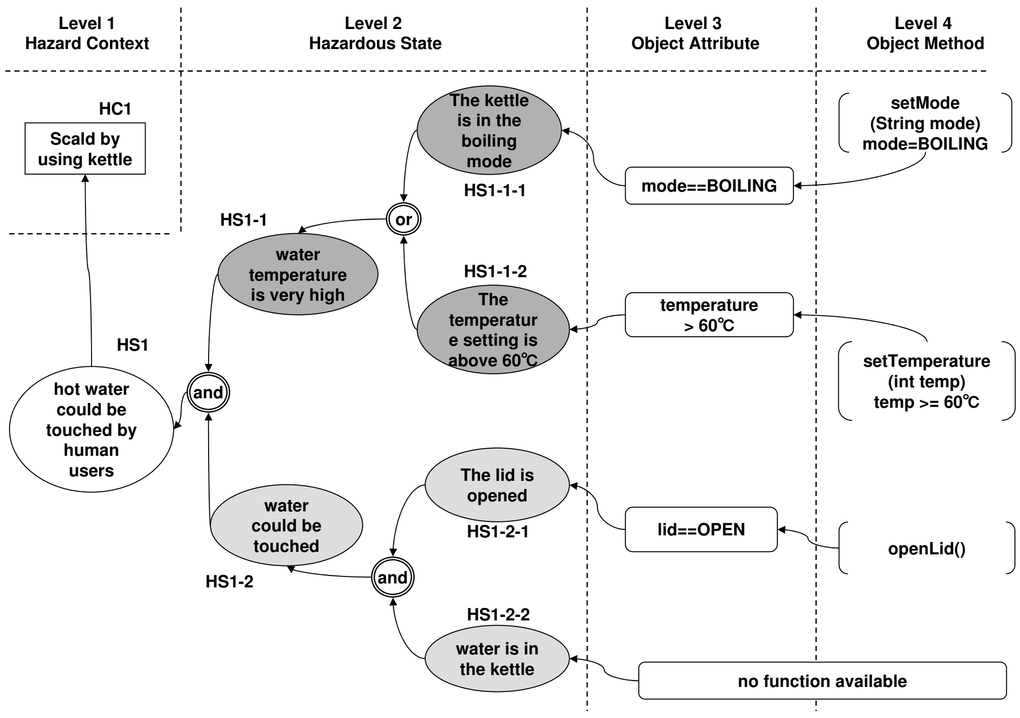


Figure 4.2. HNS-HAM for electric kettle (scald)

factors. Also a hazardous state can be composed of fine-grained sub-states. So we characterize a hazard context hc by several hazardous states hs_1, hs_2, ..., hs_n connected by logical operators (AND, OR, NOT). Moreover, a hazardous state hs_i can be decomposed into several sub-states $hs_{i1}, ..., hs_{ik}$. For a hazardous state hs, if there is no more sub-state into which hs is refined, we call hs an *atomic hazardous state*.

Figure 4.2 shows an example of *HNS-HAM* investigating the hazard context "scald" within `ElectricKettle` (let it be HC1). In the Figure 4.2 a rectangle node represents a hazard context, an oval node represents a hazardous state (other nodes will be explained later). Each arrow from node A to B denotes a causal relationship that "B is caused by A". In this example, the possible cause of the "(HC1): scald by using kettle" is characterized by the hazard state "(HS1): hot water could be touched by human users". Moreover, the hazard state "(HS1)" is characterized the AND composition of two states, "(HS1-1): water temperature is very high", AND "(HS1-2): water could be touched". In this example, "HS1-1" and "HS1-2" can be further decomposed into sub-states. For "HS1-1", it is characterized the OR composition of "(HS1-1-1): the kettle is in the boiling mode", OR "(HS1-1-2): the temperature setting is above 60". For "HS1-2", it is characterized the AND composition of "(HS1-2-1): the lid is opened", AND "(HS1-2-2): water is in the kettle". HS1-1-1, HS1-1-2, HS1-2-1 and HS1-2-2 are atomic hazardous states.

The purpose of Level 2 is to map the hazard context into concrete causes in the HNS currently focused on. In this level, we can see a condition under which the hazard context is realized (in the natural language).

(C) *Level3: Object Attribute Level*

This level encodes every atomic hazardous state defined in Level 2, in a formal condition over attributes of a HNS object. Since each hazardous state is somehow conceptual representation, this level transforms the state into rigorous expression in the HNS specification.

In Figure 4.2, a round-box node represents a condition over attributes of `ElectricKettle`. For instance, the atomic hazardous state HS1-1-1 is encoded by an expression mode==BOILING, HS1-1-2 is encoded by an expression temperature $>$ 60. and HS1-2-1 is encoded by an express lid==OPEN. Based on the function specification of our `electrickettle`, we note there is not corresponding attribute or function to manage the atomic hazardous state HS1-2-2. So we can not translate it to the level3. For this case, it means that the design of this system can not capture this hazard state automatically. This hazard state should be captured by human users.

In this level, the hazard context can be captured in terms of as concrete attribute values of the HNS objects.

(D) *Level4: Object Method Level*

Finally, this level identifies object methods that can trigger the hazard context. More specifically, for each attribute condition *cond* in Level 3, we find methods m_1, m_2, ..., m_r that can make *cond* true. These methods can be easily identified by investigating post-condition of the methods defined in given HNS specifications. The purpose of this level is to clarify operations that must be anticipated for the safety assurance.

In Figure 4.2, a node with brackets represents a method of `ElectricKettle` that makes a certain attribute condition true. For instance, we can see that executing setMode(BOILING) causes a condition mode==BOILING, executing setTemperature(temp > 60) causes a condition temperature > 60, executing openLid() causes a condition lid==OPEN, as specified in the specification in Figure 2.6, and that these execution would be one factor causing scald.

3.2 Constructing HNS-HAM

The *HNS-HAM* is constructed by the following procedure. Note in the following that a *HNS-HAM* is constructed for every pair of a hazard context *hc* and a given specification *spec*.

Step 1 (Definition of Hazard Contexts): Enumerate any hazard contexts that might occur in the given HNS. Since this step requires no technical aspect of the HNS specifications, any stakeholders can join the analysis.

Step 2 (Elaboration of Hazard States): Pick up specification *spec* from *ASpec* (or *SSpec*). For each hazard context *hc*, characterize *hc* by some hazardous states within the object of *spec*. Then, decompose each hazardous state into sub-states in a goal-oriented fashion until all atomic states are obtained, which completes Level 2. Step 2 is the most important step that determines the quality of the *HNS-HAM*. Participation of experts in safety engineering would be encouraged to improve the completeness.

Step 3 (Mapping into Attributes Conditions): Encode each atomic hazardous state *hs* by a condition over object attributes based on *spec*. This constructs Level 3. If there is no attribute corresponding to *hs*, check if *hs* can be further decomposed. If *hs* is really atomic, then revise *spec*.

Step 4 (Obtaining Methods): For each attribute condition *cond*, find methods in *spec* that make *cond* true, by consulting the post-conditions of the methods.

3.3 Deriving Safety Properties with HNS-HAM

Suppose that a *HNS-HAM* $ham(o, hc)$ is constructed with respect to a HNS object o (defined by *spec*) and a hazard context hc. Now we derive the safety properties that must be conformed by o to prevent hc from occurring. For this, we use Levels 1 and 2 of $ham(o, hc)$, extensively.

According to Levels 1 and 2 of $ham(o, hc)$, hc is characterized by a logical formula f_{hc} consisting of atomic hazardous states. If f_{hc} holds, then the hazard hc is realized. Conversely, to prevent hc from occurring, we have to assure $\neg f_{hc}$ for all the time. Thus, we want to derive the safety properties as a set of rules $R = \{r_1, r_2, ..., r_n\}$, interpreted as a conjunction $\neg f_{hc} = r_1 \wedge r_2 \wedge ... \wedge r_n$. Using the *clausal normal form* [39] in the classical logic programming, we can obtain such a set $R = \{r_1, ..., r_n\}$ that $r_i = (P_1 \wedge ... \wedge P_m) \rightarrow (Q_1 \vee ... \vee Q_n)$, where P_x and Q_y are literals.

Based on the idea, we derive the safety properties from a given *HNS-HAM* as follows.

Input: a *HNS-HAM* $ham(o, hc)$ constructed for a HNS object o and a hazard context hc.

Step 1: From Levels 1 and 2 of $ham(o, hc)$, derive a logical formula $f_{hc} = f(hs_1, ..., hs_l)$ characterizing hc by atomic hazardous states hs_i $(1 \leq i \leq l)$.

Step 2: Calculate $\neg f_{hc}$.

Step 3: Convert $\neg f_{hc}$ into the clausal normal form $R = \{r_1, ..., r_n\}$.

Step 4: Define each r_i as a safety property.

Let us derive safety properties for `ElectricKettle` using the *HNS-HAM* in Figure 4.2. According to Level 1 and 2, we get

$$f_{HC1} = (\text{HS1-1-1} \vee \text{HS1-1-2}) \wedge \text{HS1-2-1}$$

Making a negation, and applying the De Morgan's and distribution lows, we obtain

$$\begin{aligned}\neg f_{HC1} &= \neg \text{HS1-1-1} \wedge \neg \text{ HS1-1-2} \vee \neg \text{ HS1-2-1} \\ &= (\text{HS1-1-1} \rightarrow \neg \text{ HS1-2-1 }) \wedge (\text{HS1-1-2} \rightarrow \neg \text{ HS1-2-1}) \wedge \\ &\quad (\text{HS1-2-1} \rightarrow \neg \text{ HS1-1-1}) \wedge (\text{HS1-2-1} \rightarrow \neg \text{ HS1-1-2})\end{aligned}$$

Thus, we derived the following four safety properties for `ElectricKettle` to prevent the scald from occurring.

(P1) HS1-1-1 $\rightarrow \neg$ HS1-2-1: When the kettle is in the boiling mode, the lid must not be opened.
(P2) HS1-1-2 $\rightarrow \neg$ HS1-2-1: When the setting temperature is higher than 60, the lid must not be opened.
(P3) HS1-2-1 $\rightarrow \neg$ HS1-1-1: When the lid is opened, the kettle must not be in the boiling mode.
(P4) HS1-2-1 $\rightarrow \neg$ HS1-1-2: When the lid is opened, the temperature setting must be below 60.

All of the above properties are quite reasonable as the safety instructions of an electric kettle. Note that the properties P1-P4 are all *local safety properties*, since they are closed within a single HNS appliance (i.e., Electric Kettle).

3.4 Updating HNS Specifications with Derived Safety Properties

Based on the safety properties derived, we update the original specification so that the safety properties are reflected. To achieve this, we use Levels 3 and 4 of *HNS-HAM* extensively. Each safety property is a condition over atomic hazardous states, and Level 3 specifies the correspondence between the atomic states and the object attributes of the model. So each safety property can be encoded by a condition using attributes.

An encoded safety property can be specified as an *invariant* in the specification, intended that the property must hold all the time for the safety. Or, if the safety property is encoded by attributes in the same appliance, we can specify the property as pre/post-conditions of methods designated by Level 4.

Because "how to update the specification" has a close relation with the proposal for validating safety properties with design by contract. We will give the details by conducting some examples in the Section 3 of Chapter 5.

3.5 Procedure of Proposed Method

Finally, we sum up the proposed method against the problem formulated in Section 2. If the proposed method is applied to a HNS appliance, the local safety properties are derived (as seen in the kettle example). If applied to a HNS service, the global safety property can be obtained.

(A) Safety Analysis for HNS Appliance: For each appliance *app* specified in $spec \in ASpec$,

1. Define hazard contexts hc_1, hc_2 ... hc_x.
2. For each hazard context hc_i, construct a *HNS-HAM* $ham(app, hc_i)$.
3. Derive safety properties p_1, p_2, ..., p_n from $ham(app,\ hc_i)$. Put $p_1, ..., p_n$ in $LProp$.
4. For all p_j, update *spec*. Put the resultant $spec'$ in $Safe\text{-}ASpec$.

(B) Safety Analysis for HNS Service: For each service *ser* specified in $spec \in SSpec$,

1. Define hazard contexts hc_1, hc_2 ... hc_x.
2. For each hazard context hc_i, construct a *HNS-HAM* $ham(ser, hc_i)$.
3. Derive safety properties p_1, p_2, ..., p_n from $ham(ser, hc_i)$. Put p_1 ,..., p_n in $GProp$.
4. For all p_j, update *spec*. Put the resultant *spec′* in $Safe$-*SSpec*.

For each service *ser* specified in $spec \in SSpec$,

1. Define hazard contexts hc_1, hc_2 ... hc_x.
2. For each hazard context hc_i, construct a HNS-HAM $ham(ser, hc_i)$.
3. Derive safety properties p_1, p_2, ..., p_n from $ham(ser, hc_i)$. Put p_1 ,..., p_n in $GProp$.
4. For all p_j, update *spec*. Put the resultant *spec′* in $Safe$-*SSpec*.

4. Case study

As a case study, this section demonstrates analysis of `CookingPreparationService` (introduced in Chapter 2 and specified in Figure 2.8).

Figure 4.3 shows a *HNS-HAM* for this case study. As seen in the model, the HC1 is caused by the hazard state "(HS1): the gas density could be higher than the dangerous threshold". For the HS1, it can be further decomposed into two sub-states: "(HS1-1): gas is in the room" AND "(HS1-2): air is not ventilated". The possible causes of HS1-1 and HS1-2 are the "(HS1-1-1): gas system is being used" and "(HS1-2-1): the ventilator does not work". Based on the specification of this service,, the possible causes of HS1-1-1 was characterized by two atomic sub-state: "(HS1-1-1-1): gas valve is opened" AND "(HS1-1-1-2): fire is on". The

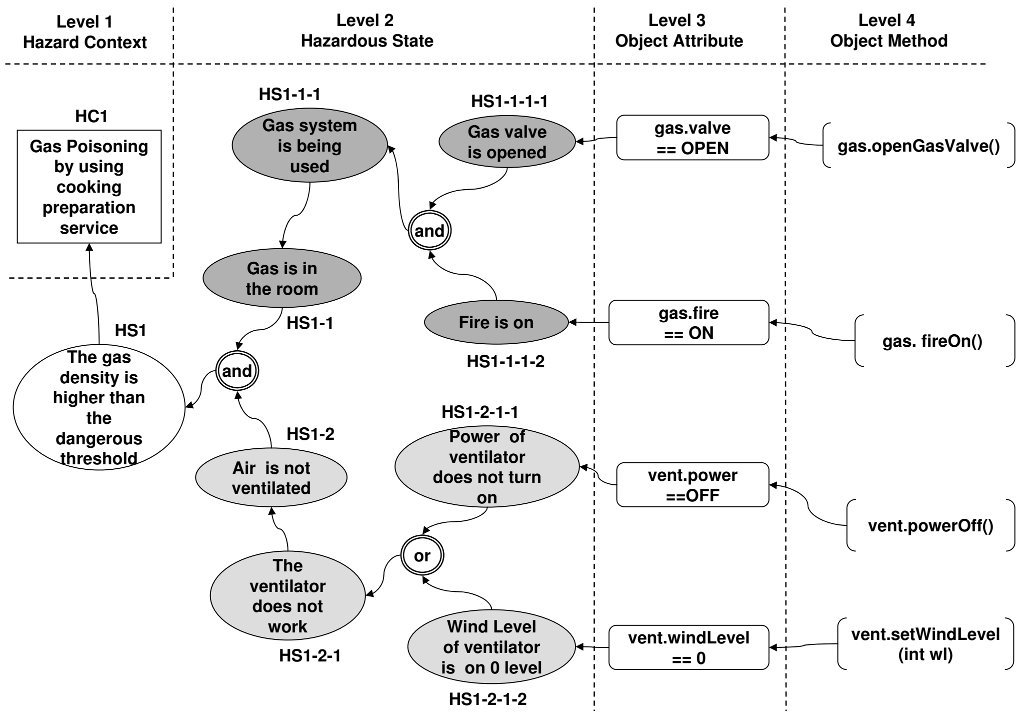


Figure 4.3. HNS-HAM for cooking preparation service (gas poisoning)

possible causes of HS1-2-1 was characterized by two atomic sub-state: "(HS1-2-1-1): power of ventilator does not turn on" OR "(HS1-2-1-2): wind level of ventilator is on 0 level". Since here, each of these states can not be further decomposed into other sub-states. From the *HNS-HAM*, we get

$$f_{HC1} = (\text{HS1-1-1-1} \wedge \text{HS1-1-1-2}) \wedge (\text{HS1-2-1-1} \vee \text{HS1-2-1-2})$$

Then, compute the negation

$$\begin{aligned}\neg f_{HC1} &= \neg\ (\text{HS1-1-1-1} \wedge \text{HS1-1-1-2}) \vee (\neg\ \text{HS1-2-1-1} \wedge\neg\ \text{HS22}) \\ &= (\text{HS1-1-1-1} \wedge \text{HS1-1-1-2} \rightarrow \neg\ \text{HS1-2-1-1}) \wedge \\ &\quad (\text{HS1-1-1-1} \wedge \text{HS1-1-1-2} \rightarrow \neg\ \text{HS1-2-1-2})\end{aligned}$$

From this, we obtain the following four safety logical formulas:

(GP1) HS1-1-1-1 $\wedge$ HS1-1-1-2 $\rightarrow \neg$ HS1-2-1-1: When the gas valve is opened and the fire is on, the ventilator must be turned on.

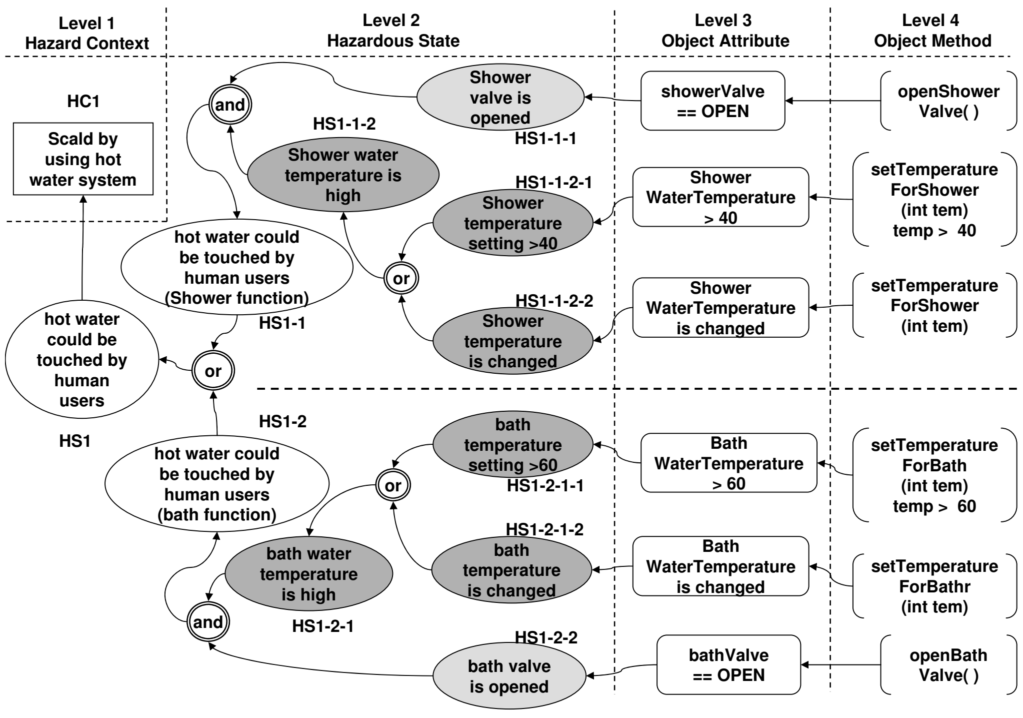
(GP2) HS1-1-1-1 $\wedge$ HS1-1-1-2 $\rightarrow \neg$ HS1-2-1-2: When the gas valve is opened and the fire is on, the wind level of the ventilator must not be 0.

(GP3) HS1-2-1-1 $\rightarrow \neg$ HS11 $\vee\neg$ HS1-1-1-2: When the ventilator power is in off, the gas valve must not be opened or fire must not be on

(GP4) HS1-2-1-2 $\rightarrow \neg$ HS11 $\vee\neg$ HS1-1-1-2: When the ventilator wind level is 0, the gas valve must not be opened or fire must not be on.

Note that these properties are global safety properties, since these are specified over different appliances (i.e., the gas valve and the ventilator). Using Level3 and Level4 of the *HNS-HAM*, GP1, GP2, GP3 and GP4 are translated as some kinds of the conditions in the specification. The details about the method for updating HNS specifications with safety properties will be described in the next Chapter 5

Based on the same way, we build another *HNS-HAM* for `hotWaterSystem` (Figure 4.4) We also have derived some safety properties based on the *HNS-HAM* as below:



Level 1
Hazard Context
Level 2
Hazardous State
Level 3
Object Attribute
Level 4
Object Method
HC1
Scald by using hot water system
hot water could be touched by human users
HS1
or
hot water could be touched by human users (Shower function)
HS1-1
and
Shower valve is opened
HS1-1-1
Shower water temperature is high
HS1-1-2
or
Shower temperature setting >40
HS1-1-2-1
Shower temperature is changed
HS1-1-2-2
hot water could be touched by human users (bath function)
HS1-2
and
bath water temperature is high
HS1-2-1
or
bath temperature setting >60
HS1-2-1-1
bath temperature is changed
HS1-2-1-2
bath valve is opened
HS1-2-2
showerValve == OPEN
Shower WaterTemperature > 40
Shower WaterTemperature is changed
Bath WaterTemperature > 60
Bath WaterTemperature is changed
bathValve == OPEN
openShower Valve( )
setTemperature ForShower (int tem) temp > 40
setTemperature ForShower (int tem)
setTemperature ForBath (int tem) temp > 60
setTemperature ForBathr (int tem)
openBath Valve( )

Figure 4.4. HNS-HAM for hot water system (scald)

(P5) HS1-1-1 $\rightarrow \neg$ HS1-1-2-1: While the shower valve is opened, the shower water temperature setting must not be above 40.

(P6) HS1-1-1 $\rightarrow \neg$ HS1-1-2-2: While the shower valve is opened, the shower water temperature must be not changed.

(P7) HS1-1-2-1 $\rightarrow \neg$ HS1-1-1: While the shower water temperature setting is above 40, the shower water valve must not be opened.

(P8) HS1-1-2-2 $\rightarrow \neg$ HS1-1-1: While the shower water temperature is changed, the shower water valve must not be opened.

(P9) HS1-2-2 $\rightarrow \neg$ HS1-2-1-1: While the bath valve is opened, the bath water temperature setting must not be above 60.

(P10) HS1-2-2 $\rightarrow \neg$ HS1-2-1-2: While the bath valve is opened, the bath water temperature must be not changed.

(P11) HS1-2-1-1 $\rightarrow \neg$ HS1-2-2: While the bath water temperature setting is above 60, the bath water valve must not be opened.

(P12) HS1-2-1-2 $\rightarrow \neg$ HS1-2-2: While the bath water temperature is changed, the bath water valve must not be opened.

5. Improving Reusability of HNS-HAM with Hazard Template

The proposed method with the *HNS-HAM* can derive safety properties for a given HNS specification, systematically. The drawback is that the construction of the *HNS-HAM* poses not a little cost even for a single instance of the HNS. If appliances and services in the HNS are added or changed, the expensive analysis has to be performed again for the new configuration, basically. To make the proposed method more practical, it is necessary to save the analysis cost.

To cope with the problem, we try to *reuse* the existing hazard analysis model for various instances of the HNS.

The most expensive (but essential) part of the proposed method is in the construction of Level 2 (i.e., the elaboration of hazardous states, see Step 2 of Section 3.2). In the *HNS-HAM*, Level 1 deals with the quite general context, whereas Level 3 is quite specific to the given HNS instance. Level 2 plays a role of *bridge* between general and specific notions, which complicates the issue.

Our key idea is to find, within the Level 2, *generic* hazardous states that are independent of specific HNS configuration, and then to extract the generic portion as *hazard template* to be reused.

Let us recall the example of the scald in Figure 4.2. If we consider the causes of the scald from the most general viewpoint, the following condition appears.

(GHS1-1:) The hot water can be touched by a human user.

GHS1-1 is further decomposed into the following two conditions.

(GHS1-1-1:) The temperature of the water is very high.

(GHS1-1-2:) The water is accessible by the user.

These conditions can be represented as a *HNS-HAM* shown in Figure 4.5. Here we should note that the model contains *general hazardous states* independent of any specific appliances or services. Therefore, we use the model as a *hazard template* for analyzing the scald.

For a given HNS specification, the hazard template is refined into concrete hazardous states according to the specification. Since the template can be reused

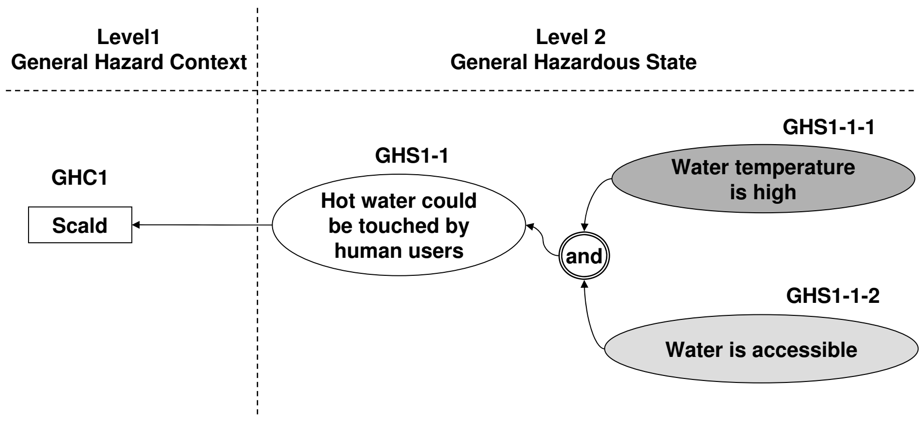


Figure 4.5. Hazard template for scald

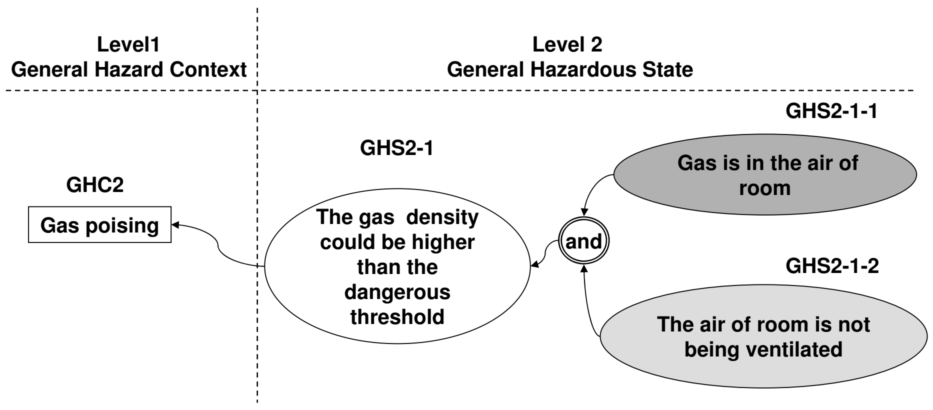

Level1
General Hazard Context
Level 2
General Hazardous State
GHC2
Gas poising
GHS2-1
The gas density could be higher than the dangerous threshold
and
GHS2-1-1
Gas is in the air of room
GHS2-1-2
The air of room is not being ventilated

Figure 4.6. Hazard template for gas poising

for any HNS appliances or services, the cost for constructing Level 2 is significantly reduced.

For instance, the template in Figure 4.5 can be used for constructing the *HNS-HAM* in Figure 4.2, whose context is “scald by ElectricKettle”. In the construction, GHS1-1-1 is realized by concrete states HS1-1 connected by HS1-1-1 or HS1-1-2, in accordance with the specific context of ElectricKettle. Similarly, GHS1-1-2 is realized by HS1-2 connected by HS1-2-1 and HS1-2-2. Thus, we can see that the structure of the hazard template is *inherited* in the *HNS-HAM*.

The hazard template of scald in Figure 4.5 can be used for other appliances and services. For instance, the *HNS-HAM* in Figure 4.4, whose context is “scald by the HotWaterSystem”, is quite different from the *HNS-HAM* for the kettle (in Figure 4.2). However, it can be seen that both models inherit the common structure of the template.

Figure 4.6 shows another template for general hazard context gas poising, and this template is used for constructing the *HNS-HAM* for “gas poising by using the cooking preparation service” in the Figure 4.3.

Figure 4.7 shows the relationship between the hazard template and the *HNS-HAM*. The hazard template has two levels of abstraction, where a general hazard context is explained by general hazardous states. Within the template, these two levels are linked by the “caused-by” relation (denoted by solid arrows), as in the original *HNS-HAM*. Then, the Levels 1 and 2 of the template are respectively linked to those of the *HNS-HAM* by the “realized-by” relation (depicted by dotted arrows).

We assume that every hazard template is constructed carefully by experts in the safety engineering. Once good templates are obtained for various hazard contexts, the cost for constructing concrete HNS-HAMs will be significantly reduced. Moreover, the derived HNS-HAMs will be more reliable and consistent.

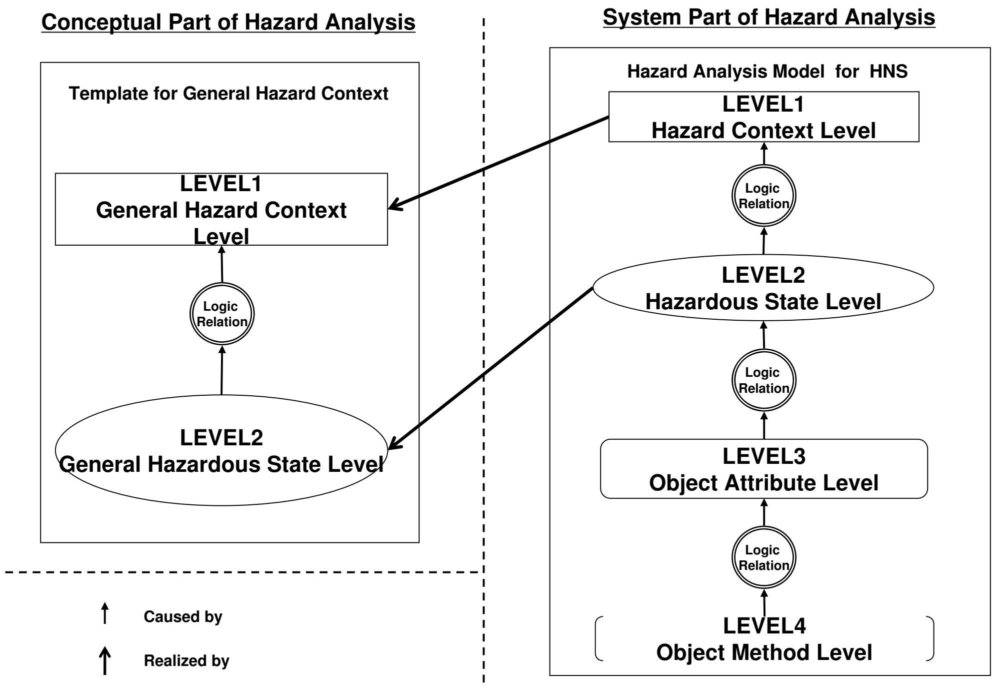


Figure 4.7. Structure and relation between hazard template and HNS-HAM

6. Summary

In order to derive reasonable safety properties, this chapter presented a require ment-engineering approach that can systematically derive the verifiable safety properties for the HNS. Specifically, we proposed a new hazard analysis model, called *HNS Hazard Analysis Model (HNS-HAM)*, which consists of four levels of abstractions. By constructing the *HNS-HAM* and investigating potential hazards within the given HNS model, the safety properties can be derived systematically.

To improve the reusability of *HNS-HAM*, we have also proposed the notion of the *hazard template*, which characterizes the generic portion of the *HNS-HAM*. For every hazard context, the hazard template is supposed to be constructed once by the safety experts. The template can be reused for various kinds of the HNS objects for the common hazard context. The reusable templates make it possible to save the analysis cost and improve the quality of the *HNS-HAM*.

By using the proposed requirement-engineering approach, the potential risks leading to the hazard are analyzed systematically. As a result, the safety properties and their responsible HNS objects are identified. This provides the strong rationale of the safety of the HNS.

To evaluate the effectiveness of the proposed method, we have shown the case study with some practical appliances and services. We have derived 12 local safety properties for `ElecricKettle` and `HotWaterSystem`, 4 global safety properties for `CookingPreparationService` from these *HNS-HAM* models. It was shown that the derived safety properties are reasonable and consistent.

7. Related Work

For the safety requirement analysis of *safety critical systems*, such as, aerospace systems and nuclear plants, several proposal have been addressed [2,3,27,36]. For our proposal, the analysis using the *HNS-HAM* is similar to the *fault tree analysis (FTA)* [42]. However, compared to the conventional fault tree, the *HNS-HAM* has the four level of abstractions customized for the HNS model. Also, our approach is not applied to accidents that were already happened, which is different from the general FTA.

Our idea of safety analysis in a goal-oriented way was originally motivated by the *goal-oriented requirements engineering* [20], which tries to find system requirements in a goal-oriented way. In this area, there is also a language called GRL [10] for the goal-oriented requirement analysis. Basically they are usually applied in the requirements stage where no design specification is developed yet. On the other hand, our problem setting is to find the non-functional requirements (i.e., safety), in the design and validation phase, assuming that the functional specifications of the HNS are available.

Chapter 5

Validating Safety Properties with Design by Contract

1. Introduction

In Chapter 4, we have presented a requirement engineering approach for deriving safety properties in the domain of the home network system. These derived safety properties can be descried by nature language. Once the safety properties were derived, the safety requirement specification of HNS can be updated based on them. These updated HNS specification with considering the safety issues can give some guidelines for designer to consider "*how to design a safe HNS integrated service*".

For a HNS, as we know, it consists of multiple stakeholders. In developing and providing an HNS integrated service, each stakeholder should obey the derived safety properties for guaranteeing the safety of HNS integrated service. Then two issues arise, for a developed HNS, how can we make clarify "*who should be responsible for these derived safety properties ?* " and "*whether did the developers* *obey these derived safety properties or not ?* ". Here, we note that making the above two questions clear is a very important step of our proposal in this dissertation.

In order to solute this issue, we will propose a framework for validating the derived safety properties in the implementation in this chapter. Specifically, we employ the technique of Design by Contract [29] with JML (Java Modeling Language) [16, 17]. The derived safety properties are represented as JML contracts and embedded in Java source code of the appliance, the service and the home objects, the derived safety properties are validated through testing using related testing tools.

2. Exploiting Design by Contract for Safety Validation of HNS

The safety validation problem has been formulated in the Definition 6 of the Section 3 in Chapter 3. We assume that the given HNS and integrated service are implemented based on the object-oriented model presented in the Chapter 2.

As seen in Chapter 2, a 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 responsibility 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) [29].

2.1 Design by Contract

Design by Contract is an approach to designing computer software. It prescribes that software designers should define precise verifiable interface specifications for software components based on some requirement of the system.

The key idea of DbC is that a software system can be seen as a set of communicating components (or entities) which have obligations to other entities based upon formalized rules between them. The specification of software should be created for each object as "contract" before the software is coded. Program execution is then viewed as the interaction between the various modules as bound by these contracts [1]. There are three kinds of contracts in the DbC.

Pre-Condition: A pre-condition of a method m is a condition that must be satisfied *before* executing m, which characterizes a premise of m.

Post-Condition: A post-condition of a method m is a condition that must be satisfied *after* executing m, which characterizes a consequence of 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.

In a human contract case, contracts are written between two parts: the supplier and the client. For the client part, the contract means the client should accept some obligations. The same ideas can apply to software design. In this case, a contract governs the relations between the method (supplier) and any potential caller (client). In our proposal, we consider this kind of contract as *pre-conditions*. For the supplier part, the contract performs some task of the supplier for the client. Each party expects some benefits from the contract, and accepts some obligations in return. In our proposal, we consider this contract as *post-conditions*. At last, if the contract should be followed by both supplier and client, we considered this contract as class *invariant conditions* in our proposal.

2.2 Key Idea: Using Design by Contract

The key idea of our method for validating the safety properties is to *cope with the safety validation problem by first describing LocalProp(s), GlobalProp(s) and EnvProp as the DbC contracts, and then embedding them into the proposed object-oriented model.*

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.

Here, we should note that our research just considers the safety issues of given HNS. In our validation approach, we suppose that the given HNS model has been already verified for satisfying the functional requirement in another approach.

3. Guidelines for Describing Safety Properties as DbC Contracts

Based on the key idea, to implement the validation of safety properties, the first step is describing the derived safety properties as DbC contracts and embedding them in the object. This motivated us to consider *"(A) how to choose the type of contract for each safety property"*, and *"(B) how to choose an object for embedding* *each contract".*

3.1 Choosing the Type of Contract for Each Safety Property

For choosing the type of contract for each safety property, we first need to consider which type of the DbC contracts is appropriate for representing a given safety property. For deriving the safety properties from the HNS-HAM (see Chapter 4), each safety property can be specified as an *invariant* in the specification, intended that the property must hold all the time for the safety. Or, if the safety property is encoded by attributes in the same appliance, we can specify the property as pre/post-conditions of methods designated by Level 4 of HNS-HAM.

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.

3.2 Choosing an Object for Each Contract

For choosing an object for each contract, we must consider carefully which object (`Appliance`, `Service`, or `Home`) should be responsible for $LocalProp(s)$, $GlobalProp(s)$ and $EnvProp$. We suppose that a safety property p is represented as a DbC contract c_p. We then have to choose an appropriate object (class) in Figure 2.3 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 $p \in \mathit{GlobalProp}(s)$, then embed c_p in `Service` or its sub-classes.
- If $p \in \mathit{EnvProp}$, then embed c_p in `Home` or its sub-classes.

Moreover, for choosing the specific attribute and method for each contract, we use level 3 and level 4 of *HNS-HAM* (see Chapter 4) extensively. In HNS-HAM, each safety property is a condition over atomic hazardous states, and Level 3 specifies the correspondence between the atomic states and the object attributes of the model. So each safety property can be encoded by a condition using attributes.

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.

3.3 Examples for Describing Safety Properties

Based on the guideline above, let us describe some contracts for the safety properties described in Chapter 4.

(A) Describing Local Safety Properties

Since the local safety properties are defined for individual appliance, `Appliance` or its sub classes should be responsible for $\mathit{LocalProp}(s)$. For instance, let us update the specification of `ElectricKettle` in Figure 2.6, based on the HNS-HAM in Figure 4.2 and the safety properties P1 to P4 derived in Section 3.3 of Chapter 4.

First we take (P1): When the kettle is in the boiling mode, the lid must not be opened. According to Level 3 and Level 4 of the HNS-HAM, (P1) is encoded to the condition `mode==BOILING => !(lid==OPEN)` over object attributes `mode` and object method `setMode(md)`. To satisfy the above invariant, we can refine the specification of method `setMode(md)` so as to check the lid status.

```
setMode(md) {
 PRE:   power==ON ;
 POST:  mode==md && mode == "BOILING" => lid != "OPEN";
}
```

The updated post-condition says that `setMode(md)` can not set the mode to "BOILING" when the lid is "OPEN". Similarly, we can update the specifications of `ElectricKettle` and `HotWaterSystem` for all the safety properties which were derived in Chapter 4 (see Table 5.1 and Table 5.2).

(B) Describing Global Safety Properties

The global safety properties depend on the contents of each integrated service. Hence, it is reasonable to specify *GlobalProp*(s) as DbC contracts in `Service` or its sub classes. For example, the global safety property "GP1: When the gas valve is opened and fire is on, the ventilator must be turned on", which was derived in Section 3.3 of Chapter 4, can be encoded as the following contract embedded in `CookingPrepatationService`:

```
Contractor:       CookingPreparationService class
Class Invariant: gas.valve == "OPEN" && gas.fire == "OFF"
                 => vent.power != "OFF"
```

The above contract prescribes a condition that at any time when the gas valve is opened and fire is on, the ventilator must be turned on. This contract is a class invariant, which must be guaranteed no matter what operations are executed within the integrated service.

Similarly, we can update the specifications for GP2, GP3 and GP4 as Table 5.3.

(C) Describing Environment Safety Properties

Since the environment safety properties are derived from residential issues, including the house manual, the emergency procedure, community rules and policies, etc. `Home` class should be in charge of *EnvProp*().

For instance, we considered that the environment safety property which is described in Table 5.4 are required for our HNS. The EnvProp1 can be encoded as follows:

```
Contractor:      Home class
Class Invariant: home.currentEnvironment.consumption <= 30
```

The attribute *consumption* is supposed to return the current total consumption of electricity, which is computed from the appliances that are being turned on. This contract is also an invariant, which must be assured whatever services or appliances are operated.

4. Implementing Safety Validation

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

Figure 5.2 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.

```
// Contract LocalProp1:  when the kettle is in the boiling mode, the lid must not be opened.
 /*@ public behavior
   @    requires power.equals("ON");
   @    assignable mode;
   @    ensures mode == wm && power == \old(power);
   @    ensures mode.equals("BOILING") ==> !(lid.equals("OPEN"));      //Contract   LocalProp1
   @*/
 public void setMode(String wm){
         // Implementation
 }

// Contract LocalProp2: when the setting temperature is higher than 60, the lid must not be opened.
  /*@ public behavior
    @    requires power.equals("ON");
    @    assignable temperature;
    @    ensures power == \old(power) && temperature == tem;
    @    ensures (temperature > 60) ==> !(lid.equals("OPEN"));         //Contract   LocalProp2
    @*/
  public void setTemperature(int tem){
         // Implementation
  }

// Contract  LocalProp3: when the lid is opened, the kettle must not be in the boiling mode.
// Contract  LocalProp4: when the lid is opened, the temperature setting must be below 60.
  /*@ public behavior
    @    requires power.equals("ON");
    @    requires !(mode.equals("BOILING"));                          //Contract   LocalProp3
    @    requires temperature < 60;                                   //Contract   LocalProp4
    @    assignable power, isWorking, lid;
    @    ensures power == \old(power) && isWorking == \old(isWorking) && lid.equals("OPEN");
    @*/
  public void openLid(){
         // Implementation
  }
```

Figure 5.1. ElectricKettle.java with JML annotations

4.1 Describing DbC Contracts in JML (Step1)

The proposed method extensively uses the JML (*Java Modeling Language*) [17] 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*. [4, 16, 17]

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 3.

Figure 5.1 shows the JML annotation describing contracts for the property *LocalProp* of *ElectricKettle* (see Table 5.1). The contracts are described in comment lines above method `setMode(String wm)`, `setTemperature(int tm)` and `openLid()`. The line starting with *requires* (or *ensures*) represents the precondition (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 `lid` and `mode` as simple string variables.

As shown in Figure 5.2, 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.

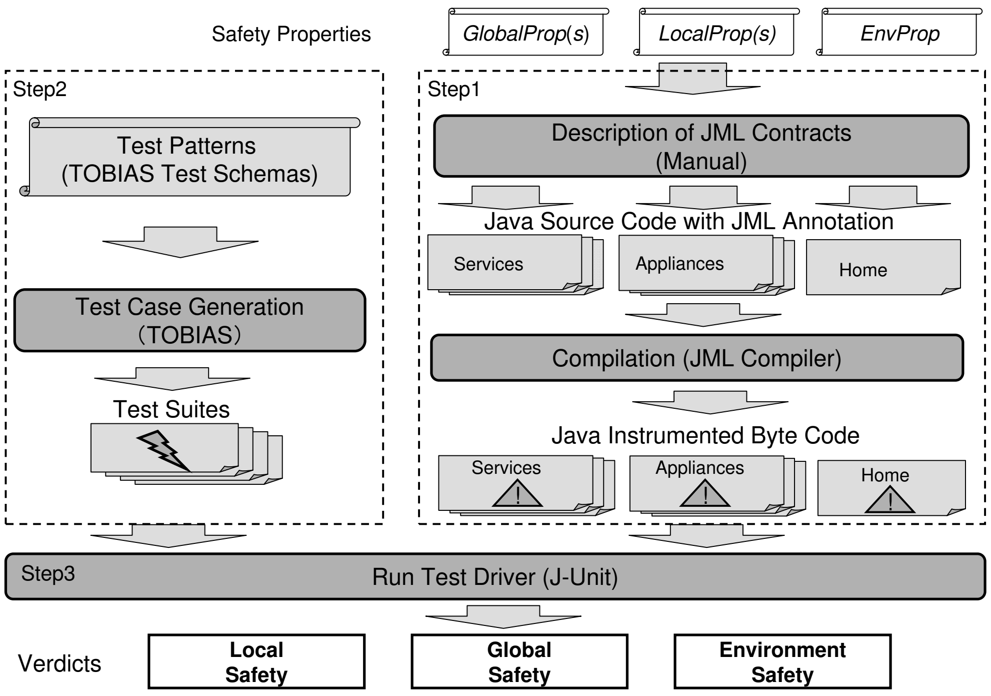


Figure 5.2. Proposed safety validation method

4.2 Generating Test Cases (Step2)

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 [5, 6, 18]. 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 [15], 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 $m_1; m_2; ...; m_n$ ($m_i \in AppOp$). Thus, the part

“$Init; AppOp^n$” is a *preamble* to generate possible states before $s.m()$ is activated. Note that the preamble can also be used as test schemas of individual appliances.

Figure 5.3 shows an example of TOBIAS schemas, which define the preamble using operations of `ElectricKettle` In the figure, `HomeInit` creates a `Home` object. `OneKettleOperation` contains 8 methods for operating the kettle in `home` from outside (see Chapter 2). `ThreeKettleOperaions` is 3-time product of `OneKettleOperation`. 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 512 ($= 8 \times 8 \times 8$) test cases will be unfolded.

By using the same way, we created another example of TOBIAS schemas (Figure 5.4) to test operations of `HotWaterSystem`. Because the `OneHotWaterSystemOperation` contains 8 methods for operating in `home` from outside, the schema `TestAllHotWaterSystem Operations` also created 512 ($= 8 \times 8 \times 8$) test cases for testing. For integrated service of HNS, we created three TOBIAS schemas for testing. (see Figure 5.5 and Figure 5.6.

The first schema `TestCookingPreparationServiceWRTkettleOP` of Figure 5.5 tests the activation of `CookingPreparationService` by using `ThreeKettleOperations` as its preamble.

The second schema `TestBathPreparationServiceWRThotWaterSystemOP` of Figure 5.5 tests the activation of `BathPreparationService` by using `ThreeHotWaterSystemOperations` as its preamble.

The third schema `TestEachServiceWRTenvironmentSettings` of Figure 5.6 deals with one activation of integrated service with initializing *Home* with varying environment parameters. For this, the safety properties concerning the environment can be validated thoroughly.

The test schemas are then translated by TOBIAS into JUnit test classes. For

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

Group OneKettleOperation ::= {
    begin seq home.OnKettle() 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 ,
    begin seq home.SetModeKettle("BOILING" ) end seq ,
    begin seq home.OffKettle() end seq };

Group ThreeKettleOperations ::= OneKettleOperation ^3..3;

Group TestAllKettleOperations ::= { begin seq HomeInit ; ThreeKettleOperations end seq }
```

Figure 5.3. TOBIAS test schemas for electric kettle

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

Group OneHotWaterSystemOperation ::= {
    begin seq home.OnHotWaterSystem() end seq ,
    begin seq home.OffHotWaterSystem() end seq ,
    begin seq home.SetBathTemWaterSystem(98 ) end seq ,
    begin seq home.SetShowerTemWaterSystem(80 ) end seq ,
    begin seq home.OpenShowerHotWaterSystem() end seq ,
    begin seq home.CloseShowerHotWaterSystem() end seq ,
    begin seq home.OpenBathHotWaterSystem() end seq ,
    begin seq home.CloseBathHotWaterSystem() end seq };

Group ThreeHotWaterSystemOperations::= OneHotWaterSystemOperation ^3..3;

Group TestAllHotWaterSystemOperations::=
                              { begin seq HomeInit ; ThreeHotWaterSystemOperations end seq }
```

Figure 5.4. TOBIAS test schemas for hot water system

```
Group TestCookingPreparationServiceWRTkettleOp ::= {
    begin seq
        HomeInit; ThreeKettleOperations;home.CookingPreparationServiceActivation()
    end seq }

Group TestBathPreparationServiceWRThotWaterSystemOp ::= {
    begin seq
        HomeInit; ThreeHotWaterSystemOperations;home.BathPreparationServiceActivation()
    end seq }
```

Figure 5.5. TOBIAS test schemas for service with appliance operations

```
Group OneActivationForEachService ::= {
    begin seq home.CookingPreparationServiceActivation() end seq ,
    begin seq home.BathPreparationServiceActivation() end seq };

Group TestEachServiceWRTevironmentSettings ::= {
    begin seq    Home home := new Home(temperatureValues,12,10,8,0 ) ;
                 OneActivationForEachService
    end seq ,
    begin seq    Home home := new Home(21,brightnessValues,10,8,0 ) ;
                 OneActivationForEachService
    end seq ,
    begin seq    Home home := new Home(22,1,volumeValues,9,15 ) ;
                 OneActivationForEachService
    end seq ,
    begin seq    Home home := new Home(17,20,5,timeValues,13 ) ;
                 OneActivationForEachService
    end seq ,
    begin seq    Home home := new Home(19,2,15,11,powerConsuptionValues ) ;
                 OneActivationForEachService
    end seq }
```

Figure 5.6. TOBIAS test schemas for service with environment settings

instance, from the group `TestEachServiceWRTenvironmentSettings`, the total 66 JUnit tests [1] have been automatically generated within just 0.078 sec. (in a mid-class PC, Pentium-M 1GHz, 760MB). 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.

4.3 Running Test (Step3)

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 Definition Section 2.4 of Chapter 3.

5. Case Study

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

5.1 Preparations

For the experiment, we have prepared Java implementations of 10 appliances and 2 integrated services, as follows:

Appliances: `AirConditioner`, `ElectricKettle`, `GasSystem`, `Light`, `Ventilator`. `DVDPlayer`, `Door`, `SoundSystem`, `TV`, `Curtain`.

[1] `OneActivationForEachService` contained 2 integrated services.

Integrated Services: `CookingPreparationService`, `BathPreparationService`

Although the appliance classes behave as virtual appliances without hardware devices, their source code has been partially taken from the service layer of a real HNS [31] 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 5.1 - 5.4 show a part of safety properties which will be validated in our experiment. The experiment has been performed in a PC with Pentium-M 1GHz, 760MB RAM, Windows XP Pro, J2SDK 1.4.2, JUnit 3.8.1, JML tools release 5.3 and TOBIAS Eclipse plug-in.

5.2 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.

5.3 Results

Table 5.5 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. The table shows only some interesting results. For each safety violation detected, we explain the cause of the violation as follows.

(A) Violation of LocalProp1, LocalProp2, LocalProp3 and LocalProp4

The test cases from `testAllKettleOperations` revealed violations of local safety properties *LocalProp*1 *LocalProp*2 *LocalProp*3 and *LocalProp*4 within the original implementation of `ElectricKettle` class.

First, the violation of *LocalProp*1 and *LocalProp*2 were caused by a omission in methods `kettle.setMode(wm)` and `kettle.setTemperature(tm)`, which sometimes bypassed the checking of the lid status. Hence in some sequences, the kettle entered boiling mode or set the water temperature in high degree (in our test case, the water temperature was set in 98 degree) without checking if the lid was surely closed.

The violation of *LocalProp*3 and *LocalProp*4 caused by a logical error among `setMode(wm)`, `setTemperature(tm)` and `openLid()` method. Therefore, a sequence such as `kettle.setTemperature(98); kettle.openLid()` (*LocalProp*3) or `kettle.setMode("BOILING"); kettle.openLid()` (*LocalProp*4) lead to an unsafe situation where the lid is opened during the boiling mode or setting the water temperature in high degree.

(B) Violation of LocalProp5, LocalProp7, LocalProp9 and LocalProp11

The test cases from `testAllHotWaterSystemOperations` revealed violations of local safety properties *LocalProp*5, *LocalProp*7, *LocalProp*9 and *LocalProp*11 within the implementation of `HotWaterSystem` class.

The violation of *LocalProp*5 and *LocalProp*9 were due to the omission of checking the water temperature in the methods `openShower()` and `openBath`. Therefore when the `bathWaterTemperature` and the `showerWaterTemperature` were set above the limited temperature, `openShower()` and `openBath` lead to an unsafe situation.

The violation of *LocalProp*7 and *LocalProp*11 were due to the omission of checking the water valve status when we executed `setTemperatureforBath()` and `setTemperatureforShower()`. Therefore, the unsafe operation for changing water temperature under the valve opening status was not forbade.

According to the test result, we fixed the errors in `HotWaterSystem` class before proceeding to the next test schema.

(C) Violation of GlobalProp2

In the validation of `testCookingWRTKettleOp`, the proposed method detected the violation of the global safety property *GlobalProp*2 within the `CookingPreparationService` class. The code inspection revealed that the invocation of `ventilator.setVentilatorLevel()` has a mistake in construct method of the service (the wind level was set in 0). Hence, the ventilator did not start the fan although the power of the ventilator was on.

(D) Violation of EnvProp1

We have found that the environment safety property *EnvProp*1 was violated in some test cases from `TestEachServiceWRTevironmentSettings`. When the total power consumption was close to maximum, if `CookingPreparationService` or `BathPreparationService` was activated, the consumption exceeded 35A. To assure environment safety, the home should have a mechanism that estimates the total consumption before every integrated service is activated.

6. Summary

In this chapter, we have proposed a comprehensive framework for describing the derived safety properties and validating the safety of HNS integrated services in the implementation. Moreover, we believe that our key 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.

The limitation of this proposal is that complex 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.

7. Related Work

Despite safety issues of HNS 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.

Pattara et al. [19] proposed a method that verifies the functional properties of HNS integrated services based on *model checking*. A model of services and appliances was proposed and expressed in the SMV language. Expected properties of the services were also expressed and the SMV model-checker was used to prove that the services satisfy the properties. The method gives an automatic and complete proof if given properties hold against an abstract HNS model defined in a finite state space. However, this approach appears to be not sufficient for at least four reasons [6].

First, the use of a model-checker greatly increases confidence in the model. But the model was not derived from the real system. Abstraction and/or mistakes in the model or the incorrect property expression may lead to misleading conclusions.

Second, HNS are supposed to ease the development of new services and/or user applications. It is not reasonable to ask a user to translate the applications to SMV and prove them. At least, an automatic translation would be necessary.

Third, it should be noticed that appliances have some influence on the environment. Switching on (resp. off) an appliance increases (resp. decreases) the power consumption. It may also have an influence on the temperature, the brightness, the sound level in the house. Modification of these parameters can influence the behavior of the whole system. For instance, temperature is measured by the air-conditioner to determine if it should heat or cool the air with respect to the required temperature. Relations between appliances and environment are very difficult to model.

Finally, for HNS, the stakeholders just provide APIs of each object for creating integrated service, so the safety validation of HNS is just wanted to be implemented at implementation level.

Yang et al. [44] 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.

We believe that our proposal has substantial value for validating the safety of HNS.

Table 5.1. Local safety properties of electric kettle

LocalProp1:	When the kettle is in the boiling mode, the lid must not be opened (Electric Kettle).
Contractor:	setMode(wm)
Condition type:	Post-Condition
Specification :	mode == “BOILING” => lid != “OPEN”
LocalProp2:	When the setting temperature is higher than 60, the lid must not be opened (Electric Kettle).
Contractor :	setTemperature(tm)
Condition type:	Post-Condition
Specification :	temperature > 60 => lid != “OPEN”
LocalProp3:	When the lid is opened, the kettle must not be in the boiling mode (Electric Kettle).
Contractor :	openLid()
Condition type:	Pre-Condition
Specification :	mode != “BOILING”
LocalProp4:	When lid is opened, the temperature setting must below 60 (Electric Kettle).
Contractor :	openLid()
Condition type:	Pre-Condition
Specification :	temperature <= 60

Table 5.2. Local safety properties of hot water system

LocalProp5:	While the shower valve is opened, the shower water temperature setting must not be above 40 (Hot Water System).
Target Object :	openShower()
Condition type:	Post-Condition
Specification :	showerValve == “OPEN” => showerWaterTemperature <= 40
LocalProp6:	While the shower valve is opened, the shower water temperature must be not changed (Hot Water System).
Target Object :	openShower()
Condition type:	Post-Condition
Specification :	showerWaterTemperature == old(showerWaterTemperature)
LocalProp7:	While the shower water temperature setting is above 40, the shower water valve must not be opened (Hot Water System).
Target Object :	setTemperatureforShower()
Condition type:	Pre-Condition
Specification :	tem > 40 => showerValve != “OPEN”
LocalProp8:	While the shower water temperature is changed, the shower water valve must not be opened (Hot Water System).
Target Object :	setTemperatureforShower(tm)
Condition type:	Pre-Condition
Specification :	showerValve != “OPEN”
LocalProp9:	While the bath valve is opened, the bath water temperature setting must not be above 60 (Hot Water System).
Target Object :	openBath()
Condition type:	Post-Condition
Specification :	bathValve == “OPEN” => bathWaterTemperature < 60
LocalProp10:	While the bath valve is opened, the bath water temperature must be not changed (Hot Water System).
Target Object :	openBath()
Condition type:	Post-Condition
Specification :	bathWaterTemperature == old(bathWaterTemperature)
LocalProp11:	While the bath water temperature setting is above 60, the bath water valve must not be opened (Hot Water System).
Target Object :	setTemperatureforBath()
Condition type:	Pre-Condition
Specification :	tem > 60 => bathValve != “OPEN”
LocalProp12:	While the bath water temperature is changed, the bath water valve must not be opened (Hot Water System).
Target Object :	setTemperatureforBath(tm)
Condition type:	Pre-Condition
Specification :	bathValve != “OPEN”

Table 5.3. Global safety properties of cooking preparation service

GlobalProp1:	When gas valve is opened and fire is on, the ventilator must be turned on (for cookingPreparationService).
Target Object :	CookingPreparationService
Condition type:	Invariant Condition
Specification :	gas.valve == “OPEN” && gas.fire == “ON” => vent.power == “ON”
GlobalProp2:	When gas valve is opened and fire is on, the ventilator wind level must not be 0 (for cookingPreparationService).
Target Object :	CookingPreparationService
Condition type:	Invariant Condition
Specification :	gas.valve == “OPEN” && gas.fire == “ON” => vent.windLevel !=0
GlobalProp3:	When the ventilator power is in off, the gas valve must not be opened or fire must not be on (for cookingPreparationService).
Target Object :	CookingPreparationService
Condition type:	Invariant Condition
Specification :	vent.power! = “OFF” => (gas.valve != “OPEN” || gas.fire != “ON”)
GlobalProp4:	When the ventilator wind level is 0, the gas valve must not be opened or fire must not be on (for cookingPreparationService).
Target Object :	CookingPreparationService
Condition type:	Invariant Condition
Specification :	vent.windLevel == 0 => (gas.valve != “OPEN” || gas.fire != “ON”)

Table 5.4. Environment safety properties of home

EnvProp1:	The total amount of current used simultaneously must not exceed 35A.
Target Object :	Home class
Condition type:	Invariant Condition
Specification :	home.currentEnvironment.getTotalConsumption() <= 35
EnvProp2:	Do not make a loud noise or sound after 9 p.m.
Target Object :	Home class
Condition type:	Invariant Condition
Specification :	home.time >= 21 => home.currentEnvironment.voiceVolume <= 80

Table 5.5. Results of safety validation

TOBIAS Test Schemas	# of Total TC	# of Failed TC	Time Elapsed	Safety Violation
TestAllKettleOperations	512	397	2.375 sec.	LocalProp1, LocalProp2 LocalProp3, LocalProp4
TestAllHotWaterSystemOperations	512	476	2.125 sec.	LocalProp5 LocalProp7 LocalProp9, LocalProp11
TestCookingPreparationWRTkettleOp	512	512	2.719 sec.	GlobalProp2
TestBathPreparationWRThotWaterSystemOp	512	0	1.469 sec.	null
TestEachServiceWRTevironmentSettings	66	4	0.187 sec.	EnvProp1

Chapter 6

Conclusion

1. Achievement

In this dissertation, we have proposed a total framework for characterizing, deriving and validating the safety properties within the integrated service of emerging home network system. The primary purpose of the framework is to give a strong rationale and a systematic method for addressing the safety issues of HNS from the system specification design stage to the testing stage.

First, we introduced the HNS by giving some examples of the integrated services. To clarify the whole structure formed by the HNS objects, we presented an *object-oriented model* for HNS. Then to capture the behavior of each HNS object, we introduced a specification language, which describes the design requirements of the HNS.

Second, we formalized the safety of the HNS by considering the nature of the HNS and integrated services. The safety was defined as (1) *local safety* which is the safety to be ensured by the safety instructions of individual appliances, (2) *global safety* which is specified over multiple appliances as required properties of an integrated service, and (3) *environment safety* which is prescribed as residential rules in the home and surrounding environment. We also formulated the safety validation problem based on the safety classification.

Third, in order to derive the correct and complete safety properties, we proposed a requirement-engineering approach for deriving the verifiable safety properties systematically. The approach is realized by constructing a new hazard analysis model, called *HNS-HAM (HNS Hazard Analysis Model)*, which investigates potential hazards within the given HNS model. The *hazard analysis model* consists of four levels: *(Level 1) hazard context*, *(Level 2) hazardous state*, *(Level 3) object attribute* and *(Level 4) object method*. The construction of *HNS-HAM* yields cause-and-effect chains from the abstract hazard contexts to the concrete attributes and operations of HNS objects, specified in the given HNS specification. In order to enhance the reusability of the HNS-HAM, we have also discussed a technique using *hazard template*.

Finally, we proposed a framework that validates the derived safety properties for the given implementation. The proposed validation method extensively uses the technique of *design by contract (DbC, for short)*. Specifically, we describe each safety property as a set of DbC contracts between calling and callee objects. Then we embed the contracts into the proposed object-oriented model. The contracts can be 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, the safety validation problem can be reduced to the testing problem of the HNS implementations. To evaluate the proposed method, we also have conducted a case study for validating the derived safety properties above. During the validating, several safety property violations were detected.

We believe that the proposed total framework can help the HNS developers significantly in designing and implementing safe HNS solutions.

2. Future Directions

We consider that the proposed safety framework is not limited within the HNS only. It can be applied to the safety issues in various kinds of other ubiquitous and distributed systems, where every service is implemented by integrating multiple objects [37]. The typical applications include the *building control system* [26, 33, 34], the *health telematics system* [40], and the *automotive network system* [14,43].

One future direction is to *generalize* the proposed framework so that it can be used in a broader domain. Our perspective of the generalization is justified by the following characteristics of the proposed framework.

(A) The structure of the HNS is not so unique in the ubiquitous domain

The structure of the HNS, where a service is implemented by multiple autonomous and self-contained objects, is not quite specific to the HNS. It can be seen in general ubiquitous and service-oriented distributed systems. Thus, we believe that the proposed object-oriented model can be adapted to other ubiquitous systems without much effort.

(B) The safety definition and validation methods are general

In such ubiquitous and distributed systems, the safety issues should be addressed carefully since many stakeholders are involved in developing the system. It is important to clarify who bears the responsibility of every instance of the safety issue. For this, the proposed three kinds of safety properties (with the validation by the DbC) works reasonably. We should note that the emerging ubiquitous systems are not quite the same as the conventional safety-critical systems, where all components are tightly coupled to achieve a monolithic and reliable system.

(C) The hazard templates can be reused in other domains

The idea of using the hazard template, which extracts the system-independent portion of the hazard analysis, is not limited in the hazard analysis of the HNS. Once the solid hazard templates are constructed, the templates can be used for other ubiquitous systems. We need to investigate the framework on how to prepare the solid template for every kind of hazard.

Another future direction is to improve the performance of the proposed method through more experimental evaluations. Application to the existing HNS implementation [31] to find the safety risk is also an interesting challenge.

Acknowledgements

The research of this dissertation could not have been accomplished without the collaboration of many other people.

First, I would like to thank my supervisor Professor Ken-ichi Matsumoto, Graduate School of Information Science, Nara Institute of Science and Technology. Professor Matsumoto gave me a lot of advice in not only my research work but also daily life. I would like to thank him for his valuable advice in the laboratory.

I am also very grateful to the members of my thesis review committee: Professor Hiroyuki Seki and Associate Professor Akito Monden for their valuable and insightful comments.

I would like to sincerely, deeply thank Associate Professor Masahide Nakamura, the Graduate School of Engineering at Kobe University, for his professional advice and technical help, extensive guidance, patience, encouragement and support through the course of this work. Without his help, I could not have the opportunity to work in his research group. I would like to thank so much for his kindness and help in all aspects of my research and daily life.

I would like to thank Associate Professor Lydie du Bousquet, Joseph Fourier University, Grenoble, France, for her professional advice and technical help, through the course of this work.

I would like to express my gratitude to all the teachers and students in the software engineering laboratory, and the students who have graduated in the last two years. It is a pleasant and fruitful experience for me to work with them, and thank for making a good and friendly research environment.

I would like to thank Professor Soichi Onishi, Graduate School of Informatics, Okayama University of Science, who was the supervisor of my master's course. Professor Onishi gave me a lot of advice in not only my master research but also daily life during the first two years after I came to Japan. Without his recommendation, I could not continue doctoral course in Nara Institute of Science and Technology. I really appreciate his kindness and help during my master's period.

Finally, I would like to express my warmest gratitude to my parents, my wife, my daughter, and my sister for their constant encouragement and generous remarks.

The research in this dissertation has been 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 JSPS and MAE under the Japan-France Integrated Action Program (SAKURA).

References

[1] A. I. Andronescu and O. Muntean.: Formal Specification of Business Components - a Design by Contract Perspective, Proceedings of 6th International Conference on Computer Systems and Technologies (CompSysTech'06), Vol.2, pp.1-6, Jun.2006.

[2] K. Allenby and T. Kelly.: Deriving Safety Requirements Using Scenarios, Proceedings of the 5th International Symposium on Requirements Engineering (RE'01), pp.228-235, Aug.2001.

[3] J.L. Boulanger, V. Delebarre, S. Natkin and J.P. Ozello.: Deriving Safety Properties of Critical Software from the System Risk Analysis, Application to Ground Transportation Systems, Proceedings of 2nd IEEE High-Assurance Systems Engineering Workshop, pp.228-235, 1997.

[4] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. T. Leavens, K. Rustan M. Leino, and E. Poll.: An Overview of JML Tools and Applications, International Journal on Software Tools for Technology Transfer (STTT), Vol.7, No.3, pp.212-232, Jun.2005.

[5] L. du Bousquet, Y. Ledru, O. Maury, and P. Bontron.: A Case Study in JML-based Software Validation, Proceedings of 19th IEEE International Conferences on Automated Software Engineering (ASE'04), Linz, pp.294-297, IEEE Computer Society Press, Sep.2004.

[6] L. du Bousquet, M. Nakamura, B. Yan, and K. Matsumoto.: Using Formal Methods to Increase Confidence in a Home Network System Implementation, Case Study, 2007 ISoLA Workshop On Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2007), pp.203-214, Dec.2007.

[7] C. Damas, B. Lambeau and A. van Lamsweerde.: Scenarios, Goals, and State Machines: a Win-Win Partnership for Model Synthesis, Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp.197-207, Nov.2006.

[8] Functional Safety and IEC 61508, available from (www.iec.ch/zone/fsafety/pdf-safe/hld.pdf).

[9] P. Gupta and J. Schumann.: A Tool for Verification and Validation of Neural Network Based Adaptive Controllers for High Assurance Systems, Proceedings of 8th IEEE International Symposium on High Assurance Systems Engineering (HASE'04), pp.277-278, 2004.

[10] Goal-oriented Requirement Language (GRL), available from (www.cs.toronto.edu/km/GRL/)

[11] International Electrotechnical Commission, Household and Similar Electrical Appliances — Safety, IEC 60335-1, Sep.2006.

[12] International Electrotechnical Commission, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems IEC61508, available from (www.ja.wikipedia.org/wiki/IEC61508).

[13] H. Igaki, M. Nakamura, K. Matsumoto, and M. Aoyama.: Adopting Model-Driven Development for Integrated Services and Appliances in Home Network Systems, Proceedings of 13th Asia-Pacific Software Engineering Conference (APSEC 2006), pp.45-52, Dec.2006.

[14] K. Itao.: Sensor Network Technology for Realization of Safe and Secure Driving, Journal of Society of Automotive Engineers of Japan, Vol.61, No.2(20070201), pp.107-113.

[15] JUnit, Testing Resources for Extreme Programming, available from (www.junit.org/).

[16] The Java Modeling Language (JML), available from (www.eecs.ucf.edu/ leavens/JML/).

[17] G. T. Leavens and Y. Cheon.: Design by Contract with JML, available from (www.jmlspecs.org), May.2006.

[18] Y. Ledru, L. du Bousquet, O. Maury, and P. Bontron.: Filtering TOBIAS Combinatorial Test Suites, Proceedings of International Conferences on Fundamental Approaches to Software Engineering (ETAPS/FASE'04), LNCS 2984, Springer-Verlag, Mar.2004.

[19] P. Leelaprute, M. Nakamura, T. Tsuchiya, K. Matsumoto, and T. Kikuno.: Describing and Verifying Integrated Services of Home Network Systems, Proceedings of 12th Asia-Pacific Software Engineering Conferences (APSEC 2005), pp.549-558, Dec.2005.

[20] E. Letier and A. van Lamsweerde.: Deriving Operational Software Specifications from System Goals, Proceedings of 10th ACM S1GSOFT symposium on the Foundations of Software Engineering (FSE'10), Charleston, Nov.2002.

[21] N. G. Leveson.: Safeware: System Safety and Computers, Addison-Wesley, 1995.

[22] N. G. Leveson and K. A. Weiss.: A New Accident Model for Engineering Safer Systems, Journal of Safety Science, Vol.42, No.4, pp.237-270, Apr.2004.

[23] N. G. Leveson.: Safety in Integrated Systems Health Engineering and Management Proceedings of Safety Science, Vol.42, No.4, Apr.2004.

[24] N. G. Leveson, K. A. Weiss.: Making Embedded Software Reuse Practical and Safe, Proceedings of Foundations of Software Engineering, Nov.2004.

[25] N. G. Leveson.: A Systems-Theoretic Approach to Safety in Software-Intensive Systems, IEEE Transactions on Dependable and Secure Computing, Vol.1, Is.1, pp.66-86, Jan.2004.

[26] A. Metzger, C. Webel.: Proceedings of Feature Interaction Detection in Building Control Systems by Means of a Formal Product Model, Feature Interaction in Telecommunications and Software Systems VII, Amsterdam, IOS Press, pp.105-121, 2003.

[27] M. A. de Miguel, B. Pauly, T. Person and J. Fernandez.: Model-Based Integration of Safety Analysis and Reliable Software Development, Proceedings of the 10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'05), Is.2-4, pp.312-319, Feb.2005.

[28] M. Motoharu, Y. Takashi, N. Masaharu and T. Atsushi.: Home Network Technology Trends and Interworking between DLNA and Mobile Devices, IEICE Technical Report. Information networks, Vol.107, No.314(20071108) pp. 25-30, IN2007-93, 2007.

[29] B. Meyer.: Applying Design by Contract, IEEE Computer, vol.25, No.10, pp.40-51, Oct.1992.

[30] A. Mili, G. Jiang, B. Cukic, Y. Liu and RB. Ayed.: Towards the Verification and Validation of Online Learning Systems: General Framework and Applications, Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04), pp.90304a, Sep.2004.

[31] M. Nakamura, A. Tanaka, H. Igaki, H. Tamada, and K. Matsumoto.: Adapting Legacy Home Appliances to Home Network Systems Using Web Services, Proceedings of International Conferences on Web Services (ICWS 2006), pp.849-858, Sep.2006.

[32] M. Nakamura, H. Igaki, and K. Matsumoto.: Feature Interactions in Integrated Services of Networked Home Appliances -An Object-Oriented Approach-, Proceedings of International Conferences on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05), pp.236-251, Jul.2005.

[33] S. Nishizawa, M. Nakamura, H. Igaki, K. Matsumoto, and K. Miura.: Applying Service-Oriented Architecture to Building Management Systems - Integration of Heterogeneous Services and Consideration of Safety-, IEICE Technical Report, Vol.107, No.261 pp.3-8, NS2007-81, Oct.2007. (in Japanese)

[34] S. Nishizawa, M. Nakamura, H. Igaki, K. Matsumoto, and K. Miura.: Formalizing Heterogeneous Building Automation Systems and Feature Interaction Problem, SIG Technical Report, Vol.2008, No.29 pp.179-186, Mar.2008. (in Japanese)

[35] B. Nicolescu, R. Velazco, M. Sonza-Reorda, M. Rebaudengo and M. Violante.: A Software Fault Tolerance Method for Safety-Critical Systems: Effectiveness and Drawbacks, Proceedings of 15th Symposium on Integrated Circuits and Systems Design (SBCCI'02), pp.101-106, 2002.

[36] C. Ponsarda, P. Massoneta, A. Rifauta, J.F. Moldereza, A. van Lamsweerdea, and H. Tran Vana.: Early Verification and Validation of Mission Critical Systems, Proceedings of the 9th International Workshop on Formal

Methods for Industrial Critical Systems (FMICS 2004), Vol.133, pp.237-254, May 2005.

[37] M. P. Papazoglou and D. Georgakopoulos.: Service-Oriented Computing, Communications of the ACM, Vol.46, No.10, pp.25-28, Sep.2003.

[38] S. Supakkul and L. Chung.: Applying a Goal-Oriented Method for Hazard Analysis: A Case Study, Proceedings of the 4th International Conference on Software Engineering Research, Management and Applications (SERA '06), Is.09-11, pp.22-30, Aug.2006.

[39] R. Socher.: Optimizing the Clausal Normal form Transformation, Journal of Automated Reasoning, Vol.7, No.3, pp. 325-336, 1991.

[40] V.F. Siang Fook, J.H. Tee, K.S. Yap, A.A. Phyo wai, J. Maniyeri, B. Jit, and P. Hin Lee.: Smart Mote-Based Medical System for Monitoring and Handling Medication Among Persons with Dementia, Proceedings of 5th International Conferences on Smart Homes and Health Telematics (ICOST2007), LNCS 4541, pp.54-62, Jun.2007.

[41] Y. Tajika, A. Toba and S. Kyuuma.: Standard Technologies for Home Network Systems (Bluetooth^{TM}, ECHONET^{TM}), Journal of TOSHIBA REVIEW, Vo.57, No.10, pp.11-15, 2002.

[42] J. Xiang, K. Futatsugi, and Y. He.: Formal Fault Tree Construction and System Safety Analysis, Proceedings of IASTED International Conferences on Software Engineering, pp.378-384, Feb.2004.

[43] T. Yokoyama.: A Distributed System Framework for Automotive Control, IEICE Technical Report, Vol.99, No.725, pp.3-10, CPSY99-118, Mar.2000. (in Japanese)

[44] H.-I Yang, J. King, S. Helal, and E. Jansen.: A Context-Driven Programming Model for Pervasive Spaces, Proceedings of 5th International Conferences on Smart homes and health Telematics (ICOST2007), pp.31-43, Jun.2007.

Appendix

A. Specification of Appliance

```
APPLIANCE Appliance
   ATTRIBUTES
      power    : {ON, OFF} = OFF;
   METHODS
      on(){
         PRE:   power == OFF;
         POST:  power == ON;
      }
      Off(){
         PRE:   power == ON;
         POST:  power == OFF;
      }
      getPower(){
         PRE:   power == ON;
         POST:  ReturnValue == power;
      }
   INVARIANTS
      true;
```

B. Specification of HotWaterSystem

```
APPLIANCE HotWaterSystem EXTENDS Appliance
   ATTRIBUTES
      bathWaterTemperature     : {30, 31···49, 50} = 40;
      showerWaterTemperature   : {30, 31···39, 40} = 35;
      showerValve              : {OPEN, CLOSE} = CLOSE;
      bathValve                : {OPEN, CLOSE} = CLOSE;
      showerIsWorking          : {true, false} = false;
      bathIsWorking            : {true, false} = false;
   METHODS
      openShower(){
         PRE:   power == ON && showerValve == CLOSE && showerIsWorking ==false;
         POST:  showerValve == OPEN && showerIsWorking == true;
      }
      closeShower(){
         PRE:   power == ON &&  showerValve == OPEN && showerIsWorking == true;
         POST:  showerValve == CLOSE && showerIsWorking == false;
      }
      openBath(){
         PRE:   power == ON && bathValve == CLOSE && bathIsWorking ==false;
         POST:  bathValve == OPEN && bathIsWorking == true;
      }
      closeBath(){
         PRE:   power == ON && bathValve == OPEN && bathIsWorking ==true;
         POST:  bathValve == CLOSE && bathIsWorking == false;
      }
      setTemperatureforBath(tem){
         PRE:   power == ON && bathIsWorking == false;
         POST:  bathWaterTemperature == tem;
      }
      setTemperatureforShower(tem){
         PRE:   power == ON && showerIsWorking == false;
         POST:  showerWaterTemperature == tem;
      }
      getShowerWorkingStatus(){
         PRE:   power == ON;
         POST:  ReturnValue == showerIsWorking;
      }
      getBathWorkingStatus(){
         PRE:   power == ON;
         POST:  ReturnValue == bathIsWorking;
      }
      getBathWaterTemperature(){
         PRE:   power == ON;
         POST:  ReturnValue == bathWaterTemperature;
      }
      getShowerWaterTemperature(){
         PRE:   power == ON;
         POST:  ReturnValue == showerWaterTemperature;
      }
   INVARIANTS
      true;
```

C. Specification of Air-Conditioner

```
APPLIANCE Air-Conditioner EXTENDS Appliance
   ATTRIBUTES
      requiredTemperature   : {20, 21···29, 30} = 20;
      currentTemperature    : {20, 21···29, 30} = 20;
      mode                  : {COLDING, HEATING, WAITING} = WAITING;
      windLevel             : {SOFT, MID, STRONG} = SOFT;
      isWorking             : {false, true} = false;
   METHODS
      switchOn(){
         PRE:   power == ON && isWorking == false;
         POST:  isWorking == true;
      }
      switchOff(){
         PRE:   power == ON && isWorking == true;
         POST:  isWorking == false;
      }
      setNewMode(){
         PRE:   power == ON;
         POST:  mode == COLDING || mode == HEATING || mode == WAITING;
      }
      setRequiredTemperature(tp){
         PRE:   power == ON;
         POST:  requiredTemperature == tp;
      }
      upRequiredTemperature(){
         PRE:   power == ON && requiredTemperature < 30;
         POST:  requiredTemperature == requiredTemperature + 1;
      }
      downRequiredTemperature(){
         PRE:   power == ON && requiredTemperature > 20;
         POST:  requiredTemperature == requiredTemperature - 1;
      }
      setWindLevel(wl){
         PRE:   power == ON;
         POST:  windLevel == wl;
      }
   getRequiredTemperature(){
      PRE:   power == ON;
      POST:  ReturnValue == requiredTemperature;
   }
   getCurrentTemperature(){
      PRE:   power == ON;
      POST:  ReturnValue == currentTemperature;
   }
   getWindLevel(){
      PRE:   power == ON;
      POST:  ReturnValue == windLevel;
   }
   getMode(){
      PRE:   power == ON;
      POST:  ReturnValue == mode;
   }
   getWorkingStatus(){
      PRE:   power == ON;
      POST:  ReturnValue == isWorking;
   }
INVARIANTS
   true;
```

D. Specification of WaterValve

```
APPLIANCE WaterValve EXTENDS Appliance
   ATTRIBUTES
      valve          : {OPEN, CLOSE} = CLOSE;
      waterVolume    : {LOW, MID, LARGE} = LOW;
   METHODS
      open(){
         PRE:    power == ON && valve == CLOSE;
         POST:   valve == OPEN;
      }
      close(){
         PRE:    power == ON && valve == OPEN;
         POST:   valve == CLOSE;
      }
      setWaterVolume(lev){
         PRE:    power == ON;
         POST:   waterLevel == lev;
      }
      getValve(){
         PRE:    power == ON;
         POST:   ReturnValue == valve;
      }
      getWaterVolume(){
         PRE:    power == ON && valve == OPEN;
         POST:   ReturnValue== waterVolume;
      }
   INVARIANTS
      true;
```

E. Specification of Light

```
APPLIANCE Light EXTENDS Appliance
   ATTRIBUTES
      isWorking     : {true, false} = false ;
      brightness    : {SOFT, MID, STRONG} = MID;
   METHODS
      switchOn(){
         PRE:   power == ON && isWorking == false;
         POST:  isWorking == true;
      }
      switchOff(){
         PRE:   power == ON && isWorking == true;
         POST:  isWorking == false;
      }
      setBrightnessLevel(lev){
         PRE:   power == ON && (lev == SOFT || lev == MID || lev == STRONG);
         POST:  brightness == lev;
      }
      getWorkingStatus(){
         PRE:   power == ON;
         POST:  ReturnValue == isWorking;
      }
      getBrightness(){
         PRE:   power == ON;
         POST:  ReturnValue == brightness;
      }
   INVARIANTS
      true;
```

F. Specification of Ventilator

```
APPLIANCE Ventilator EXTENDS Appliance
   ATTRIBUTES
      isWorking     : {false, true} = false;
      windLevel     : {0, 1, 2, 3} = 0;
   METHODS
      switchOn(){
         PRE:   power == ON && isWorking == false;
         POST:  isWorking == true;
      }
      switchOff(){
         PRE:   power == ON && isWorking == true;
         POST:  isWorking == false;
      }
      setWindLevel(lev){
         PRE:   power == ON && (lev == 0 || lev == 1 || lev == 2 || lev ==3);
         POST:  windLevel == lev;
      }
      upWindLevel(){
         PRE:   power == ON && isWorking == true && windLevel < 3;
         POST:  windLevel == windLevel + 1;
      }
      downWindLevel(){
         PRE:   power == ON && isWorking == true && windLevel > 0;
         POST:  windLevel == windLevel - 1;
      }
      getWorkingStatus(){
         PRE:   power == ON;
         POST:  ReturnValue == isWorking;
      }
      getWindLevel(){
         PRE:    power == ON;
         POST:   ReturnValue == windLevel;
      }
   INVARIANTS
      true;
```

G. Specification of GasSystem

```
APPLIANCE GasSystem EXTENDS Appliance
   ATTRIBUTES
     isWorking    : {false, true} = false;
     fireLevel    : {1, 2, 3} = 1;
     valve        : {OPEN, CLOSE} = CLOSE;
   METHODS
     fireOn(){
        PRE:    power == ON && valve == OPEN && isWorking == false;
        POST:   isWorking == true;
     }
     fireOff(){
        PRE:    power == ON && valve == OPEN && isWorking == true;
        POST:   isWorking == false;
     }
     setFireLevel(lev){
        PRE:    power == ON;
        POST:   fireLevel == lev;
     }
     openGasValve(){
        PRE:    power == ON && valve == CLOSE && isWroking == false;
        POST:   valve == OPEN;
     }
     closeGasValve(){
        PRE:    power == ON && valve == OPEN && isWorking == false;
        POST:   valve == CLOSE;
     }
     getWorkingStatus(){
        PRE:    power == ON;
        POST:   ReturnValue == isWorking;
     }
     getFireLevel(){
        PRE:    power == ON;
        POST:   ReturnValue == fireLevel;
     }
     getGasValve(){
        PRE:    power == ON;
        POST:   ReturnValue == valve;
   INVARIANTS
     true;
```

H. Specification of Service

```
SERVICE Service
   ATTRIBUTES
      serviceNumbber    : {0, 1, 2, 3} = 0;
      isWorking         : {true, false} = false;
   METHODS
      getServiceNumber(){
         PRE:   true;
         POST:  ReturnValue == serviceNumbber;
      }
      getWorkingStatus(){
         PRE:   true;
         POST:  ReturnValue == isWorking;
      }
   INVARIANTS
      true;
```

I. Specification of BathPreparationService

```
SERVICE BathPreparationService EXTENDS Service
  APPLIANCES
     hotWaterSys      : hotWaterSystem;
     bathLight        : Light;
     bathAirCon       : airConditioner;
  ATTRIBUTES:
     bathWaterTem       : {30, 31···49, 50} = 40;
     bathRoomBrightness : {SOFT, MID, STRONG} = MID;
     bathAirConWind     : {SOFT, MID, STRONG} = SOFT;
     bathAirConTem      : {20···35} = *;
  METHODS
     activation(){
        PRE:   service.isWorking == false;
        POST:  hotWaterSys.power == ON && hotWaterSys.waterTemperatureForBath == bathWaterTem &&
               hotWaterSys.bathIsWorking == true && hotWaterSys.bathValve == OPEN &&
               bathLight.power == ON && bathLight.brightness == bathRoomBrightness &&
               bathLight.isWorking == true && bathAirCon.Power == ON &&
               bathAirCon.isWorking == true && bathAirCon.windLevel == bathAirConWind &&
               bathAirCon.requiredTemperature == bathAirConTem && service.isWorking == true;
     }
     stop(){
        PRE:   service.isWorking == true;
        POST:  hotWaterSys.bathValve == CLOSE && hotWaterSys.bathIsWorking == false &&
               hotWaterSys.power == OFF && bathLight.isWorking == false &&
               bathLight.power == OFF && bathAirCon.isWorking == false &&
               bathAirCon.power == OFF && service.isWorking == false;
     }
     setBathWaterTemperature (tem){
        PRE:   service.isWorking == false;
        POST:  bathWaterTem == tem;
     }
     setBathRoomLightBrightness (lb){
        PRE:   service.isWorking == false;
        POST:  bathRoomBrightness == lb;
     }
     setBathRoomAirConTemperature (tem){
        PRE:   service.isWorking == false;
        POST:  bathAirConTem == tem;
     }
  setBathRoomAirConWindLevel (wl){
    PRE:   service.isWorking == false;
    POST:  bathAirConWind == wl;
  }
INVARIANTS
  true;
```