

Service specification and validation in the context of the home

Lydie DU BOUSQUET ^{a,1}, Ajitha RAJAN ^a and Catherine ORIAT ^a and
Jean-Luc RICHIER ^a and German VEGA ^a

^a *LIG, Universités de Grenoble, BP 72, 38402 Saint Martin d'Hères, France*

Abstract. We explore and discuss different ways of expressing service specifications in the context of home automation systems implemented on OSGiTM. We found that the approach used for expressing services affects the amount of service interaction in the system. Some approaches, as opposed to others, artificially increase the number of situations where services are considered to badly interact. We discuss the pros and cons of three approaches for service specification in the context of the home.

Keywords. Home automation, specification, JML, OSGiTM, iPOJO

1. Introduction

Emerging technologies enable general household appliances to be connected to LAN at home. Such smart home appliances are generally called networked appliances. A Home Network System (HNS) consists of multiple networked appliances, intended to provide more convenient and comfortable living for home users. Research and development of the HNS is currently a hot topic in the area of ubiquitous/pervasive computing [29,20,3,22,28].

The HNS can provide several applications and services. They typically take advantage of wide-range control and monitoring of appliances inside and outside the home. Integrating different appliances via a network yields more value-added and powerful services, which are called HNS *integrated services* [19]. For instance, orchestrating a TV, a DVD player, 5.1ch speakers, lights, curtains and an air-conditioner allows to provide an integrated service, called *DVD theater service*, where a user can watch movies in a theater-like atmosphere.

The Laboratoire d'Informatique de Grenoble (LIG) has developed a platform for home automation, called H-Omega [5,13]. This is a gateway, implemented on OSGiTM and iPOJO, that eases the creation and deployment of new services. The new services are developed as classical Java components and are embedded in containers that deals with technical issues such as heterogeneity and dynamism.

¹Correspondence to: L. du Bousquet, LIG, BP 72, F-38402 Saint Martin d'Hères Cedex. Tel.: +33 476 82 72 32; Fax: +33 476 82 72 87; E-mail: lydie.du-bousquet@imag.fr

This work has been partially funded by the Université Joseph Fourier iPOTest project.

The dynamism of service-oriented applications like the home automation system impose a shift in the V&V process. The Home Network System is required to exhibit qualities such as correctness and safety with high assurance in the presence of service dynamism. It has been identified that home services are likely to interact [23,25,32]: a new service can change the behavior of pre-existing ones, break them, or even crash the system. This is a well known problem in the telecommunication industry known as the “feature interaction problem” [15]. Let us consider two services S_1 and S_2 , specified by two set of properties P_1 and P_2 (also called requirements in the literature). Validating S_i consists of comparing behaviors of S_i and expected behaviors characterized by P_i . Now suppose that S_1 and S_2 respectively satisfy their properties P_1 and P_2 when considered in isolation. Informally, we say that S_1 and S_2 interact whenever either P_1 or P_2 or both are violated when the services S_1 and S_2 are considered together.

To tackle this feature interaction problem, a large set of approaches have been studied [4,8,9,18,7,1,27,12]. One of them is to formally specify the services and use the specification for proof, for test, or for monitoring, on-line or off-line. We successfully used this approach [11,31].

This paper is an illustration of how a service can be specified, developed and deployed in the H-Omega platform. In particular, we focus on the different ways of specifying a service. We observed that some approaches to specifying services are more “interaction” prone than others. We used the Java Modeling Language (JML) to specify services [31]. JML is an executable specification language for Java, based on assertions. They can be embedded in the components and deployed on the gateway, to monitor the execution.

Section 2 gives a short presentation of the gateway, developed by the LIG laboratory. Section 3 describes JML and how it is used in specifications. In Sections 4 and 5 we present some approaches to writing a specification. Section 4 focuses on the relation between assertion expression and feature interaction. Section 5 focuses on the time of evaluation of assertion expressions. Finally, Section 6 discusses related work and concludes.

2. H-Omega, a dynamic Home control gateway

In [5,13], the authors investigated how to implement home context applications on a service platform. Domain experts have established that dynamism and heterogeneity are unsolved issues in home-control applications. To tackle these problems, the authors have proposed to use service-oriented computing concepts to implement home-control applications.

H-Omega is a home application server. It is a framework that transparently manages non-functional properties at run time. The server manages issues related to dynamic deployment, heterogeneity and dynamism. It also provides a message oriented middleware for asynchronous communication, offers application management facilities, embeds a lightweight database to support persistence and provides a scheduling service managing delayed or periodical actions. All these technical services are published inside the server service registry. This relieves application developers from the responsibilities for non-functional properties, allowing them to concentrate on the application business code.

The platform provides a uniform view of the computing environment: devices and remote facilities are all seen as services. Integrated services (also called applications in

this context) use these devices and remote services as local services. Technically, each service is executed inside a container that manages interactions with all the other services. Applications describe their requirements and then their container injects required objects inside the business logic, or notifies the application.

The framework manages bindings between services, the dynamism and the interactions with technical services. This allows a clear separation between the business logic contained in the code (sometimes called POJO for *Plain Old Java Object*) and the non functional requirements. All non functional requirements are externalized and managed by the container through handlers (see Fig. 1). The resulting application code is as simple as possible. The H-Omega server gateway is implemented on top of OSGiTM[26] and iPOJO [14]. The iPOJO component model has been extended to support home-context application requirement. It provides a uniform view of the computing environment: devices, remote facilities and integrated services are all seen as services, that require and provide services.

2.1. Running Example

Let us consider an integrated service, called *WakeUp*. Informally, when the service is activated, it opens the shutter of the bedroom, increases expected temperature of the bedroom and the bathroom heaters, turns on the lights of the bedroom and displays “Good Morning” on LCDs available in the bedroom. To behave correctly, the *WakeUp* service *requires* four types of devices: dimmer, shutter, heater and LCD. They are considered as services (see Fig. 2). Two of these services are mandatory (dimmer and heater). The other two are optional. We present a brief description of the interfaces offered in the devices below.

The devices can be switched-on or switched-off (`void turnOn()` and `void turnOff()`). It is also possible to checked if they are currently on or off (with `boolean isOn()`).

The *Dimmer* device is a simple lamp with a dimmer. Its intensity ranges from 0 to 100 where 0 is OFF, and 100 is full intensity. It is possible to set the intensity of the light (`void setLevel(int level)`), or get the current intensity (`int getLevel()`).

It is possible to open the *shutter* (for window) from 0 to 100 % (`void setLevel(int level)`) and get the current shutter level (`int getLevel()`). It’s also possible to close or open the shutter (`void open()` and `void close()`). Closing the shutter is equivalent to setting the level to 0%.

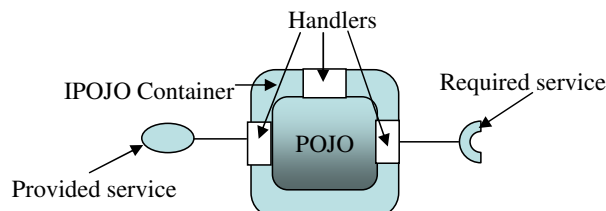


Figure 1. Component in iPOJO

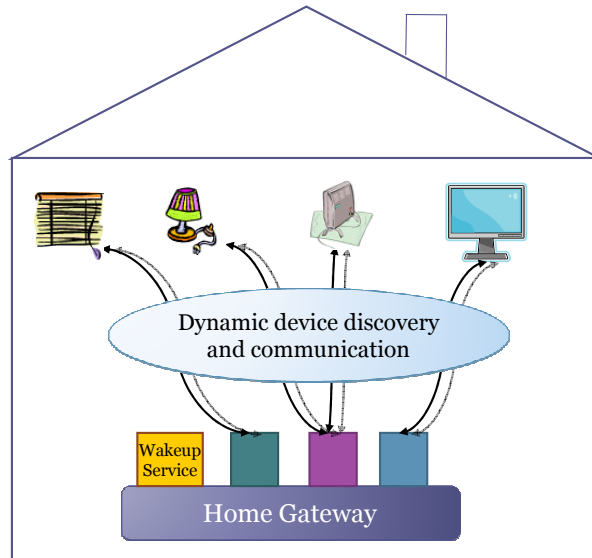


Figure 2. WakeUp service

It is possible to set the desired temperature of a room using the *heater* (`void setTargetedTemperature(int temp)`) and to get the desired temperature of the room (`int getTargetedTemperature()`).

The *LCD* device can display strings (`void display(String message)`). `String getDisplayedMessage()` allows to check what message is currently displayed. This method returns the empty string if there is no message or if the LCD is not switched-on.

A simple implementation of the WakeUp service in Java is given in Fig. 3. The method `execute()` is called to invoke the service. The attributes `m_dimmers`, `m_shutters`, `m_heaters` and `m_lcd` are arrays that correspond to the instances of required services (devices) available in the home. In this simple POJO version, the creator method `WakeUp` is used to initialize those attributes.

When the program is transformed into a iPOJO component, the initialisation and update of the available required services is done automatically. A XML file is used to make the relation between the attributes in the Java files and the services available in the iPOJO environment. In Fig. 4, the association between attributes and required services are done in lines 8 to 13; the filter part allows the selection of the appliances with respect to their location. The declaration of what is provided by WakeUp is declared between lines 4 and 7, in Fig. 4.

3. Specification of the service

In order to develop a reliable well validated system, we decided to provide a formal specification; so that the validation can be performed in a rigorous and formal way using proof tools and/or specification-based testing techniques

```

public class WakeUp implements ExtemporalScenario {

    private String m_name;
    private Dimmer[] m_dimmers;
    private Shutter[] m_shutters;
    private Heater[] m_heaters;
    private LCD[] m_lcd;

    public synchronized void execute() {

        for(int i = 0; i < m_heaters.length; i++) {
            m_heaters[i].setTargetedTemperature(23); }

        for(int i = 0; i < m_dimmers.length; i++) {
            (m_dimmers[i]).turnOn(); }

        for(int i = 0; i < m_shutters.length; i++) {
            m_shutters[i].open(); }

        for(int i = 0; i < m_lcd.length; i++) {
            m_lcd[i].display("Good morning"); }
    }

    public WakeUp(String m_name, Dimmer[] m_dimmers, Shutter[] m_shutters,
        Heater[] m_heaters, LCD[] m_lcd) {
        super();
        this.m_name = m_name;
        this.m_dimmers = m_dimmers;
        this.m_shutters = m_shutters;
        this.m_heaters = m_heaters;
        this.m_lcd = m_lcd;
    }
}

```

Figure 3. A part of a Java program for WakeUp service

Our specification approach relies on a Design by Contract strategy [24]. Several formal specification languages are associated with Java, such as Jcontractor² [17,16], Jass³ [2], or JML⁴ [21]. For all of them, it is possible to express formal properties and requirements on the classes and their methods. We picked JML (Java Modelling Language) since it has well-established semantics and since it is compatible with several tools that support runtime assertion checking, extended static checking, or verification [6]. Moreover, it has been successfully used for both off-line and on-line home service validation in [31,10]. In the following, we first briefly present JML and illustrate its use with WakeUp.

3.1. JML: Java Modelling Language

The Java Modelling Language (JML) is an annotation language used to specify Java programs by expressing formal properties and requirements on the classes and their methods [21].

²<http://jcontractor.sourceforge.net/>

³<http://csd.informatik.uni-oldenburg.de/~jass/>

⁴<http://www.cs.ucf.edu/~leavens/JML/>

```

1 <iPOJO>
  <Component className="homega.impl.scene.wakeup.Wakeup"
              architecture="true">
    <Provides>
5      <property name="friendly.name" field="m_name" value="Wakeup" />
      <property name="name" type="java.lang.String" value="wakeup" />
    </Provides>
    <requires filter="(location=bedroom)" field="m_dimmers" />
    <requires filter="(location=bedroom)" field="m_shutters"
10              optional="true" />
    <requires filter="(location=bedroom)" field="m_lcd" />
    <requires filter="(|(location=bedroom)(location=bathroom))"
              field="m_heaters" optional="true" />
    </Component>
15 <instance name="Wakeup" component="homega.impl.scene.wakeup.Wakeup">
  <property name="name" value="Wakeup Instance" />
  </instance>
</iPOJO>

```

Figure 4. XML file to configure the WakeUp service in the iPOJO environment

The JML specification appears within special Java comments, between `/*@` and `@*/` or starting with `//@`. The specification of each method precedes its interface declaration. JML annotations rely on three kinds of assertions: class invariants, pre-conditions and post-conditions. Invariants have to hold in all visible states. A visible state roughly corresponds to the initial and final states of any method invocation. JML relies on the principles of Design by Contract defined by B. Meyer [24], which states that to invoke a method, the system must satisfy the method pre-condition, and as a counterpart, the method has to establish its post-conditions. A method’s precondition is given by the *requires* clause. If that is not true, then the method is under no obligation to fulfill the rest of the specified behavior.

JML extends the Java syntax with several keywords. `\result` denotes the return value of the method. It can only be used in *ensures* clauses of a non-void method. `\old(Expr)` refers to the value that the expression `Expr` had in the initial state of a method. `\forall` and `\exists` designate universal and existential quantifiers. The JML compiler (`jmlc`) translates the annotated Java code into instrumented bytecode to check that the Java program respects the specification during the execution.

3.2. A simple specification of WakeUp in JML

Informally, as described previously, the WakeUp integrated service is expected to switch-on all the dimmers in the bedroom, set the targeted temperature of the bedroom and bathroom heaters to 23°C, open the shutters of the bedroom (if they are any), and to display the message “Good Morning” of the bedroom LCDs (if they are any). A simple description of the expected properties in JML are as follows:

```

// Dimmer properties D1 and D2
//@ ensures (\forall int i; 0<=i && i<m_dimmers.length;
//@           m_dimmers[i].isOn());
//@ ensures (\forall int i; 0<=i && i<m_dimmers.length;
//@           m_dimmers[i].getLevel(>0);

```

```

// Heater properties H1 and H2
//@ ensures (\forall int i; 0<=i && i<m_heaters.length;
//@          m_heaters[i].isOn());
//@ ensures (\forall int i; 0<=i && i<m_heaters.length;
//@          m_heaters[i].getTargetedTemperature() == 23);

// Shutter property S1
//@ ensures (\forall int i; 0<=i && i<m_shutters.length;
           m_shutters[i].getLevel()==0);

// LCD properties L1 and L2
//@ ensures (\forall int i; 0<=i && i<m_lcd.length; m_lcd[i].isOn());
//@ ensures (\forall int i; 0<=i && i<m_lcd.length;
           m_lcd[i].getDisplayMessage().equals("Good morning"));

```

The JML specification here takes advantage of the fact that the set of appliances is automatically updated by the iPOJO environment. Thus, `m_dimmers` always represents the set of dimmers available in the bedroom.

It is interesting to note that the initial implementation of `WakeUp` (given in Fig. 3) expected the call to `setTargetedTemperature` method to automatically switch on the heaters used in the service. However, this was not the case. The heaters were not automatically switched on when the new target temperature was set, resulting in a JML assertion violation. This implementation error, however, is easily rectified by ensuring that at the start of the wake up service, all the devices are systematically switched on.

4. Influence of the specification on feature interactions

This section is the first part of our discussion on how to write specifications. It is dedicated to the illustration of how specification design can influence interaction between two services.

Specification used for on-line verification needs to be general enough to support changes. Devices may evolve or change and the specification should be able to accommodate these evolutions.

Notice that in the specification proposed previously for `WakeUp`, the specification is restricted to the usage of dimmers. The user requirements can also be achieved if we were to use “binary lamps” in place of dimmers. However, the proposed specification would not allow that since the expected result is given with respect to the state of dimmers. Using the state of devices in the specification restricts the type of devices that can be used in the service.

To illustrate another issue with the aforementioned specification, consider another integrated service for energy saving: *SaveLight*. It switches-off the lamps and dimmers if “enough” light is provided from outside (the shutters are open and sun-light is sufficient). If *SaveLight* is executed in concurrency with `WakeUp`, dimmers might-be switched-off before the end of the `WakeUp` service execution. The `WakeUp` assertions related to the dimmers might thus be violated, although there is some light now in the bedroom. The property violation may also occur with respect to the *SaveLight* service when the dimmers and lamps remain switched-on despite enough light from outside.

SaveLight and WakeUp services are said to interact “badly” because the expected properties of the services can potentially be violated when both services run concurrently. Nevertheless, notice that if the lamps are switched off because the light level is enough, the user expectations for both services are satisfied. In short, the WakeUp service is mainly used to increase the brightness of the bedroom and to obtain a temperature of 23°C or greater. From the SaveLight point of view, the expectation is to save light from lamps and dimmers by using natural light (sunlight) when possible (if the light intensity “sufficient”).

With respect to these two observations, we investigated whether it is possible to provide a specification that would

- support easy evolutions in the type of devices, and
- allow interaction to remain “undeclared” if the user expectations are satisfied.

Expressing specification using sensors

The user expectations about WakeUp mainly concern the brightness and the temperature, in the bedroom and in the bathroom. At the end of service execution, the brightness should be greater than a required level and the temperature should be greater than 23°C.

We explore a second solution, where the brightness intensity and the temperature are evaluated using sensors such as illuminometers and thermometers. In this situation, the specification of the WakeUp service can be re-written as:

```
// D2-a
//@ ensures (\forall int i; 0<=i && i<be_illuminometer.length;
//@           be_illuminometer[i].getLevel() > 50);
// H2-a
//@ ensures (\forall int i; 0<=i && i<ba_thermometer.length;
//@         ba_thermometer [i].getTemperature() >=23);
```

where `be_illuminometer` and `ba_thermometer` denote respectively the set of illuminometers in the bedroom and the set of thermometers in the bathroom.

Expressing specification using sensors presents two advantages. First, the service specification is expressed as close as possible to the user expectations. So, if another service such as SaveLight modifies the state of the shutters, the dimmers or the heaters, an interaction is detected only when the user expectations are truly violated. Artificial interactions such as the ones introduced by using state of devices in the specification are not detected in this approach. A second advantage is that the specification is more general than the previous approach, in the sense that no adaptation is required to support other types of appliances. Therefore, if dimmers were to be replaced by binary lamps and heaters by air-conditioners, the specification need not be modified.

Nevertheless, this second approach to expressing specification presents some disadvantages. First, the approach introduces a service dependency on the availability of sensors. For instance, if the illuminometer or thermometer is unavailable, the WakeUp service will not be deployed. This is undesirable since those sensors are simply used for monitoring the service and should not interfere with the state of the service. Additionally, the specification is now dependant on the type of sensors (here illuminometers and thermometers) although we wanted to be independent from the type of devices.

Expressing specification using a model of the environment

We explore a third solution, which consists of expressing the requirements with respect to a *model* of the home environment, as it is done in [31]. Let us introduce an object “environment” that captures the temperature and the brightness in each room. In this situation, the specification of the WakeUp service can be re-written as:

```
// D2-b
//@ ensures env.getBrightness("bedroom") > 50;
// H2-b
//@ ensures env.getTemperature("bathroom") >=23;
```

The idea is to use this environment object as a model of the home. This model can be updated in one of two ways. Sensors, if available, can be used to update the environment. If sensors are not available, a model of evolution can be implemented: if an appliance is solicited, the expected result is computed using the model. This solution presents several advantages. The specification is close to the user requirements. As discussed previously, this reduces the number of situations where interactions are detected owing to the absence of devices in the specification. Additionally, this solution using a model of the environment can be deployed without using sensors, thereby removing the dependency on sensor availability.

Unfortunately, this solution also presents some drawbacks. First, a model has to be provided. For off-line detection, the model can be simple, since it aims at detecting potential interactions. However, for on-line detection, if the model is too simple or too abstract, there are high risks that its evolution will deviate from reality. Building an adequate model for on-line detection is thus a challenging task. Moreover, it has to be configured specifically for each house, and may require lots of computations. For instance, a model of temperature evolution in the house can be viewed as a simple or complex model. A simple model for instance could state that the temperature increases uniformly as soon as the heaters are switched on. A complex model, on the other hand, could take into account several factors for computing the temperature once the heaters are switched on such as the position of the heaters, the configuration of the rooms in the house, the surroundings, the insulation, the weather, etc. Moreover, there is always the risk that the real environment evolves differently from the model if the appliances were to be used independently of the HNS services. For instance, the TV or the lamps can be switched on or off directly with the remote-controller or the switch button.

In summary, during our case study we noticed the following points. Service specifications are naturally expressed with respect to the states of the appliances that are controlled by the services. However, specifications expressed in this manner may artificially increase the number of situations where services are considered to badly interact. In order to limit these situations, we propose two alternate ways of expressing the specification. One alternate way is to use sensors to express the state of devices in the service specification. Specifications expressed using sensors provide a level of abstraction, in that the specifications do not involve the type or state of the device directly. Therefore, modifications in the device used, assuming the sensors remain unchanged, will not affect the specification. This method of expressing specifications is thus closer to the user expectation, and is independent from the devices. Nevertheless, this alternate way has

a drawback in that it introduces a dependency on sensor type and availability. Specification expressed using sensors also have the drawback of potentially triggering bad interactions. The reason being sensors, like devices, can be jointly used by two or more services causing unwanted interactions. To avoid the problems in the first alternate way, we proposed another approach that uses a model of the home environment evolution to express specifications. Specifications expressed using such a model will not involve the state and type of devices directly. The specifications are instead expressed using the state of the environment. This approach is closer to user expectations since only the environment state is usually visible to the user. To compute and update environment state, the model could either use sensors, or alternately it could use a model of evolution of the environment. Thus, in this approach, specifications are not dependent on sensors as long as there is a model of environment evolution. Unwanted interactions and dependencies introduced by sensors or devices will not be encountered in this approach. Nevertheless, as mentioned earlier, it is important to keep in mind that creating a model of environment evolution is not easy. Additionally, ensuring that this model of evolution is the same or close to the real environment is a difficult, if not impossible, task.

We have thus far presented three ways of expressing service specifications, their respective advantages and disadvantages. We do not believe any of these approaches is a silver arrow for expressing specifications. The user has to carefully choose and evaluate each of these approaches according to his/her application. The goal of this paper is to enlighten the user in the available approaches for specifying services, and in the pros and cons of each of those approaches. We believe the choice of approach for specifying services is highly dependent on the application, and therefore we do not categorically recommend any one approach.

5. Satisfiability of specification

During our case study, we expressed the specification as assertions that are evaluated at the end of the execution of the service (post-conditions). This section outlines some approaches to expressing specifications with respect to the instant they are evaluated.

For the WakeUp implementation (given Fig. 3), when the service is activated, it executes a finite sequence of actions and then stops. This is a problem for the evaluation of several post-conditions, such as D2-a, H2-a or L2. The problem arises because several devices used in the wake up service do not produce instantaneous output (output that is observed at the same time as the input). Instead, the device uses non zero amount of time to produce output corresponding to a particular input. For instance, when the LCD is powered on, the display is not turned on instantaneously. The string to be displayed may appear after a one second delay (for example). Property L2 may therefore be evaluated to false at the end of WakeUp service activation, because the LCD needs time to be powered on and may not display “Good Morning” instantaneously. A similar problem can be illustrated with the dimmer. The dimmer may require time to reach the required light intensity (especially for low-consumption lamps). Therefore, D2-a property may be evaluated to false at the end of the WakeUp service activation.

This problem is more evident in the use of heaters. Consider property H2 where the temperature is evaluated just after the heaters are switched-on (second and third specifications). The temperature, however, will not reach 23°C immediately: some delay is re-

quired because of the thermal inertia. To accommodate this non-zero delay, the requirements will have to be expressed as a liveness property: the temperature will *eventually* reach 23°C. It is not possible to set a bounded time limit on this liveness property.

These examples raise two different issues. The first one is related to the fact that electric devices often need non-zero time to modify their states. Usually this is a short period of time that can be bounded. The second issue is related to properties in the environment that cannot be changed instantaneously (temperature, humidity, air quality, etc.). Requirements about environment attributes that do not change instantaneously and that cannot be bounded by time can be expressed as liveness properties.

In the following, we distinguish two types of services. We call *instantaneous* services, services that execute a finite sequence of actions and then stop. *Monitoring* services, on the contrary, monitor the evolution of the environment. The issues mentioned previously are mainly related to the instantaneous services.

To address the first issue, we recommend delaying the end of an instantaneous service execution. The delay would allow the change, if any, to be reflected in the environment attributes. The delay should be long enough to allow the appliances to be ready. However, care should be taken that the delay is not longer than required, since otherwise, unforeseen property violations may occur. For instance, in the WakeUp service, after the service is activated, the user may decide to leave the bedroom and switch off the dimmers five minutes after waking up. Thus if the delay at the end of wake up service is longer than five minutes, the service post-condition stating that the dimmers should be turned on, will be violated.

The second issue, expressing liveness properties as post-conditions for an instantaneous service is contradictory by nature. An instantaneous service is supposed to terminate immediately or in bounded time after its activation. Liveness properties by their definition are not bounded in time. Thus, the statement is a contradiction in itself. As a result, we recommend associating liveness properties to monitoring services rather than instantaneous services. If instantaneous services need to be associated with liveness properties, as in the case of the heater in the wake up service, then we recommend creating a monitoring service for the wake-up service that can then be associated with the liveness property. For instance, a monitoring version for the WakeUp service could be activated a moment before the WakeUp time. The service could check the temperature. According to the observations, if the observed temperature is lower than expected, the service can switch on the heaters and monitor that the temperature progressively increases.

6. Related Work and Conclusion

We believe the feature interaction problem in the context of home automation is important and can have significant consequences in the safety and security of such systems. The feature interaction problem encountered in this context is different from the problem encountered in telecommunications. One difference lies in the fact that the HNS does not completely control all the aspects in the home. Moreover, interaction among services can be caused by the home environment. These types of interactions are sometimes called implicit interactions.

Several approaches have been proposed for the home. In [23], Andreas Metzger and Christian Webel have proposed a solution for interaction detection based on the require-

ment document. It relies on the PROBA_{ND} requirement engineering method. The detection solution is an off-line solution, which requires identifying the implicit interactions. In [25], Masahide Nakamura *et al* also proposed a off-line solution based on analysis of pre and post conditions associated to the methods exercised by the services. In [32], Ben Yan adopts an approach based on pre and post-conditions that can be used for both off-line and on-line detection. In [30], Michael Wilson *et al.* proposed an online detection approach to prevent bad interaction to occur while allowing positive ones. This approach considers both incompatible use of devices and environment analysis.

In this article, we do not propose a new method to detect interactions. We study the problem of interaction, and more generally the problem of service validation, in the context of specification by assertions. In particular, we focus on the impact of assertion expressions on interaction and validation. Different points of view can be adopted to express assertions. One may specify only the expected state of the controlled devices at the end of the service activation. One may express the expected requirement with respect to sensors. Finally, one may express assertions with respect to a model of the environment. In our case study, we found that using a specification dealing with only devices can signal interactions between services even if user expectations for the services are not violated. Specifications that use sensors in place of devices reduce the number of false alarms with respect to feature interaction and only signal interaction in situations where user expectations are also violated.

During our case study, we also noticed that user expectations can be expressed as liveness properties. Presently, post-conditions do not seem well adapted to expressing such types of properties, especially for services that terminate immediately after their activation. Further research is needed to investigate a suitable way of expressing liveness properties, particularly for services that are used to monitor the home.

References

- [1] Daniel Amyot and Luigi Logrippo, editors. *Feature Interactions in Telecommunications and Software Systems VII, June 11-13, 2003, Ottawa, Canada*. IOS Press, 2003.
- [2] Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - Java with Assertions. *Electr. Notes Theor. Comput. Sci.*, 55(2), 2001.
- [3] J. Bohn, V. Coroama, M. Langheinrich, F. Mattern, and M. Rohs. Social, Economic, and Ethical Implications of Ambient Intelligence and Ubiquitous Computing. In W. Weber, J. Rabaey, and E. Aarts, editors, *Ambient Intelligence*, pages 5–29. Springer-Verlag, 2005.
- [4] Wiet Bouma and Hugo Velthuisen, editors. *Feature Interactions in Telecommunications Systems, May 8-10, 1994, Amsterdam, The Netherlands*. IOS Press, 1994.
- [5] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin. A dynamic-soa home control gateway. In *IEEE International Conference on Services Computing (SCC 2006)*, pages 463–470. Chicago, Illinois, USA, September 2006. IEEE Computer Society.
- [6] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.
- [7] Muffy Calder and Evan H. Magill, editors. *Feature Interactions in Telecommunications and Software Systems VI, May 17-19, 2000, Glasgow, Scotland, UK*. IOS Press, 2000.
- [8] Kong E. Cheng and Tadashi Ohta, editors. *Feature Interactions in Telecommunications III, October 11-13, 1995, Kyoto, Japan*. IOS Press, 1995.
- [9] Petre Dini, Raouf Boutaba, and Luigi Logrippo, editors. *Feature Interactions in Telecommunications Networks IV, June 17-19, 1997, Montréal, Canada*. IOS Press, 1997.

- [10] L. du Bousquet, M. Nakamura, B. Yan, and H. Igaki. Using formal methods to increase confidence in one home network system implementation. case study. In *workshop on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA 2007)*, Poitiers, France, december 2007.
- [11] L. du Bousquet, F. Ouabdesselam, J.-L. Richier, and N. Zuanon. Feature interaction detection using synchronous approach and testing. *Computer Networks and ISDN Systems*, 11(4):419–446, 2000.
- [12] Lydie du Bousquet and Jean-Luc Richier, editors. *Feature Interactions in Software and Communication Systems IX, International Conference on Feature Interactions in Software and Communication Systems, ICFI 2007, 3-5 September 2007, Grenoble, France*. IOS Press, 2007.
- [13] C. Escoffier, J. Bourcier, P. Lalanda, and Jianqi Yu. Towards a home application server. In *5th IEEE Consumer Communications and Networking Conference (CCNC 2008)*, pages 321–325, January 2008.
- [14] C. Escoffier, R.S. Hall, and P. Lalanda. ipoj: an extensible service-oriented component framework. In *IEEE International Conference on Services Computing (SCC 2007)*, pages 474–481, July 2007.
- [15] N. D. Griffeth and Y.-J. Lin. Extending telecommunication systems: The feature-interaction problem. *IEEE Computer*, pages 14–18, August 1993.
- [16] M. Karaorman and P. Abercrombie. jContractor: Introducing Design-by-Contract to Java Using Reflective Bytecode Instrumentation. *Formal Methods in System Design*, 27(3):275–312, 2005.
- [17] M. Karaorman, U. Holzle, and J. Bruno. jContractor: A Reflective Java Library to Support Design by Contract. Technical report, University of California at Santa Barbara, Santa Barbara, CA, USA, 1999.
- [18] Kristofer Kimbler and Wiet Bouma, editors. *Feature Interactions in Telecommunications and Software Systems V, September 29 - October 1, 1998, Malmö, Sweden*. IOS Press, 1998.
- [19] M. Kolberg, E.H. Magill, and M. Wilson. Compatibility issues between services supporting networked appliances. *IEEE Communications Magazine*, 41:136–147, Nov. 2003.
- [20] Marc Langheinrich, Vlad Coroama, Jürgen Bohn, and Friedemann Mattern. Living in a Smart Environment – Implications for the Coming Ubiquitous Information Society. *Telecommunications Review*, 15(1):132–143, February 2005.
- [21] G.T. Leavens, A.L. Baker, and C. Ruby. JML: A Notation for Detailed Design. In H. Kilov, B. Rumpe, and I. Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer, 1999.
- [22] Ltd. Matsushita Electric Industrial Co. Kurashi net (jp).
- [23] Andreas Metzger and Christian Webel. Feature interaction detection in building control systems by means of a formal product model. In *Feature Interactions in Telecommunications and Software Systems VII (ICFI)*, pages 105–122. IOS Press, June 2003.
- [24] B. Meyer. Applying “Design by Contract”. *Computer*, 25(10):40–51, October 1992.
- [25] Masahide Nakamura, Hiroshi Igaki, and Ken ichi Matsumoto. Feature interactions in integrated services of networked home appliances: An object-oriented approach. In *Feature Interactions in Telecommunications and Software Systems VIII, (ICFI’05)*, pages 236–251, Leicester, UK, June 2005. IOS Press.
- [26] OSGi Alliance OSGi Alliance. *OSGi Service Platform: Release 3, March 2003*. IOS Press, 2003.
- [27] Stephan Reiff-Marganiec and Mark Ryan, editors. *Feature Interactions in Telecommunications and Software Systems VIII, ICFI’05, 28-30 June 2005, Leicester, UK*. IOS Press, 2005.
- [28] TOSHIBA. Toshiba home network: Feminity.
- [29] M. Weiser. Some Computer Science Issues in Ubiquitous Computing. *Com. of the ACM*, 36(7):74–84, 1993.
- [30] Michael Wilson, Mario Kolberg, and Evan H. Magill. Considering side effects in service interactions in home automation - an online approach. In *Feature Interactions in Software and Communication Systems IX, (ICFI’07)*, pages 172–187, September, Grenoble, France 2007. IOS Press.
- [31] B. Yan, M. Nakamura, L. du Bousquet, and K.-I. Matsumoto. Validating safety for integrated services of home network system using jml. *Journal of Information Processing (IPSJ)*, 16:38–49, 2008.
- [32] Ben Yan. Considering safety and feature interactions for integrated services of home network system. In *Feature Interactions in Software and Communication Systems IX, (ICFI’07)*, pages 199–202, September, Grenoble, France. IOS Press.