Innovations Syst Softw Eng (2009) 5:181–196
DOI 10.1007/s11334-009-0092-5

ORIGINAL PAPER

Using formal methods to increase confidence in a home network system implementation: a case study

Lydie du Bousquet · Masahide Nakamura · Ben Yan · Hiroshi Igaki

Received: 11 March 2009 / Accepted: 5 June 2009 / Published online: 20 June 2009

Abstract A home network system consists of multiple networked appliances, intended to provide more convenient and comfortable living for home users. Before being deployed, one has to guarantee the correctness, the safety, and the security of the system. Here, we present the approach chosen to validate the Java implementation of a home network system. We rely on the Java Modelling Language to formally specify and validate an abstraction of the system.

Keywords Formal methods · Validation by testing · Home network system

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 are currently a hot topic in the area of ubiquitous/pervasive computing [5,26,41]. Several HNS products are already on the market (e.g. [30,40]).

A 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 we call HNS *integrated services* [24]. For instance, orchestrating a TV, a DVD player, 5.1ch speakers, lights, curtains and an air-conditioner implements an integrated service, called *DVD theater service*, where a user can watch movies in a theater-like atmosphere.

For practical use of such services, it is essential to guarantee the correctness, the safety and the security of the services. A service should behave as specified (functional correctness). It must be free from the conditions that can cause injury or death to users, damage to or loss of equipment or environment (safety). And it must be protected against malicious intrusions or hijacking the service (security). For instance, a *RemoteLock* service (that checks and locks doors and windows even from outside the home) must be disabled in case of a fire; otherwise a user might be locked into the room.

In this article, we present the approach we used to specify and then to validate a set of HNS *integrated services* that have been developed by Nakamura et al. [34,35]. Our approach relies on a Design by Contract strategy [31,32]. The Java Modelling Language (JML) [21], an executable specification language, is used for both off-line and on-line validation. Sections 2–8 are dedicated to the presentation of the approach. Section 9 describes some related works. Section 10 concludes on the lessons learnt and draws some perspectives.

This is a substantially revised version of our paper that appeared in the proceedings of the Workshop On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), Poitiers-Futuroscope, France, December 2007.

L. du Bousquet (✉)
Laboratoire d'Informatique de Grenoble (LIG),
Universités de Grenoble (UJF), BP 72,
38402 Saint Martin d'Hères cedex, France
e-mail: lydie.du-bousquet@imag.fr

M. Nakamura · H. Igaki
Graduate School of Engineering Science,
Kobe University, Kobe, Japan
e-mail: masa-n@cs.kobe-u.ac.jp

H. Igaki
e-mail: igaki@cs.kobe-u.ac.jp

B. Yan
Graduate School of Information Science,
Nara Institute of Science and Technology, Nara, Japan
e-mail: hon-e@is.naist.jp

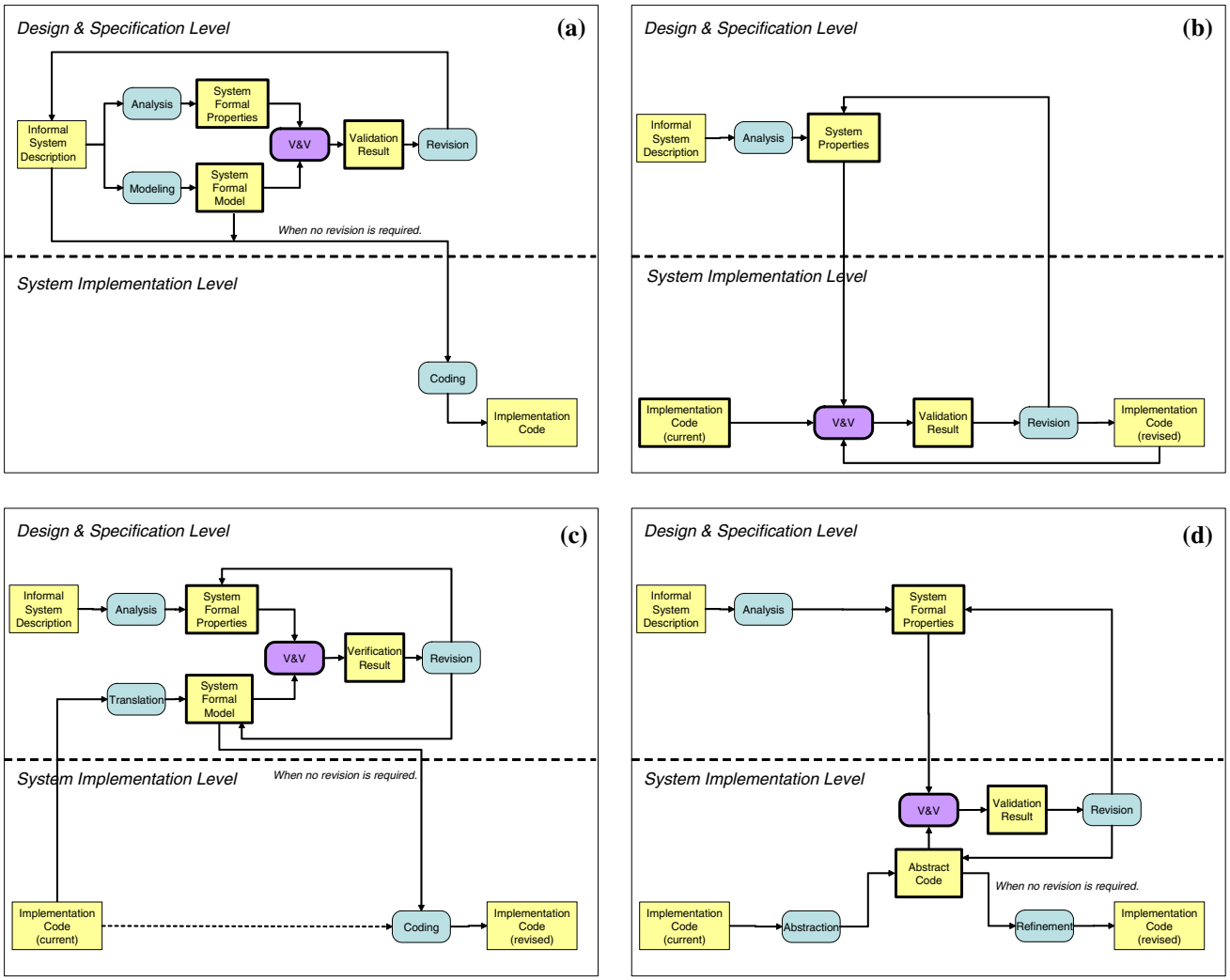

Fig. 1 Different approaches for formal method application. **a** Classical life cycle, **b** direct proof of the code, **c** possible approach, **d** approach followed

2 Motivations and chosen approach

The use of formal methods in the development of computing systems promises better quality in general, and in particular safer and more reliable systems [20]. Formal methods are mathematically based, and thus can provide several benefits. First, they define "what is meant for a design to be correct with respect to its definition". It also allows some properties to be deduced by a process of mechanical logical deduction [39]. This article is a case study of the use of formal methods to demonstrate that the current implementation of our HNS services is correct and safe.

Figure 1a illustrates how formal methods are classically applied during the development. First, the properties and a formal model from the system specifications are derived from the informal description. The model is validated/verified considering the properties with the help of several kinds of approaches, such as tests, static analysis, model-checking, decision procedures... [39]. Several revision steps may be required before being satisfied with the model. Once it is done, an implementation is derived (possibly automatically). Some additional validation steps may be required to demonstrate that the implementation conforms to the model.

In our case, the implementation was available at the beginning of our validation process. The ideal situation would have been to apply directly formal methods to this implementation as illustrated in Fig. 1b. Formal properties would have been derived from the informal description and then compared directly to the real code. Revisions would have been made directly on the code. This approach is possible for certain type of applications and programming languages, especially with model-checking. In our case, the implementation deals with a large number of technical libraries. Before being able to prove the core of the system (i.e. the services), one would

have to prove all these libraries. Another approach has then to be chosen.

At this step, one possible approach would have been to translate the core of our implementation into a formal language on which validation and verification tools could have been applied (cf. Fig. 1c). Once this model verified, a second translation step would have been required in order to modify our implementation. This approach was not selected since there was a high-risk of introducing faults during the two translations phases, which may have led to misleading conclusions.[1]

Instead of this approach, we prefer to keep with the implementation language (Java) and try to focus the validation on an abstraction of the implementation (cf. Fig. 1d). The expected properties were expressed in an annotation language for Java. Several formal specification languages are associated with Java, such as Jcontractor[2] [22,23], Jass[3] [3], or JML[4] [27]. For all of them, it is possible to express formal properties and requirements on the classes and their methods. We have chosen JML since it has a well-established semantics and since it is recognized by a dozen tools, which support runtime assertion checking, extended static checking, or verification [7].

To validate and improve our HNS implementation, we have carried out the following steps.

1. The core of the application was identified and some abstractions were carried out. Subsequently, we call the result of this phase *abstract implementation*. The real and the abstract implementations are described in Sects. 3 and 4.
2. The expected properties of the services were deduced from the informal description. The abstract implementation was then annotated with JML. This part of the work is detailed Sect. 5.
3. Several validation steps were carried out. They have two complementary goals: (1) to detect inconsistencies in the abstract implementation and to improve correction/robustness of the services, (2) to improve, to detail or to correct the JML annotations. Several validation approaches were carried out: test (Sect. 6), static analysis and deductive proof (Sect. 7).
4. During the validation, the abstract implementation was modified. A refinement phase was then carried out to obtain an updated real implementation. Since the final implementation was not proven, the JML annotations were left in the new real implementation code in order to monitor the execution (Sect. 8).

[1] This has been observed in [16].

[2] http://jcontractor.sourceforge.net/.

[3] http://csd.informatik.uni-oldenburg.de/~jass/.

[4] http://www.cs.ucf.edu/~leavens/JML/.

3 The home network system application

3.1 Introduction to home network system (HNS)

A HNS consists of one or more networked appliances connected to LAN at home. A networked appliance is usually equipped with smart embedded devices, including a network interface, a processor and storage. Each networked appliance has a set of control APIs, so that the user or 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.

One of the major HNS applications is the integration of networked home appliances in order to provide services (called *integrated services* bellow). An integrated service orchestrates different home appliances via network in order to provide more comfortable and convenient living for the users. For instance, the *DVD Theater Service* turns on a DVD player, switches off the lights, selects 5.1ch speakers and adjusts the volume automatically.

The *Relax Service* is another example. It integrates a DVD player, a sound system, a light, an air-conditioner, and an electric kettle. When the service is started, the DVD player is turned on with a 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, and the kettle is turned on to a boiling mode to prepare hot water for tea.

3.2 A framework for implementing HNS integrated services

As the embedded devices become cheaper, smaller, and as their power consumption is reduced, it is expected in the near future that more appliances will be networked [19]. However, transition to networked appliances is gradual. Most people are still using legacy appliances, which are the conventional non-networked home appliances.

The use of networked appliances has not yet spread for several reasons. Networked appliances are expensive and the selection of available appliances is limited. Due to the interoperability problem, the integration of appliances is often limited, especially in the multi-vendor environment. Moreover, there is a major requirement that the users want to keep using their own legacy appliances since it is not easy for users to immediately replace all the existing legacy appliances with networked ones.

To cope with both the emerging HNS and the legacy appliances, Nakamura et al. have proposed a framework that can deal with legacy appliances with conventional infrared remote controllers [34,35]. The key ideas are (1) to use a programmable infrared remote controller to control the different appliances, and (2) to rely on a service-oriented architecture (SOA) (see [29,36]).

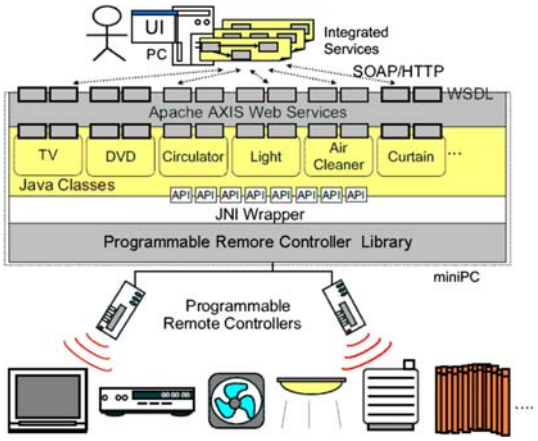

Fig. 2 The architecture the HNS developed by Nakamura et al.

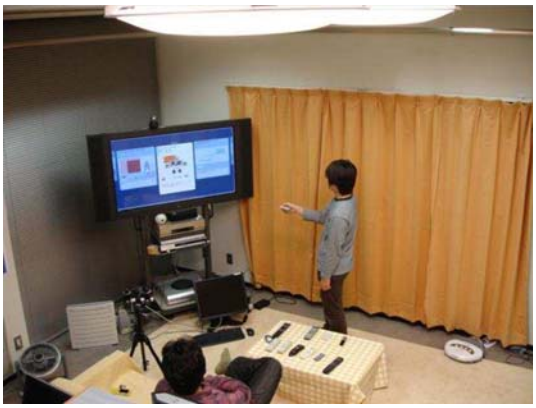

Fig. 3 The showroom of the system presented in Fig. 2

For each appliance, a self-contained component is implemented in Java and deployed as a web service (using Apache AXIS) (Fig. 2). Methods like `On()` and `OFF()` are open interfaces for accessing the basic features of the appliance. They use a set of APIs by which the PC can send infrared signals to the appliances (Ir-APIs). Ir-APIs have been implemented by wrapping the programmable infrared remote controller with a Java Native Interface (JNI Wrapper).

Some HNS applications may need the current status of an appliance to perform an appropriate action. A typical example is an energy-saving service, which stops a DVD player when a TV is turned off. However, the communication between a user and a legacy appliance is basically *one-way* from the remote controller to the appliance. Since it is impossible for the external application to obtain the current status from the legacy appliance, an appliance component has a supplementary feature that stores its *current state* according to the history of the execution.[5] For each appliance component, a `getStatus()` method returns the current state (i.e., the attribute values).

[5] The status is modified with respect to the executed operation.

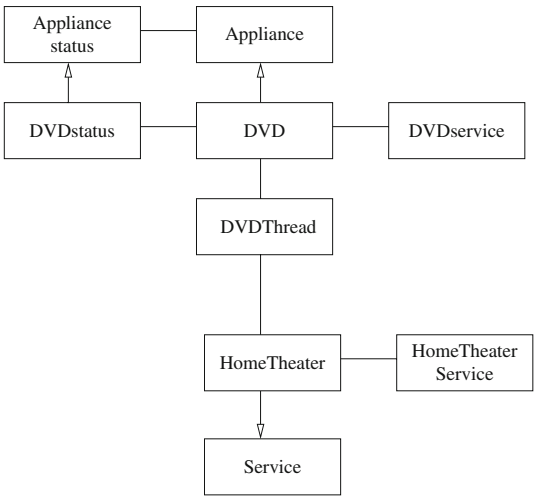

Fig. 4 The class organisation in the real implementation

A HNS is installed in an environment, which can be described as a set of attributes. They include the current energy consumption, the sound level, the temperature, etc. Their values can be obtained via sensors (such as thermometer, fire detector...), which are implemented as web service like other appliance components.

3.3 The real implementation

Figure 3 presents the showroom of Kobe University. The different integrated services can be activated and deactivated thanks to a specific interface.

Figure 4 describes how the implementation classes are organised. In order to simplify, only one appliance (a DVD player) and one service (HomeTheater) are represented in this diagram. Drivers and libraries are not described.

All appliances such as the `DVD` player inherit from the `Appliance` class. All integrated services such as `HomeTheater` inherit from the `Service` class. For each appliance and each integrated service, there is (1) a basic Java class to encode the expected behaviors (resp. `DVD` and `HomeTheater`) and (2) a class that exports the methods for Tomcat/Axis (resp. `DVDService` and `HomeTheaterService`). The status of the appliance is described through a specific class (`ApplianceStatus`) and specialised for each of them (e.g. `DVDstatus`). The integrated services use the appliances through *Threads*.

The showroom of Kobe University deals with nine types of appliances and four integrated services. The core of the application (described in Fig. 4) represents 9000 lines of code, among which 1000 for the `Status` classes, 7000 for appliance classes, threads and associated services (classes like `DVD`, `DVDThread`, and `HomeTheater`) and 1000 for integrated services (classes like `HomeTheaterService` and `DVDService`).

4 Providing an abstract implementation

An application such as our HNS cannot be easily tested or proven. For proof, one problem comes from the complexity of the system. The core of the application (the services) is embedded in a technical framework, composed of drivers and web-service management. For testing, one problem is related to the fact that services may behave with respect to the environment. For instance, the light level might be adjusted with respect to the light intensity measured in the room. To provide relevant tests, one needs to influence the state of the environment. Another point is that the appliances may be damaged if they are too much solicited during a test.

To ease the process of validation, we have then built an abstraction from the real implementation. The aim of this part is to detail how it was done.

4.1 Abstraction of the environment

In the real application, the environment state is captured thanks to sensors that are implemented as appliances. For instance, the current temperature is measured by connected thermometer(s). For the abstract level, we introduce a specific class, called `HomeEnvironment`, to capture the environment state. This class includes several attributes representing, respectively, the temperature, the light intensity, current consumption, the time and so on. It also includes public methods to change the environment state. The environment can be changed either by an appliance when it changes its state (the light intensity of the environment changes when a light is switched-on), or during a test sequence (to initialize the environment, for instance).

A `Home` class was also introduced. It corresponds to a particular configuration of the HNS. From the `Home` interface, it is possible to call any public methods of the integrated services, appliances and environment.

4.2 Abstraction of the appliances

Two main abstractions were carried out for the appliances. A first abstraction consists in replacing part of the code related to the remote controller into simple printing message calls, as it is shown for the `switchOn` method of the `TV` class. The original code is given in Fig. 5 and the abstract one is given in Fig. 6. This abstraction allows you to validate the appliance classes independently from the remote control drivers.

An abstraction of the status was also performed. For the real implementation, the status is encoded with a specific class that is specialized with respect to each type of appliance. The main attribute of the status class is a String. At the abstract level, the status is encoded with a simple attribute of type String. This allows you to validate each appliance class with a limited set of dependencies among the classes.

```
public void switchOn(void) {
   /* Controller and signal objects of Ir-API */
 IrControler con = new IrController();
 IrSignal sig = new IrSignal();

   /* set signal ON for TV_A and send it */
 sig.setSignalType(SWITCH_ON, TV_A);
 con.sendSignal(sig);
 sleep(2);
 state="ON";
}
```

Fig. 5 Real implementation of the method `switchOn()` for TV_A

```
public void switchOn(void) {
  System.out.println("SWITCH_ON, TV_A");
  internalState="ON";
}
```

Fig. 6 Abstraction of the method `switchOn()` for TV_A

```
public void switchOn(void) {
  if (powerState.equals("ON")) {
   setApplianceConsumption(maxConsumption);
   System.out.println("SWITCH_ON, TV_A");
   internalState="ON"; }
}
```

Fig. 7 Evolution of method `switchOn()` for TV_A in the model

The real implementation did not take into account the fact that appliances can be powered on or off. The internal state of appliances and the code of some methods were completed to deal with the power state. In order to synchronize the appliances with the environment, we introduce some code that describes their impact on the environment. For instance, when the TV is switched on, the current consumption is increased in the environment (see Fig. 7).

4.3 Abstraction of the integrated services

The real implementation of the integrated services concerns threads and exceptions. It also uses the "log4j" library in Apache, to log all the events. At the abstraction level, the code of the integrated services deals only with the statements necessary to achieve the expected scenarios. The integrated services are connected to the environment thanks to their inheritance with the Service class. Figure 8 shows the abstraction of the DVD Theater service.

Figure 9 shows the class organisation of the abstract implementation. Appliances still inherit for the `Appliance` class, and integrated services inherited from the `Service` class.

```
public class DVDtheater extends Service{
  private DVD dvd;
  private TV tv;
  private Blind blind;
  private Light light;
  private Speaker speaker;

public DVDtheater(DVD dvd0,TV tv0,
              Blind blind0, Light light0,
              Speaker speaker0){
   sp = "OFF";   //state of the service
   dvd=dvd0;
   tv=tv0;
   blind=blind0;
   light=light0;
   speaker=speaker0;
  }


  void activation() {
    /* TV is switched on */
    tv.powerOn();
    tv.switchOn();
    tv.setSoundIntputMode("DVD");

    /* DVD player is swtiched on */
    dvd.powerOn();

    /* Speakers are switched on */
    speaker.powerOn();
    speaker.setIntputSource("DVD");
    speaker.setApplianceVolume(10);

    /* bind is closed */
    blind.powerOn();
    blind.Close();

    /* brightness is minimised*/
    light.powerOn();
    light.setBrightnessLevel(2);

    /* play the dvd*/
    dvd.Play();
 }


  void deactivation() {
    tv.switchOff();
    tv.powerOff();

    dvd.switchOff();
    dvd.powerOff();

    speaker.switchOff();
    speaker.powerOff();

    light.powerOff();
  }
}
```

Fig. 8 Abstraction of the DVD theatre service

Threads are not introduced and, thus, integrated services directly call the appliance methods. There are no classes for exporting the services as in the real implementation.

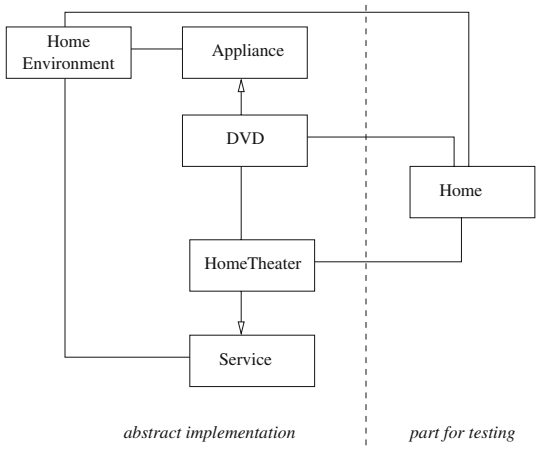

Fig. 9 The class organisation in the abstract implementation

4.4 New appliances and services

The abstract implementation of the real system was also used to implement and validate new appliances and integrated services, not yet present in the Japanese showroom. New appliances such as an electric kettle, a gas valve, or a hot water system were thus added. It was possible to introduce new integrated services such as the *Relax Service* (see Sect. 3.1) or the *Cooking Preparation Service* (that prepares the kitchen for cooking). Our current abstract implementation is composed of 25 classes among which there are 14 appliance components, 7 integrated services, the `HomeEnvironment`, the `Home`, the `Appliance` and the `Service` classes.

5 Specifying the HNS integrated services

Before providing a HNS and integrated services, one must guarantee that the implementation is correct and "safe" for inhabitants, properties and their surrounding environment. This section is dedicated to the expression and formalisation of the requirements.

5.1 Requirements at different levels

In Nakamura et al. [42], have identified three levels of requirements for integrated services.

For every electric appliance, the manufacturer prescribes a set of safety instructions for proper and safe use of the appliance. Traditionally, these instructions have been written for human users. 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: *do not open the lid while the water is boiling, or there is a risk*

of scalding. Any integrated service using the kettle must be implemented so that it will never open the lid while the kettle is in the boiling mode.

Since an integrated service orchestrates different multiple appliances simultaneously, it is also necessary to consider global properties of the multiple appliances. For instance, the Cooking Preparation Service (which automatically sets up the kitchen configuration for preparing for cooking) must avoid carbon monoxide poisoning. *While the gas valve is opened, the ventilator must be turned on.*

In general, each house has a set of residential rules for its inhabitants' and neighbours' safety. Since the integrated services have various impacts on the surrounding environment (including the room, the building, the neighbours, etc), the services must satisfy these rules. For instance, *do not make loud voice or sound after 9 p.m.*

An integrated service is *locally correct* if and only if it satisfies all local properties, i.e. all properties derived from the appliance instructions. It is *globally correct* if and only if it satisfies all properties prescribed for it. It is *environmentally correct* if and only if it satisfies all properties derived from the environment where it is provided.

Let s be a given integrated service, and

- let $App(s) = \{d_1, d_2, \ldots, d_n\}$ be a set of networked appliances used by s,
- let $LocalProp(d_i) = \{lp_{i1}, lp_{i2}, \ldots, lp_{im}\}$ be a set of local properties derived by the instructions of the appliance d_i,
- For a service s, the set of local properties is $LocalProp(s) = \cup_{d_i \in App(s)} LocalProp(d_i)$,
- let $GlobalProp(s) = \{gp_1, gp_2, \ldots, gp_k\}$ be a set of global properties prescribed by s,
- let $EnvProp(s) = \{ep_1, ep_2, \ldots, ep_l\}$ be a set of environment properties derived from the environment where s is provided.

[Local Correctness:] s is *locally correct* if and only if s satisfies all properties in $LocalProp(s)$.
[Global Correctness:] s is *globally correct* if and only if s satisfies all properties in $GlobalProp(s)$.
[Environment Correctness:] s is *environmentally correct* if and only if s satisfies all properties in $EnvProp(s)$.

5.2 Brief description of JML

The JML is an annotation language used to specify Java programs by expressing formal properties and requirements on the classes and their methods (see [27]).

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 [21]. JML relies on the principles of Design by Contract defined Meyer [31,32], 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 fulfil 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 if the Java program respects the specification.

5.3 Using JML to express the expected properties

We have used the JML assertions for two purposes. First it was used to check consistency of the model. Second, it was used to express explicit requirements (such as those given in Sect. 5.1).

As explained previously, we have derived an abstract implementation from the real one, and have modified it, especially at the appliance level. In order to increase confidence with respect to those modifications, we have introduced JML assertions dedicated to the appliance internal state consistency specification. Those assertions can be both expressed as invariant and post-conditions. Figure 10 describes a part of the `Appliance` object code. The `powerState` and the `internalState` are specified by four invariants given lines 6–9. The evolution of those attributes is specified as post-conditions associated with each method (e.g. lines 26–27 and 43–44).

The implementation of local, global and environment properties were done systematically. Local properties were described both as pre-conditions and invariants in the appliance objects. For example, for the electric kettle, the requirement "*do not open the lid while the water is boiling, or there is a risk of scalding*" is expressed as an invariant stating that the lid should be closed when the kettle is heating:

```
public invariant ((heatingMode.equals ("ON")
&& powerState.equals("ON"))==>
lidStatus.equals("CLOSE"));
```

Moreover, to prevent any misuse of the kettle, we state that the lid should be open only if it is not heating. As a pre-condition of the `openLid` method, we have

```
requires heatingMode.equals("OFF");
```

```
1  public  class Appliance {
     protected /*@  spec_public @*/ String Name; // used for printing messages
     protected /*@  spec_public @*/ String  powerState = "OFF";
     protected /*@  spec_public @*/ String  internalState = "OFF";
5
     //@ public invariant  (!powerState.equals("OFF") ==> powerState.equals("ON"));
     //@ public invariant  (!powerState.equals("ON") ==>  powerState.equals("OFF"));
     //@ public invariant  (!internalState.equals("OFF") ==> internalState.equals("ON"));
     //@ public invariant  (!internalState.equals("ON") ==> internalState.equals("OFF"));
10
     protected /*@  spec_public  non_null @*/ HomeEnvironment currentEnv ;

     // consumption
     //@ public invariant minConsumption<=maxConsumption;
15   //@ public invariant minConsumption >= 0 ;
     //@ public invariant  applianceCurrentConsumption>=0 && applianceCurrentConsumption<=maxConsumption;
     //@ public invariant powerState.equals("ON") ==> (applianceCurrentConsumption >= minConsumption);
     //@ public invariant powerState.equals("OFF") ==> (applianceCurrentConsumption == 0);

20   protected /*@  spec_public  @*/ int maxConsumption=0;
     protected /*@  spec_public  @*/ int minConsumption=0;
     protected /*@  spec_public  @*/ int applianceCurrentConsumption=0;

     //@ requires (minPower <= maxPower) && (minPower >=0);
25   //@ ensures (maxConsumption == maxPower) && (minConsumption == minPower);
     //@ ensures powerState.equals("ON");
     //@ ensures internalState.equals("OFF");
     //@ ensures applianceCurrentConsumption == minPower;
     //@ ensures currentEnv != null ;
30
     public Appliance(int minPower, int  maxPower){
        currentEnv = new HomeEnvironment();
        Name = "Appliance";
        maxConsumption = maxPower;
35      minConsumption = minPower;

        powerState = "ON";
        internalState = "OFF";
        applianceCurrentConsumption = minPower;
40   }


     //@ ensures powerState.equals("ON");
     //@ ensures internalState.equals("OFF");
45   //@ ensures applianceCurrentConsumption == minConsumption;

     public /*@  spec_public  @*/ void powerOn(){
        int preVal = applianceCurrentConsumption;

50      internalState = "OFF";
        powerState="ON";
        applianceCurrentConsumption = minConsumption;
        currentEnv.updateConsumption(minConsumption - preVal);
        System.out.println(Name + " is powered on");
55   }
   [...]
```

Fig. 10 Appliance class annotated with JML

Global properties generally express the expected behaviour of the service. If it is the case, they are expressed as post-conditions of the method. For instance, it is expected that the *DVDTheater* service activation results in switching on the TV. As a post-condition of `Activation` method, we state that

```
ensures (tv.getStatus().equals("ON"));
```

For *cookingPreparation* service, the property "*while the gas valve is opened, the ventilator must be turned off*" is encoded as an invariant:

```
public invariant gasValve.getStatus().
equals("OPEN")
==> (ven.getPower().equals("ON") &&
ven.getStatus().equals("ON"));
```

Environment properties were expressed in the `HomeEnvironment` class, as invariants. For instance, let us consider that the maximum power consumption should not be greater than 30 amperes. This is expressed as

```
//@ public invariant (getConsumption()
<= 30);
```

This property is relevant in our abstract implementation since for each time an appliance is solicited, the powerConsumption of the environment is updated.

In our abstract implementation, we have inserted 209 JML annotations (17 pre-conditions, 150 post-conditions, and 42 invariants).

6 Validation by test

At this point, we want to demonstrate that the abstract implementation conforms to the executable specification. Thanks to the choice of JML, several types of approaches can be followed in order to achieve the validation. In this section, we report the work done with a testing tool. Since JML specifications can be used as an oracle for a test process, we have used a combinatorial testing approach to generate test data. Here, we first cover some principles of testing with JML, before introducing our approach for combinatorial testing.

6.1 JML as a test oracle

JML is executable. It is possible to use invariant assertions, as well as pre- and post-conditions as an oracle for conformance testing. JML specifications are translated into Java by the `jmlc` tool, added to the code of the specified program, and checked against it, during its execution.

The executable assertions are thus executed before, during and after the execution of a given operation. Invariants are properties that have to hold in all *visible states*. A visible state roughly corresponds to the initial and final states of any method invocation [21]. When an operation is executed, three cases may happen. **All checks succeed**: the behaviour of the operation conforms to the specification for these input values and initial state. The test delivers a PASS verdict.

An **intermediate or final check fails**: this reveals an inconsistency between the behaviour of the operation and its specification. The implementation does not conform to the specification and the test delivers a FAIL verdict.

An **initial check fails**: in this case, performing the whole test will not bring useful information because it is performed outside of the specified behavior. This test delivers an INCONCLUSIVE verdict. For example, $\sqrt{x}$ has a precondition that requires x being positive. Therefore, a test of a square root method with a negative value leads to an INCONCLUSIVE verdict.

6.2 Principles of the test case generation

Combinatorial testing performs combinations of selected input parameters values for given operations and given states. For example, a tool like JML-JUnit generates test cases which consist of a single call to a class constructor, followed by a single call to one of the methods (see [11]). Each test case corresponds to a combination of parameters of the constructor and parameters of the method.

The LIG laboratory has developed Tobias [6,28], a test generator based on combinatorial testing [12]. It adapts combinatorial testing to the generation of sequences of operation calls. The input of Tobias is composed of a test pattern (also called test schema) which defines a set of test cases. A schema is a bounded regular expression involving the Java methods and their associated JML specification. Tobias unfolds the schema into a set of test cases: all combinations of the input parameters for all operations of the schema are computed. The test suite can be turned into a JUnit file thanks to Tobias.

The schemas may be expressed in terms of *groups*, which are structuring facilities that associate a method, or a set of methods, to typical values. Groups may also involve several operations. For instance, let us consider the Blind class. It has four main methods (`powerOn()`, `powerOff()`, `Open()`, and `Close()`) and several constructors (but for the following we only consider the simplest one `Blind()`). For testing the Blind class, one can design the following schema:

```
T-Blind = Init ; BlindOp^{4..4} with
  Init = {Blind aBlind = new Blind()}

  BlindOp = {aBlind.powerOn()}∪
  {aBlind.powerOff()}∪{aBlind.Open()}
  ∪{aBlind.Close()}
```

The Schema is unfolded in Tobias tool and can be viewed in an intermediate format (see Fig 11). `Init` is a set of only one instantiation. `BlindOp` is a set of four instantiations. The suffix `^{4..4}` means that the group is repeated four times. `T-Blind` is unfolded into $1^*(4^*4^*4^*4) = 256$ test cases. It is then possible to produce an executable JUnit file (see Fig. 12).

6.3 Validation of the abstract implementation

To validate our abstract implementation, we have designed several test schemas corresponding to different phases in the validation process. First, each appliance was tested in isolation (with schema such as T-Blind presented above) and in the context of the home. Schemas similar to `T-Blind` were produced for each appliance. Those schemas allow you to reach 100% of statement coverage for each appliance component.

```
  1. Blind aBlind = new Blind(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOn();
  2. Blind aBlind = new Blind(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOff();
  3. Blind aBlind = new Blind(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOn(); aBlind.Open();
  4. Blind aBlind = new Blind(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOn(); aBlind.Close();
  5. Blind aBlind = new Blind(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOff(); aBlind.powerOn();
  6. Blind aBlind = new Blind(); aBlind.powerOn(); aBlind.powerOn(); aBlind.powerOff(); aBlind.powerOff();
  ...
255. Blind aBlind = new Blind(); aBlind.Close(); aBlind.Close(); aBlind.Close(); aBlind.Open();
256. Blind aBlind = new Blind(); aBlind.Close(); aBlind.Close(); aBlind.Close(); aBlind.Close();
```

Fig. 11 Abstract representation of the 256 test cases produced from T-Blind

```
import junit.framework.TestCase;
import junit.framework.*;
public class Testsuite_T-Blind extends TestCase {

    static int nb_inc = 0;
    static int nb_fail = 0;

    public static void main(String args[]) {
        junit.textui.TestRunner.run(new junit.framework.TestSuite(Testsuite_T-Blind.class));
        System.out.println("inconclusive tests: "+Testsuite_T-Blind.nb_inc);
        System.out.println("failures : "+Testsuite_T-Blind.nb_fail );
    }

    public void testSequence_1(){
      try{
            Blind aBlind = new Blind();
            aBlind.powerOn();
            aBlind.powerOn();
            aBlind.powerOn();
            aBlind.powerOn();
        } catch(org.jmlspecs.jmlrac.runtime.JMLEntryPreconditionError e$){
            System.out.println("\n INCONCLUSIVE "+
                  (new Exception().getStackTrace()[0].getMethodName())+ "\n\t "+ e$.getMessage());
            Testsuite_T-Blind.nb_inc++;
        }
        catch(org.jmlspecs.jmlrac.runtime.JMLAssertionError e$){ // test failure
            int l$ = org.jmlspecs.jmlrac.runtime.JMLChecker.getLevel();
            org.jmlspecs.jmlrac.runtime.JMLChecker.setLevel(org.jmlspecs.jmlrac.runtime.JMLOption.NONE);
            try {
                Testsuite_T-Blind.nb_fail++;
                junit.framework.Assert.fail("\n\t" + e$.getMessage());
            }
            finally {
                org.jmlspecs.jmlrac.runtime.JMLChecker.setLevel(l$);
            }
          }
       }

     public void testSequence_2(){
       try{
            Blind aBlind = new Blind();
            aBlind.powerOn() ;
            aBlind.powerOn();
            aBlind.powerOn();
            aBlind.powerOff();
          } ...
```

Fig. 12 JUnit file produced from T-Blind

Appliance isolation testing revealed several INCONCLUSIVE verdicts because some pre-conditions of some operations were not satisfied. For instance, within the kettle test schemas, the `openLid` method of the kettle can be called when it is in the boiling mode. This is not supposed to be done due to the kettle's local properties. These INCONCLUSIVE

verdicts were expected each time some specific local properties were implemented. They correspond to a usage of the appliance that does not conform with its manual.

Appliance isolation testing also revealed several FAIL verdicts, which were not expected. A careful analysis showed that appliance implementations were sometimes inconsistent with the JML assertions. Those inconsistencies resulted mainly in the evolutions of the model and the specification, which was sometimes not completely carried out.

In a second phase, we have focused on the integrated service validation. The main objective was to activate each service in different situations (in order to be sure that a service can be activated in any cas). Two types of test sequences were produced and executed. Both sets of tests were composed of a prologue followed by the activation of the service under test. Those schemas made it possible to reach 100% of statement coverage for each service component.

The first set of tests was dedicated to the service activation's validation with respect to the different appliance states. To do that, the test prologue consisted of three or four different calls to one appliance. This was aimed at checking that the services could work correctly whatever the state of each appliance (taken independently). For example, the following schema `TS-vs-Blind` initializes the home, put the blind in a specific state and then activates DVDtheatre or Relax services.[6]

```
TS-vs-Blind = Init ; BlindOp^{4..4} ; S-Ac with
  Init = {Home aHome = new Home()}

  BlindOp = {aHome.blindPowerOn()}∪
  {aHome.blindPowerOff()}∪{aHome.blindOpen()}
  ∪{aHome.blindClose()}


  S-Ac = {aHome.DVDTheatreActivation()}∪
  {aHome.RelaxServiceActivation()}
```

This type of test sequences allowed us to detect that some calls or checks were forgotten for some services. For instance, in the RelaxService, the kettle was not closed before switching on. This problem was not discovered during preliminary tests because when the kettle object is created, its lid is closed. By applying several consecutive calls on the kettle before activating the RelaxService, we were able to discover the implicit requirement about the kettle lid. We corrected the service by systematically closing the lid before switching it on.

The second test set was dedicated to the service activation validation against different environment states (temperature, time, sound level, current consumption, etc.). To do that, the prologue consisted of applying different parameters to the environment attributes (thanks to the public methods such as `set_temperature(int)`). This aimed at checking that the service could work correctly whatever the state of the environment. For example, the following schema `T-Service-vs-time` initializes the home, set the current time and then activates DVDtheatre or Relax services.

```
T-Service-vs-time = Init ; timeOp ; S-Ac with
  Init = {Home aHome = new Home()}

  timeOp = {aHome.set_time(val)}

  val = {2, 8, 12, 15, 18, 22}

  S-Ac = {aHome.DVDTheaterActivation()}∪
  {aHome.RelaxServiceActivation()}
```

Two environmental properties were stated in our system with respect to the loud sound ("*do not make loud voice or sound after 9 p.m.*") and the power consumption (*"the maximum power consumption should not be greater than 30 amperes"*.). These were not taken in to account by real implementation, and thus six errors were found.

Five errors are related to the five services that switch on some appliances. By choosing a initial power-consumption near by 30 amperes, it is possible to make those services violating the power-consumption property. To make the services consistent with this requirement, the power-consumption should be checked before switching on any appliance.

One error is related to the DVD Theater Service that violates the environment property "*do not make loud voice or sound after 9 p.m.*" if it is activated after 9 p.m. To make the DVD Theater Service consistent with this requirement, one has to modify the implementation so that the sound level is adjusted with respect to the time.

More than 30 test schemas were described and unfolded in the Tobias plug-in for Eclipse. Schemas were unfolded in test suites, having between 500 and 5000 test cases. The unfolding phase lasts at most 2 min for the biggest schemas.[7] Test cases then were translated in the JUnit format and executed within the Eclipse environment. It took at most 500 s for the biggest sets of test cases. Ten errors were found at the appliance and service levels (local and global properties). The statement coverage was checked with EclEmma plug-in.[8]

[6] As indicated Sect. 4.3, at the abstract level, a `Home` object is used to configure a specific home configuration. At the home level, it is possible to access to all public methods of the appliance and the services. It is also possible to configure the environment.

[7] Tobias was executed on a laptop, equipped with a 1.5GHz processor and 512 MO of RAM, with Window XP OS.

[8] http://www.eclemma.org/.

7 Verification by extended static checking or proof

7.1 Tools for verification

Several tools can be chosen to carry out verification. During our experimentation, we used two of them:

ESC/Java 2 and KeY.

ESC/Java[9] is an *Extended Static Checking* tool [17]. The verification is *static* since the code is verified without being executed. It is *extended* since the tool detects more errors than can be detected with traditional static analysis.

ESC/Java 2 uses automatic theorem-proving techniques to reason on the program semantics. It raises warnings in case of classical runtime errors, such as null dereferences, array bound errors, type cast errors etc. It also warns about synchronization errors in concurrent programs (race conditions and deadlocks). Finally, ESC/Java 2 issues warnings if the source code violates the JML assertions.

The KeY [10] system is a formal software development tool that aims to integrate design, implementation, formal specification, and formal verification of object-oriented software as seamlessly as possible [1,4]. At the core of the system is a theorem prover for the first-order Dynamic Logic for Java. The tool has an easy-to-use graphical interface and seamlessly integrates automated and interactive proving.

7.2 Using ESC/Java 2

ESC/Java 2 was the first tool to be used during the proof process. The verification was carried out on the code, which was tested and corrected. The verification of the Java classes was long and not easy. Even if the JML assertions were used during the test, they were not enough to allow an automatic verification.

The first ESC/Java 2 warnings were obtained for `Appliance.java`. They were related to the use of set methods of `HomeEnvironment.java`. The use of these methods in `Appliance.java` could lead to a violation of `HomeEnvironment` assertions. In order to solve the problem and to simplify the relation between the appliances and the environment `Appliance.java` class had to be refactored. This operation impacted all sub classes of `Appliance.java`.

Then, to remove several warnings, several JML assertions had to be inserted. Indeed, several post-conditions were added to specify the type of the returned result. For instance, to the method `public String getPower()`, the following postcondition was added `//@ ensures \ result.equals (powerState);`.

Several warnings were related to the problem of null values. Some assertions were added in order to specify that the variables or the attributes (of type String) were not null. For instance, for `Appliance` classes, in lines 3 and 4 given in Fig. 10, expressions `/*@ spec_public @*/` were replaced by `/*@ spec_public non_null @*/`. Moreover, all constructors of appliances were modified in order to initialize all attributes explicitly as it is done for `Appliance` class.

After the code refactoring, the code size represents 2400 lines of code. The new JML assertions represent more than 600 lines of code. At the end of the process, all appliance and service properties seemed to be validated: there were no remaining warning. However, one has to be careful. ESC/Java 2 is neither complete nor sound. Some errors may not be reported and false alarms may be reported.

7.3 Using KeY

Proving the application with KeY was not as successful as expected. The verification could be carried out only for the file `HomeEnvironment.java`. The main reason was that the JML assertions deal with strings, which are not currently supported by Key.

In this version, the class `HomeEnvironment.java` has 27 methods (1 constructor, 11 get methods, 15 set methods) and 3 invariants. Each get method has been declared as `pure`. Half of the set methods were associated with a precondition. None of them has a postcondition. KeY produced between 3 and 5 proof obligations for each method. All of them were proved. Most of them were proved automatically with Simplify or Yices provers. For three methods, the "elementary arithmetic strategy" has to be used. For one method, we have to increase the number of computing steps (1100 instead of 1000 by default).

7.4 Analysis of the verification process

From a general point of view, the validation of the code with ESC/Java 2 required more work than expected for two reasons. First, the code of appliance API and integrated services is quite simple (no loop for instance), so it was expected that the verification would be easy. Second, it was expected that the assertions and the code were consistent, since the code was previously intensively tested.

We can distinguish two types of works done to support the verification process. First, code and an implementation compatible with the tool abilities had to be provided. During our experiment, we have to refactor both the code and the Java implementation in order to carry out verification with ESC/Java 2. It mainly consisted in a simplification of the

[9] ESC/Java can be downloaded at http://kind.ucd.ie/products/opensource/ESCJava2/download.html.

[10] KeY can be downloaded at http://www.key-project.org/download/.

```
public class TVservice {
  TV tv = new TV();

  public boolean up_vol(int vol) {
    try {
      tv.up_vol(vol);
    }
    catch (org.jmlspecs.jmlrac.runtime.
          JMLEntryPreconditionError e) {
      System.out.println("TV: precondition error");
    }
  }
```

Fig. 13 New TVService class

coupling of the methods and classes. Second, one may have to add specific assertions to help the tools. For this application, the assertions to be inserted were mainly related to null values. It also concerned some indications of the value of the return results.

It must be noticed that a large effort was required to adapt the code and the JML specification to the verification process. However, it was not possible to relate the warnings obtained with ESC/Java 2 to any bug in the code or in the specification. It seems that improvements only restructured the code and increased the redundancy of the specification.

8 On-line monitoring

During the process of validation, we improved our JML specifications and the code of the abstract application. The final step of the approach was to update the final implementation as presented in Fig. 1d.

To do that, we started from the Java files of the abstract implementation, organised as described in Fig. 8. We re-introduced the code necessary to manage the remote controller. This step corresponds to the "refinement" step. The Java files of the real implementation were replaced by the new Java files of the abstract level with the same name, and were compiled with the `jmlc` compiler. The classes used for exporting the component as services were modified in order to catch the JML exceptions, as it is illustrated in Figs. 13 for TVService class. The JML assertion mechanism is embedded in the final system. So if the system is in a state that is incompatible with the pre-conditions of a service that is executed, a JML exception is raised and the execution of the service is suspended. The showroom of Kobe university has been controlled with this new system in order to check the feasibility of the approach. One of our perspective now is to measure the reliability of the first and the second implementations, in order to evaluate how much it was improved by the process.

9 Related work

Service oriented architecture (SOA) is used for creating modern services and systems. But lack of confidence prevents its adoption by mainstrain service computing. A key issue is thus to provide all actors the means to check that a service delivers the expected function with the expected quality of service (QoS) [10,13].

In the introduction, we underline the fact that for practical use of such services, it is essential to guarantee the correctness, the safety and the security of the services. In this article, we have focused on correctness and safety. In [2], Balfe et al. examine useful properties for security in the context of pervasive environment. In [9], Candolin presents some security requirements in the context of service-oriented architecture. A security framework is presented as part of a security architecture for network centric environment. In [15], Dragoni et al. advocate the notion of security-by-contract as a solution to pervasive download problems. The idea is that each application to be downloaded should come with a contract containing the description of the relevant features. Only an application matching the platform security policy will be allowed to be downloaded.

About correctness, a large amount of works focus on service composition (orchestration or choregraphy) of (web) services. In [37], challenges and solutions for model-driven web service composition are described. [8,18,33,38,43] focus on formal verification of the composition. For instance, in [33], the services specification (in Web Service Flow Language—WSFL) are translated into Promela. Properties such as reachability and deadlock freedom are expressed in LTL. They are verified thanks to the SPIN model-checker. In [43], the verification of properties such as liveness and reachability is performed thanks to petri-nets. In [38], SMV is used to detect such properties in the context of telephony services.

All those approaches use a specification language, very different from the service implementation language. This means that one has to deal with at least two (very) different languages. In our approach, the model is abstract from the real implementation. It is expressed in Java like the implementation. Moreover, the expected properties are expressed also in a Java-like language. This eases the process of specification.

We have chosen a formal method approach based on contracts. The use of contract is suggested by practitioners as a help for service design.[11,12,13] Contracts have also been used for formally specified web services [25,14]. The contracts were especially used as a testing oracle and for monitoring in [14].

[11] http://dev2dev.bea.com/blog/bhensle/archive/2007/04/soa_and_design.html.

[12] http://archive.devx.com/javaSR/articles/smith1/smith1p.asp.

[13] http://www.myarch.com/design-by-contract-for-web-services.

10 Summary, lessons learnt and perspectives

Summary

Home network systems are critical applications. Before becoming widespread, it is essential to guarantee the correctness, the safety and the security of the services. In this article, we express the correctness and the safety properties as assertions. The use of JML makes it possible to embed them into the Java code. It was possible to use the assertions as an executable test oracle and use it as a specification to be conformed to in a verification process.

During the experimentation, we derived an abstraction of the real implementation. This was necessary before being able to carry out tests and proofs. It allows us also to experiment new types of appliances and new integrated services. The specification was composed of 209 JML annotations (17 pre-conditions, 150 post-conditions, and 42 invariants). The abstract implementation was composed of 14 appliances and 7 integrated services. It was tested with combinatorial tests. Testing allowed us to detect 16 errors. Ten errors were related to local and global properties (inconsistencies between the code and the specification, corrected during the process). Six errors were related to the environmental properties that were not taken into account by the real implementation.

Tools for static extended checking and deductive proof were also experimented. In order to check the abstract implementation, refactoring had to be carried out for both code and assertions. It is still not clear to us if the refactoring has to be carried out in order to help the tools or in order to correct some remaining errors.

The abstract code was then refined and finally executed in the real implementation context. The assertions were used to monitor the execution of the showroom.

Lessons learnt

The main objective of our work was to find some solutions to establish the correctness of a real implementation of a home network system. An application such as our HNS cannot be easily tested or proved, either because the system is too complex to be proved, or because services may behave with respect the state of the environment (e.g. temperature), which cannot be easily influenced with software tests.

In order to carry out the validation and verification, we choose to build an abstraction of the implementation. One lesson is that the choice of the level of abstraction is very important. During this study, we mainly carried out an abstraction on the web-service technology, the drivers and the environment. Now, it is clear to us that this abstraction was more important than necessary for the testing phase. For instance, the abstraction on the drivers was not necessary (it would have been possible to execute the same tests with the drivers). However, it is also clear that the abstraction was not enough to carry out verification by proof. We encoded the state of the appliances by Strings, which are not supported by the tool used. The level of abstraction should be determined with respect to the planned validation phases and the known the limits of the different tools.

A similar remark can be made for the expression of the JML assertions. First, one should choose an adequate sub-set of JML. If JML is a quite rich language for expressing assertions, it is not yet completely supported by all the tools. Tools such as ESC/Java 2 or KeY support only a part of the JML constructions. Similarly, testing tools such as Tobias rely on the fact that the assertions have to be executable. And some JML constructions are not executable by default (for instance, a `\forall` is not executable if the following expression does not concern a JML set or an integer interval).

Perspectives

There are two main directions in which we want to work. In our context, local, global and environmental properties were often associated with a risk to be prevented. For instance, the kettle usage is restricted in order to prevent the risk of scalding. Of course, one of the difficulty is to express "all" the requirements. Another one is to be able to translate them at the system level, since the risks depend on the configuration of the home and the appliances. For the kettle, scalding can be prevented only if it is possible to close the lid thanks to the system; and it is not the case for most of the kettles. Gas-poisoning is a risk that can be prevented only if the system is able to command the gas-valves or if it is able to renew the air. We are currently working to associate the presented approach with a methodology in order to collect and to translate properties into relevant assertions (with respect to the home configuration).

The second main direction of our work is to provide a framework to ease the extension of the existing system. Indeed, the translation of the real implementation into an abstract implementation is not yet automated and, thus, is error-prone. So, to add new appliances and services, we propose to express them directly at the abstract level, and to validate them. Then, it would be possible to transform the classes of the abstract implementation into skeletons of classes for the real implementation, which have to be completed before their final use. To reduce the probability of errors introduced at that step, the skeletons should be as complete as possible.

Acknowledgments This research was partially supported by the Japanese Ministry of Education, Science, Sports and Culture, Grant-in-Aid for Young Scientists (B) (No. 18700062), Scientific Research (B) (No. 17300007), and Comprehensive Development of e-Society Foundation Software program. It is also supported by JSPS and MAE under the Japan-France Integrated Action Program (PHC-SAKURA) 2007-08.

References

1. Ahrendt W, Baar T, Beckert B, Bubel R, Giese M, Hähnle R, Menzel W, Mostowski W, Roth A, Schlager S, Schmitt PH (2005) The KeY Tool. Softw Syst Model 4:32–54
2. Balfe S, Li S, Zhou J (2006) Pervasive trusted computing. In: 2nd international workshop on security, privacy and trust in pervasive and ubiquitous computing (SecPerU), pp 88–94. IEEE Computer Society, Lyon
3. Bartetzko D, Fischer C, Möller M, Wehrheim H (2001) Jass-Java with Assertions. Electr Notes Theor Comput Sci 55(2)
4. Beckert B, Hähnle R, Schmitt PH (eds) (2007) Verification of object-oriented software: the KeY approach. LNCS 4334. Springer
5. Bohn J, Coroama V, Langheinrich M, Mattern F, Rohs M (2005) Social, economic, and ethical implications of ambient intelligence and ubiquitous computing. In: Weber W, Rabaey J, Aarts E (eds) Ambient intelligence. Springer, Berlin, pp 5–29
6. Bousquet L, Ledru Y, Maury O, Oriat C, Lanet JL (2004) A case study in JML-based software validation (short paper). In: Proceedings of 19th Int. IEEE Conf. on Automated Sofware Engineering (ASE'04), pp 294–297
7. Burdy L, Cheon Y, Cok DR, Ernst MD, Kiniry JR, Leavens GT, Leino KRM, Poll E (2005) An overview of JML tools and applications. STTT 7(3):212–232
8. Busi N, Gorrieri R, Guidi C, Lucchi R, Zavattaro G. (2005) Towards a formal framework for choreography. In: Enabling technologies: infrastructure for collaborative enterprise. 14th IEEE international workshops on 13–15 June 2005, pp 107–112
9. Candolin C (2007) A security framework for service oriented architectures. In: Military communications conference, 2007. MILCOM. IEEE, pp 1–6, 29–31 Oct 2007
10. Canfora G, Di Penta M (2006) Testing services and service-centric systems: challenges and opportunities. IT Prof 8(2):10–17
11. Cheon Y, Leavens G (2002) A simple and practical approach to unit testing: the JML and JUnit way. In: ECOOP 2002. LNCS, vol 2474. Springer, pp 231–255
12. Cohen D, Dalal S, Parelius J, Patton G (1996) The combinatorial design approach to automatic test generation. IEEE Softw 13(5):83–88
13. Controneo D, Di Flora C, Russo S (2003) Improving dependability of service oriented architectures for pervasive computing. Object-oriented real-time dependable systems, 2003. (WORDS 2003). In: Proceedings of the eighth international workshop on 15–17 Jan 2003, pp 74–81
14. Dai G, Bai X, Wang F, Dai F (2007) Contract-based testing for web services. In: 31st annual international computer software and applications conference (COMPSAC). IEEE Computer Society, Beijing, China, pp 517–526
15. Dragoni N, Massacci F, Naliuka K, Siahaan I (2007) Security-by-contract: toward a semantics for digital signatures on mobile code. In: 4th European public key infrastructure workshop: theory and practice (EuroPKI). LNCS, vol 4582. Springer, Palma de Mallorca, Spain, pp 297–312
16. du Bousquet L (1999) Feature interaction detection using testing and model-checking, experience report. In: World congress on formal methods. LNCS, vol 1708, Springer, Toulouse, pp 622–641
17. Flanagan C, Leino KRM, Lillibridge M, Nelson G, Saxe JB, Stata R (2002) Extended static checking for Java. In: Proc. of the ACM SIGPLAN 2002 conference on programming language design and implementation. ACM Press, pp 234–245
18. Foster H, Uchitel S, Magee J, Kramer J (2003) Model-based verification of Web service compositions. In: Automated software engineering, 2003. Proceedings. 18th IEEE international conference on 6–10 Oct 2003, pp 152–161
19. Geer D (2006) Nanotechnology: the growing impact of shrinking computers. Pervasive Comput 5(1):7–11
20. Jackson M (1999) The role of formalism in method. In: Formal methods, world congress on formal methods in the development of computing systems (FM99). LNCS, vol 1708. Springer, Toulouse, p 56
21. The JML Home Page (2005). http://www.jmlspecs.org
22. Karaorman M, Abercrombie P (2005) Contractor: introducing design-by-contract to java using reflective bytecode instrumentation. Form Methods Syst Des 27(3):275–312
23. Karaorman M, Holzle U, Bruno J (1999) Contractor: a reflective java library to support design by contract. Tech. rep., Santa Barbara
24. Kolberg M, Magill E, Wilson M (2003) Compatibility issues between services supporting networked appliances. IEEE Commun Mag 41:136–147
25. Lamparter S, Luckner S, Mutschler S (2007) Formal specification of web service contracts for automated contracting and monitoring. In: 40th Hawaii international conference on systems science (HICSS). IEEE Computer Society, Big Island, p 63
26. Langheinrich M, Coroama V, Bohn J, Mattern F (2005) Living in a smart environment—implications for the coming ubiquitous information society. Telecommun Rev 15(1):132–143
27. Leavens G, Baker A, Ruby C (1999) JML: a notation for detailed design. In: Kilov H, Rumpe B, Simmonds I (eds) Behavioral specifications of businesses and systems. Kluwer, Dordrecht, pp 175–188
28. Ledru Y, du Bousquet L, Maury O, Bontron P (2004) Filtering TOBIAS combinatorial test suites. In: Fundamental approaches to software engineering (FASE'04). LNCS, vol (to appear). Springer, Barcelona
29. Loke SW (2003) Service-oriented device ecology workflows. In: First international conference on service-oriented computing (ICSOC 2003). LNCS, vol 2910. Springer, Trento, pp 559–574
30. Matsushita Electric Industrial Co., L Kurashi Net (jp). http://national.jp/appliance/product/kurashi-net/
31. Meyer B Object-oriented software construction, 2nd edn
32. Meyer B (1992) Applying design by contract. Computer 25(10): 40–51
33. Nakajima S (2002) Model-checking verification for reliable web services. In: Workshop on Object-Oriented Web Services, collocated with OOPSLA
34. Nakamura M, Tanaka A, Igaki H, Tamada H, Matsumoto K (2006) Adapting legacy home appliances to home network systems using web services. In: Int. Conf. on Web Services (ICWS 2006). IEEE, pp 849–858
35. Nakamura M, Tanaka A, Igaki H, Tamada H, Matsumoto K (2008) Constructing home network systems and integrated services using legacy home appliances and web services. Int J Web Serv Res, to appear
36. Papazoglou MP, Georgakopoulos D (2003) Special issue: service-oriented computing. Introduction. Commun ACM 46(10): 24–28
37. Pfadenhauer K, Dustdar S, Kittl B (2005) Challenges and solutions for model-driven Web service composition. Enabling technologies: infrastructure for collaborative enterprise, 2005. In: 14th IEEE international workshops on 13–15 June 2005, pp 126–131
38. Plath M, Ryan MD (2000) The feature construct for SMV: semantics. In: Feature interactions in telecommunications and software systems VI. Glasgow, pp 129–144
39. Rushby JM (1999) Mechanized formal methods: where next? In: Formal methods, world congress on formal methods in the development of computing systems (FM99). LNCS, vol 1708. Springer, Toulouse, pp 48–51

40. TOSHIBA: Toshiba home network: feminity. http://www3.toshiba.co.jp/feminity/feminity_eng/
41. Weiser M (1993) Some computer science issues in ubiquitous computing. Commun ACM 36(7):74–84
42. Yan B, Nakamura M, du Bousquet L, ichi Matsumoto K (2007) Characterizing safety of integrated services in home network system. In: 5th international conference on smart homes and health telematics (ICOST), LNCS, vol. 4541. Springer, Nara, pp 130–140
43. Yi X, Kochut K (2004) A CP-nets-based design and verification framework for Web services composition. IEEE int. conf. on web services, 6–9 July 2004, pp 756–760