Using Formal Methods to increase confidence in one Home Network System implementation: Case study

Lydie du Bousquet*, Masahide Nakamura**, Ben Yan***, Hiroshi Igaki**

*Universités de Grenoble, Laboratoire LIG, BP 72, 38402 Saint Martin d'Hères cedex, France
lydie.du-bousquet@imag.fr
**Graduate School of Engineering Science, Kobe University, Japan
{masa-n,igaki}@cs.kobe-u.ac.jp
***Graduate School of Information Science, Nara Institute of Science and Technology, Japan
hon-e@is.naist.jp

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 one home network system. We relies on the Java Modeling Language (JML), to formaly specify and validate a model of the system. it.

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. Several HNS products are already on the market (e.g. Hitachi (2007); Matsushita (2007); Toshiba (2007)).

A HNS provides many 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 (see Kolberg et al. (2003)), which we call HNS *integrated services*. 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 adversaries to intrude or hijack 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 paper, 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. (2006, 2008). Our approach relies on a Design by Contract strategy. The Java Modeling Language JML (2005), an executable specification language, is used for both off-line and on-line validation.

In section 2, we first present the Home Network System (HNS) principles. We then describes the framework developed by Nakamura et al., in which *integrated services* are deployed. In section 3, we detail the needs in terms of validation for HNS *integrated services*. Section 4 and 5 describe how the model was built and specified. Section 6 focuses on the validation process. Section 7 concludes and draws some perspectives.

2 Context

2.1 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 integrated services of networked home appliances (called *integrated service* in the following). An integrate 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* 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 with a boiling mode to prepare hot water for coffee.

2.2 A framework for implementing HNS integrated services

As the embedded devices become more downsized, cheaper, and more energy-saving, it is expected in the near future that every object will be networked Geer (2006). However, transition to the networked appliances is gradual. Most people are still using legacy appliances, which are the conventional non-networked home appliances, although it is usual to see a network (and PCs) at home. Indeed, networked appliances are expensive. Due to the interoperability problem, the integration of appliances is limited especially in the multi-vendor environment.

To cope with both the emerging HNS and the legacy appliances, Nakamura et al. (2006, 2008) have proposed a framework that adapts the legacy appliances with conventional infrared remote controllers. 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 Loke (2003); Papazoglou and Georgakopoulos (2003)).

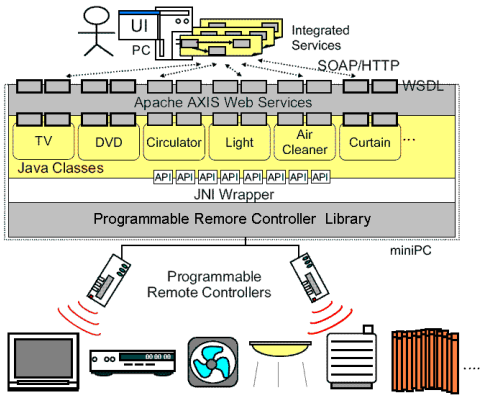

FIG. 1 – *HNS*

For each appliance, a self-contained component is implemented in Java and deployed as web service (using Apache AXIS) (Fig. 1). Methods like `On()` and `OFF()` are open interfaces for accessing 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. 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. For each appliance component, a `getStatus()` method returns the current state (i.e., the attribute values).

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 value can be obtained by the way of some sensors (such as thermometer, fire detector...), which are implemented as web service like other appliance components.

2.3 HNS Integrated Services

Integrated services can be implemented in this framework as client applications. An integrated service invokes the methods of the appliance components.

Fig. 2 shows a Java-like pseudo code which implements the scenario DVD Theater service. In this figure, X.Y() means the invocation of API Y() of appliance X. In the considered framework, an integrated service component has methods, to initialize the service (method `initialisation` in Fig. 2), and to activate it (`activation` in Fig. 2). Some services may also provide method to deactivate the service. For instance, method `deactivation` in Fig. 2, switches off the appliances used by the DVD theater service.

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

  void initialisation(DigitalTV aTV, DVDPlayer aDvd,
      SoundSystem aSpeaker, Light aLight, Curtain aCurtain){
      tv= aTV; ...
  }
  void activation() {
    tv.switchOn();              /* Turn on TV */
    tv.setVisualInput(.DVD.);
    dvd.switchOn();             /* Turn on the DVD Player */
    dvd.setSoundOutput(.5.1.);
    speaker.switchOn();         /* Turn on the Sound System */
    speaker.setInputSource(.DVD.);
    speaker.setVolumeLevel(25);
    curtain.closeCurtain();     /* Close curtain */
    light.setBrightnessLevel(1); /* Minimize brightness */
    dvd.playDvd();              /* Play DVD */
  }
  void deactivation() {
    tv.switchOff();
    dvd.switchOff();
    speaker.switchOff();
    curtain.openCurtain();
} }
```

FIG. 2 – *A pseudo-code for the DVD Theater Service*

```
public void switchOn(void) {
 IrControler con = new IrController();  /* Controller and */
 IrSignal sig = new IrSignal();         /* signal objects of Ir-API*/
 sig.setSignalType(SWITCH_ON, TV_A);    /* set signal ON for TV_A*/
 con.sendSignal(sig);                   /* send the signal*/
 sleep(2);                              /* Sleep during 2 seconds*/
 state="ON";
}
```

FIG. 3 – *Concrete implementation of method* `switchOn()` *for TV_A*

3 Increasing confidence in HNS integrated services

Before providing a HNS and integrated services, one must guarantee that the implementation is correct and "safe" for inhabitants, house properties and their surrounding environment.

3.1 Requirements at different levels

Yan et al. (2007) have identified three levels of requirements that can be expected from integrated services.

For every electric appliance, the manufacturer prescribes a set of safety instructions for proper and safe use of the appliance. Conventionally, these instructions have been designated for human users. However, in the HNS integrated service, the instructions must be guaranteed within the software. For instance, the following shows a safety instruction for an electric kettle: *do not open the lid while the water is boiling, or there is a risk of scald*. 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 necessary to consider global properties over the multiple appliances. For instance, the Cooking Preparation Service (which automatically sets up the kitchen configuration of 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 inhabitants and neighbors for safety. Since the integrated services give various impacts against the surrounding environment (including the room, the building, the neighbors, 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 satisfies all local properties, i.e. all properties derived from the appliance instructions. It is *globally correct* iff it satisfies all properties prescribed for it. It is *environmentally correct* iff it satisfies all properties derived from the environment where it is provided.

3.2 Using formal methods: choosing an approach

We want to increase the confidence of the real implementation of the integrated services. A first experimentation of formalization and validation was done in Leelaprute et al. (2005). A model of services and appliances was proposed and expressed in the SMV language McMillan (1993). Expected properties of the services were also expressed and the SMV model-checker was used to prove that the services satisfy the properties.

The experimentation has shown that SMV is expressive enough to describe practical services. It also enables a compact modeling independent of the underlying HNS protocols or specific platform. However, this approach appears to be not sufficient for at least three reasons.

The use of a model-checker greatly increases confidence in the *model*. However, the model was not derived from the real system. Abstraction and/or mistakes in the model or the incorrect property expression may lead to misleading conclusions[1].

[1] This has been observed in du Bousquet (1999).

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

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

With respect to this analysis, we choose to express a model in the same language used for the real application implementation: Java. Moreover, we choose the Java Modeling Language (JML) to express the specification, for several reasons. It is very close to Java, and it is supposed to be easier for programmers to learn. It is executable and can be embedded in the final code to monitor its execution. Several tools are available for proof and testing. It supports more constructions than the “assert mechanism" available in Java.

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

1. A model was extracted from the real application (especially appliances and services) and completed (see Sec. 4) .
2. The model was annotated with JML (see Sec. 5).
3. A validation step was carried out (see Sec. 6). This step has two complementary goals. First it was required to detect inconsistencies in the model and to improve correction/robustness of the services. Second, it helped to improve, to detail or to correct the JML annotations.
4. On this basis, the real implementation is currently modified. JML annotations are introduced in the real implementation code in order to monitor the execution (current work).

Steps 1 to 3 are dedictated to the elaboration and the validation of the specifications of the HNS system and integrated services (expressed as annotations). It also helps to test a *model* of the existing system. This model is too abstract from the real implementation to guarantee the correction of the real implementation, that is why step 4 is carried out. The annotations cannot directly be inserted in the implementation; otherwise the real appliances have to be sollicitated during the specification validation (test).

4 Model construction

The basis of the model was thus built as a simplification of the implementation. Each component representing the appliances and the services were modified so that the calls to the remote control API (Fig. 3) were transformed into simple printing message calls (Fig. 4).

Then the model was extended. First, the environment of the house was implemented as a specific object, called `Environment`. Environment object attributes represent sensor measurements. For instance, a `temperature` attribute is the abstraction of thermometer sensor output. For each of these attributes, a `set_attribute()` method is provided to allow the

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

FIG. 4 – *Simplification of method* `switchOn()` *for TV_A in the model*

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

FIG. 5 – *Evolution of method* `switchOn()` *for TV_A in the model*

environment changes. A `get_attribute()` method allows the interrogation of the sensor by any appliances or services.

Second, a home object was implemented. It corresponds to a particular configuration of the home network systems. From the home interface, it is possible to call any public methods of service, appliance or environment objects.

Finally, the appliance objects were modified. The initial implementation did not take into account the fact that appliances can be powered on or off. Internal state of appliances and the code of some methods were completed to deal with the power state. One inheritance level was introduced in order to factor common methods such as `powerOn()`, `powerOff`, `switchOn()` and `switchOff()`. At last, the influences of the appliances on the environment (such as temperature, sound, ...) were also added. Our current model is composed of 25 classes among which 14 appliance components and 7 services.

5 Using Java Modeling Language

5.1 Brief description of JML

The Java Modeling Language (JML) is an annotation language used to specify Java programs by expressing formal properties and requirements on the classes and their methods (see Leavens et al. (1999)).

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 (2005). JML relies on the principles of Design by Contract defined by Meyer (1992), 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)` `\forall` and `\exists` designate universal and existential quantifiers.

5.2 Specification of our system

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 section 3.1).

As explained previously, we have derived a model from the real implementation 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 prost-conditions. For instance, for any object, its internal state space is specified by an invariant. The evolution of its state is specified as post-conditions associated with each method.

The implementation of local, global and environment properties were done in a systematical way. Local properties were described both as pre-conditions and invariants in the appliance objects. Service expected behaviors were described with post-conditions. Global properties were expressed as invariant in the service classes. Environment properties were expressed in the `Environment` class, as invariants. In our actual model, we have inserted 209 JML annotations (17 pre-conditions, 150 post-conditions, and 42 invariants).

6 The Testing Process

JML specifications can be used as an oracle for a test process. For improving confidence within the model, we have used a combinatorial testing approach. 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 (JML (2005)). When an operation is executed, three cases may happen. **All checks succeed**: the behavior 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 behavior 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 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 Cheon and Leavens (2002)). Each test case corresponds to a combination of parameters of the constructor and parameters of the method.

The LIG laboratory has developed Tobias (see du Bousquet et al. (2004); Dupuy-Chessa et al. (2005); Ledru et al. (2004)), a test generator based on combinatorial testing (see Cohen et al. (1996)). 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, for testing the Blind class, one can design the following schema:

$$\texttt{T-Blind} = \begin{cases} \texttt{T-Blind} = \texttt{Init ; BlindOp\^{}\{4..4\}} \text{ with} \\ \texttt{Init} = \{\texttt{Blind aBlind = new Blind()}\} \\ \texttt{BlindOp} = \{\texttt{aBlind.powerOn()}\} \cup \{\texttt{aBlind.powerOff()}\} \\ \cup \{\texttt{aBlind.Open()}\} \cup \{\texttt{aBlind.Close()}\} \end{cases}$$

`Init` is a set of only one instantiation. `BlindOp` is a set of 4 instantiations. The suffix `^{4..4}` means that the group is repeated 4 times. `T-Blind` is unfolded into 1*(4*4*4*4)= 256 test cases.

To validate our model, we have designed several test schemas corresponding to different phases in the validation process. First, each appliance was tested in isolation and in the context of the home. Thus, schemas similar to `T-Blind` were produced for each appliance. This phase 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 local properties. These INCONCLUSIVE verdicts were expected, each time some specific local properties were implemented.

This phase 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 our work on the 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 cases). 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.

The first set was dedicated to the service activations validation with respect to the different appliance states. To do that, the test prologue consisted of 3 or 4 different calls to one appliance. This was aimed at checking that the services could work correctly whatever the state of each appliance (taken independently). This allowed us to detect that some calls or checks were forgotten for some services. For instance, in the RelaxService, 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 close. 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...). To do that, the prologue consisted of applying different parameters to the environment attributes. This aimed at checking that the service could work correctly whatever the state of the environment. Of course, it was very easy to show how services were not consistent with the usage rules. For instance, the DVD Theater Service violate the environment property "*do not make loud voice or sound after 9 p.m.*" if it is activated after 9 p.m.

More than 30 test schemas were described and unfolded in the Tobias plug-in for Eclipse. Schemas have between 500 and 5000 test cases. Unfolding phase lasts at most two minutes for the biggest schemas[2]. Tests cases then were translated in the JUnit format and executed within the Eclipse environment. It took at most 500 seconds for the biggest sets of test cases. Approximately 10 errors were found (at the appliance and service levels).

7 Conclusion and perspectives

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 paper, we study the use of Java Modelling Language, to specify and validate an HNS implementation. The work is carried out in two steps. During the first step, we have derived a Java model of the real implementation. The model was then annotated and validated with a combinatorial testing approach. The actual model is composed of 14 appliances and 7 services. The specification is composed of 209 JML annotations (17 pre-conditions, 150 post-conditions, and 42 invariants). More than 30 test schemas were described and unfolded into a set of 500 to 5000 executable test cases within the Tobias tool.

A second step of the work consisted of the insertion of the annotations in the real implementation. This allows a continual monitoring of the application. This part is under work.

There are two main directions in which we want to work. The first one is the use of a prover as a complement to the testing approach. JML was chosen because several tools are available (see Leavens et al. (2000); van den Berg and Jacobs (2001); Burdy et al. (2003); Flanagan et al. (2002)). We first tried to use the JACK tool (see Burdy et al. (2003)). However, the current version of JACK does not support the JML version we used. We are currently evaluating ESC/JAVA (see Flanagan et al. (2002)).

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

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 a model is difficult to automate and is error-prone. In order to ease the process when introducing new services or appliances, we propose to follow a different strategy. Instead of directly implementing appliances and/or services in the real implementation, one would have to express them directly into the model. Thus, it would be possible to validate their use in the model (with test and proof), and to correct it (if needed). Then, it will be possible to transform the classes of the model into skeleton of classes for the 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.

Acknowledgement This research was partially supported by the 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).

References

Burdy, L., A. Requet, and J.-L. Lanet (2003). Java applet correctness: a developer-oriented approach. In *the 12th Int. FME Symposium*, Italy.

Cheon, Y. and G. Leavens (2002). A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP 2002*, Vol. 2474 of *LNCS*, pp. 231–255. Springer.

Cohen, D., S. Dalal, J. Parelius, and G. Patton (1996). The combinatorial design approach to automatic test generation. *IEEE Software 13*(5), 83–88.

du Bousquet, L. (1999). Feature interaction detection using testing and model-checking, experience report. In *World Congress on Formal Methods*, Vol. 1708 of *LNCS*, Toulouse, France, pp. 622–641. Springer Verlag.

du Bousquet, L., Y. Ledru, O. Maury, C. Oriat, and J.-L. Lanet (2004). A case study in JML-based software validation (short paper). In *19th Int. IEEE Conf. on Automated Sofware Engineering (ASE'04)*, pp. 294–297.

Dupuy-Chessa, S., L. du Bousquet, J. Bouchet, and Y. Ledru (2005). Test of the ICARE platform fusion mechanism. In *12th Int. Workshop on Design, Specification and Verification of Interactive Systems*, Vol. 3941 of *LNCS*, pp. 102–113. Springer.

Flanagan, C., K. R. M. Leino, L. M., G. Nelson, J. B. Saxe, and R. Stata (2002). Extended static checking for Java. In *Proc. of the ACM SIGPLAN 2002 Conf. on Programming language design and implementation*, pp. 234–245. ACM Press.

Geer, D. (2006). Nanotechnology: the growing impact of shrinking computers. *Pervasive Computing 5*(1), 7–11.

Hitachi. The electric household appliances revolution will change your lifestyle! `http://www.hitachi-cable.co.jp/en/hc-news/353/index_1.html`.

JML (2005). The JML Home Page. http://www.jmlspecs.org.

Kolberg, M., E. Magill, and M. Wilson (2003). Compatibility issues between services supporting networked appliances. *IEEE Communications Magazine 41*, 136–147.

Leavens, G., A. Baker, and C. Ruby (1999). JML: A notation for detailed design. In H. Kilov, B. Rumpe, and I. Simmonds (Eds.), *Behavioral Specifications of Businesses and Systems*, pp. 175–188. Kluwer.

Leavens, G. T., K. R. M. Leino, E. Poll, C. Ruby, and B. Jacobs (2000). JML: notations and tools supporting detailed design in Java. In ACM (Ed.), *OOPSLA 2000 Companion*. Also available as Tech. Report TR 00-15, Dep. of Computer Science, Iowa State Univ., Aug. 2000. ftp://ftp.cs.iastate.edu/pub/techreports/TR00-15/.

Ledru, Y., L. du Bousquet, O. Maury, and P. Bontron (2004). Filtering TOBIAS combinatorial test suites. In *Fundamental Approaches to Software Engineering (FASE'04)*, Vol. 2984 of *LNCS*, Barcelona, Spain. Springer.

Leelaprute, P., T. Tsuchiya, T. Kikuno, M. Nakamura, and K.-I. Matsumoto (2005). Describing and verifying integrated services of home network systems. In *12th Asia-Pacific Software Engineering Conf. (APSEC'05)*, Taiwan, pp. 549–560. IEEE.

Loke, S. W. (2003). Service-oriented device ecology workflows. In *First Int. Conf. on Service-Oriented Computing (ICSOC 2003)*, Vol. 2910 of *LNCS*, Italy, pp. 559–574. Springer.

Matsushita Electric Industrial Co., L. Kurashi net (jp). `http://national.jp/appliance/product/kurashi-net/`.

McMillan, K. L. (1993). *Symbolic Model Checking*. Kluwer Academic Publishers.

Meyer, B. (1992). Applying "Design by Contract". *Computer 25*(10), 40–51.

Nakamura, M., A. Tanaka, H. Igaki, H. Tamada, and K. Matsumoto (2006). Adapting legacy home appliances to home network systems using web services. In *Int. Conf. on Web Services (ICWS'06)*, pp. 849–858. IEEE.

Nakamura, M., A. Tanaka, H. Igaki, H. Tamada, and K. Matsumoto (2008). Constructing home network systems and integrated services using legacy home appliances and web services. *Int. Journal of Web Services Research*.

Papazoglou, M. P. and D. Georgakopoulos (2003). Special issue: Service-oriented computing. Introduction. *Commun. ACM 46*(10), 24–28.

Toshiba. Toshiba home network: Feminity. `http://www3.toshiba.co.jp/feminity/`.

van den Berg, J. and B. Jacobs (2001). The LOOP Compiler for Java and JML. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, Vol. 2031 of *LNCS*. Springer.

Yan, B., M. Nakamura, L. du Bousquet, and K. ichi Matsumoto (2007). Characterizing safety of integrated services in home network system. In *5th Int. Conf. On Smart Homes and Health Telematics (ICOST)*, Vol. 4541 of *LNCS*, Japan, pp. 130–140. Springer.