IEICE TRANS. INF. & SYST., VOL.E93–D, NO.4 APRIL 2010

PAPER

Modeling and Detecting Feature Interactions among Integrated Services of Home Network Systems

Hiroshi IGAKI[†a)] ***and*** **Masahide NAKAMURA**[†b)]**,** ***Members***

SUMMARY This paper presents a framework for formalizing and detecting feature interactions (FIs) in the emerging smart home domain. We first establish a model of home network system (HNS), where every networked appliance (or the HNS environment) is characterized as an object consisting of properties and methods. Then, every HNS service is defined as a sequence of method invocations of the appliances. Within the model, we next formalize two kinds of FIs: (a) appliance interactions and (b) environment interactions. An appliance interaction occurs when two method invocations conflict on the same appliance, whereas an environment interaction arises when two method invocations conflict indirectly via the environment. Finally, we propose offline and online methods that detect FIs before service deployment and during execution, respectively. Through a case study with seven practical services, it is shown that the proposed framework is generic enough to capture feature interactions in HNS integrated services. We also discuss several FI resolution schemes within the proposed framework.

key words: *feature interaction, home network system, online detection, integrated services*

1. Introduction

With the rapid diffusion of ubiquitous technologies, general household appliances are being equipped with smart processors and network interfaces. Smart home appliances, such as TVs, DVDs, speakers, air-conditioners, lights, doors and sensors, are connected to a LAN at home, comprising a *home network system* (HNS). Several services for HNS are being developed and provided (e.g., [1]–[4]). Currently, most conventional services use a single or a small set of appliances. However, the main advantage of HNS lies in the *integration* (or *orchestration*) of different appliances together [5]. The integration would yield more value-added and sophisticated services, which we call *HNS integrated services*.

We here introduce two examples. In the examples, we assume that a DVD player, a TV, a 5.1ch speaker, blinds, a light, an illuminometer and a door are installed in an experimental room. Also, the integrated services share these appliances.

DVD Theater Service: Integrating the DVD player, the TV, the speaker, the blinds and the light, the service allows a user to watch movies in a theater-like atmosphere just within a single operation. The DVD player is switched on, the TV is turned on in DVD mode, the blinds are closed, the brightness of the light is minimized, 5.1ch mode of the speaker is selected, and the sound volume of the speaker is automatically adjusted.

Coming Home Light Service: Integrating the door sensor, the light and the illuminometer, the service supports a user in entering the house. When the door sensor notices that the user has come home, the light is automatically turned on. Then, the brightness of the light is adjusted to the optimal value based on the current degree obtained from the illuminometer.

As the number of appliances grows, many integrated services will be provided to meet various customers' needs. However, combined use of multiple services may cause functional conflicts, which significantly decreases the quality of services and may raise serious safety-critical issues. The conflicts among services are generally known as the *feature interaction problem* (FI) [6]. For instance, the above two services cause FIs.

FIs between DVD Theater & Coming Home Light: Suppose that a user *A* is watching movie with the DVD Theater. Simultaneously, suppose that another user *B* comes home, activating the Coming Home Light. Then the following FIs occur.

FI-(a): Although the DVD Theater minimizes the brightness of the light, the Coming Home Light sets the brightness comfortable for *B*. This may ruin *A*'s experience of watching the movie in a good atmosphere.

FI-(b): If the blinds are closed (by DVD Theater) immediately after the lights read the degree from the illuminometer (by Coming Home Light), the lights may fail to set the optimal illumination. This is because the blinds make the room darker.

FIs have been studied for years mainly in the telecommunication services (e.g., [7], [8]). However, little research has been done on this emerging domain of HNS (see Sect. 6.4).

As more and more appliances and services are provided in the future, the FI problem in the HNS will be more and more critical. For instance, the FI among *RemoteKeyLock* service and *FireEvacuation* service may cause a serious accident, where people are locked in the room in case of fire. Also, as the number of services increases, the number of

Manuscript received May 21, 2009.
Manuscript revised November 23, 2009.

[†]The authors are with the Faculty of Computer Science and Systems Engineering, Kobe University, Kobe-shi, 657–8501 Japan.
a) E-mail: igaki@cs.kobe-u.ac.jp
b) E-mail: masa-n@cs.kobe-u.ac.jp

DOI: 10.1587/transinf.E93.D.822

such potential FIs grows combinatorially. Thus, it is essential to establish a solid foundation to manage the FI problem systematically.

The goal of this paper is to propose a framework for formalizing FIs in HNS and to develop detection methods of such FIs. We first establish a formal model of HNS in an object-oriented fashion. Specifically, we model each appliance as an *object* consisting of *properties* and *methods*. The behavior of the appliance is simply characterized by *pre-condition* and *post-condition* in each method, prescribing rules for state transitions. We similarly define an object for the *home environment*. Next, we define each integrated service as a sequence of *invocations* of the appliance methods.

Within the model, we formalize the FI as a pair of method invocations m and m' that are incompatible with each other. For this, we define two types of FIs: *appliance interaction* and *environment interaction*. Intuitively, the appliance interaction refers to a situation where m and m' directly conflict on the same appliance. The above **FI-(a)** corresponds to an appliance interaction, where two methods, say, `Light.setBrightness(5)` and `Light.setBrightness(100)` conflict on `Light` object. On the other hand, the environment interaction is that m and m' do not share the same appliance but they conflict indirectly via the home environment. The above **FI-(b)** is an environment interaction, where two methods, say, `Blinds.close()` and `Illuminometer.getBrightness()` conflict on `Brightness` property of the home environment[†].

Based on the framework, we develop *offline* and *online* detection methods for FIs in HNS. The offline detection tries to detect all *potential* FIs, before deploying services. On the other hand, the online method detects FIs only when they occur during runtime. In a case study with seven practical services, we detected as many as 67 potential FIs (43 appliance interactions and 24 environment interactions).

The digest version of this article was published as a conference paper in ICFI'05 [9]. Changes were made to this version, most significantly the revision of definitions of HNS and integrated services (Sect. 3). We also added the on-line FI detection, case study, and evaluation parts (from Sect. 5.2 to the end). We believe that these new results clarify the applicability and limitations of the proposed method in practical settings.

2. Preliminaries

2.1 Home Network System (HNS)

A HNS consists of one or more *networked appliances* connected to a LAN at home. Each networked appliance has a set of *control APIs*, so that the user or external software agents can control the appliance via the network (e.g., [10], [11]). For example, every air-conditioner should have APIs for controlling power and temperature settings. A speaker will have APIs for volume and channel (2ch or 5.1ch). To process the API calls, every appliance generally has embedded units including a processor and a storage. If a HNS contains multiple appliances in the same kind, we regard them as *independent* objects. For example, if there are four lights in the room, we put four objects `Light1`, `Light2`, `Light3` and `Light4` in our model.

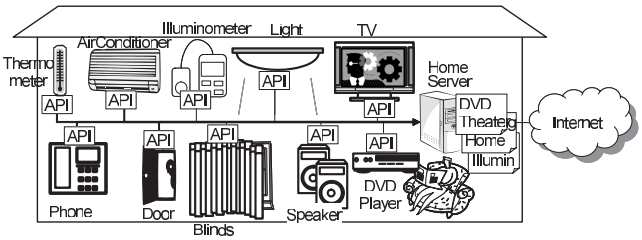

Fig. 1 A home network system.

Figure 1 shows an example of HNS, which consists of ten appliances and a *home server*. The home server typically plays a role of gateway to the external network. It also works as an application server, where the HNS applications are installed [12]. As seen in Fig. 1, every HNS integrated service is implemented as a software application that invokes the APIs according to a certain control flow. The services are supposed to be installed in the home server. When a user requests to activate an integrated service, the home server launches the service and invokes appliance APIs defined in the service.

Communications among the appliances and the home server are performed by an underlying *HNS protocol*. Various standards for the HNS protocols have been proposed, such as X-10 [13], HAVi [14], ECHONET [11] and UPnP [15]. In this paper, we aim to propose a framework independent of specific protocols or underlying platforms. Therefore, abstracting the network layer or below, we assume that a certain mechanism (e.g., middleware) for handling the communications is available in a given HNS.

2.2 Examples of HNS Integrated Services

For more comprehensive discussion, we introduce examples of HNS integrated services taken from actual HNS products [1]. In the examples, we assume the HNS in Figure 1, where ten appliances (a DVD player, a TV, a speaker, a light, an illuminometer, a door with a sensor, a telephone, an air-conditioner, a thermometer and blinds) are installed.

Each example shows a *service scenario* labeled by SS_i ($1 \le i \le 7$).

SS_1: Auto-TV Service - The TV is turned on, the speaker's channel is set to 2ch, and the volume of the speaker is automatically adjusted for the TV mode.

SS_2: DVD Theater Service - The DVD player is switched on, the TV is turned on in DVD mode, the blinds are closed, the brightness of the light is minimized, 5.1ch mode of the speaker is selected, and the volume of the

[†]Here, if the DVD Theater Service and the Coming Home Light Service use different light appliances, the environment interaction occurs.

speaker is automatically adjusted.

SS_3**: Coming Home Light Service -** When the door (sensor) notices that the user has come home, the light is automatically turned on. Then, the illumination of the light is adjusted to the optimal value based on the current degree obtained from the illuminometer.

SS_4**: Coming Home Air Conditioning Service -** When the door sensor notices that the user has come home, the air-conditioner is turned on, and its temperature setting is adjusted to the optimal based on the current degree of temperature provided by the thermometer.

SS_5**: Ringing and Mute Service -** When the telephone rings, the volume of the speaker is muted.

SS_6**: Blinds Service -** When sunlight is available, the blinds are opened.

SS_7**: Sleep Service -** When the user goes to bed or goes outside, all appliances are turned off.

2.3 Assumptions

Here, we impose the following assumptions to make the scope of our framework clearer.

Assumption A1: All appliances used by the integrated services are located in an experimental room.

Assumption A2: For every appliance name *app* used in the integrated service, there exists exactly one actual appliance *d* corresponding to *app*.

Assumption A3: We formalize and detect feature interactions between every *pair* of services only. Three-way (or more) interactions, which occur only within more than two services, are beyond this paper.

Assumption A1 implies that all the appliances share the same environment.

Assumption A2 indicates that the one-to-one relationship between the appliance references in the service and the real appliances should be given in advance. For instance, if the name DVD is used in a service, there must exists an actual DVD player uniquely specified in the HNS. If a service contains two lights Light1 and Light2, the concrete lights, for instance, a ceiling light and a downlight are respectively assigned to Light1 and Light2. Such *static binding* between the appliance references and the real appliances can be seen in the current HNS (e.g., [1], [11]). So, we consider that Assumption A2 is reasonable to a certain extent. However, there exists more flexible HNS that allows dynamic bindings [16]. For such an advanced HNS, the assumption must be relaxed. More detailed discussion about the dynamic binding will be given in Sect. 6.5.

In Assumption A3, we suppose that we aim to detect feature interactions within every pair of two services. Theoretically speaking, there exists three-way interactions which occur only when three or more services are executed. However, we assume that these are out of scope of this paper.

3. Formalizing HNS and Integrated Services

3.1 Key Idea

Our key idea to formalize the HNS and integrated service is to capture every appliance as a *self-contained* object. In general, every appliance has an *internal state* and *behaviors*. For instance, an air-conditioner has at least two states `ON or OFF` with respect to power. Also, the air-conditioner has states each of which corresponds to a specified temperature setting. The internal state is changed by executing the APIs (see Sect. 2.1), which determines the behavior of the air-conditioner. For instance, executing an API `setTemperature(23)` changes the current temperature setting to 23 degree. Thus, it is quite natural to regard every appliance as an *object* consisting of *properties* and *methods*. The properties are primary attributes characterizing the internal state of an appliance. In the above small example, properties of the air-conditioner would be `Power` and `TempSet`.

On the other hand, the methods correspond to the APIs of the appliance. The details of each API are usually encapsulated in a vendor-specific implementation. Therefore, various abstraction levels can be considered to model the corresponding method. In this paper, we simply characterize each method by a pair of pre-condition and post-condition to achieve the generality of the model. Intuitively, the pre-condition and the post-condition respectively work as a *guard* and an *action*. For instance, the API `setTemperature(23)` is modeled as a method with a pre-condition: `[Power=='ON']` and a post-condition: `[TempSet==23]`. It is interpreted that when the power of the air-conditioner is ON, if the API is executed, then the temperature setting becomes 23.

Based on the key idea, we give a formal definition of a HNS and integrated services in the following subsections.

3.2 Networked Appliances

Definition 3.1 (Property): A property is defined as a pair of *property name p* and *property type tp*. *p* can take a value *a* that must be of type *tp*.

Definition 3.2 (State): Let $P = \{p_1, ..., p_n\}$ be a given set of properties. A *state s* over *P* is defined as $\langle a_1, ..., a_n \rangle$, where a_i is the (current) value of p_i.

Table 1 summarizes properties for the ten appliances introduced in Sect. 2.2[†]. In the table, a property type is described as an enumeration type or an integer type with lower and upper bounds. For instance, the air-conditioner has properties `Power` and `TempSet`, where `tPower = {ON, OFF}` and `tTempSet = {18...28}`. This implies that property `Power` takes a value of `ON` or `OFF`, whereas `TempSet` takes an integer between 18 and 28. `<ON, 23>` represents a state of the

[†]Due to limited space, properties irrelevant to the SS_1 to SS_7 are omitted from the table.

Table 1 Appliance properties.

Appliance Name	Appliance Property		
	Name	Type Name	Type
AirConditioner	Power	tPower	{ON,OFF}
	TempSet	tTempSet	{18...28} (deg.)
Thermometer	Power	tPower	{ON,OFF}
	CurrentTemp	tTemp	{0...40} (deg.)
Speaker	Power	tPower	{ON,OFF}
	Input	tInput	{TV,DVD}
	Channel	tChannel	{2,5.1}
	VolumeSet	tVolumeSet	{0...50} (dB)
Light	Power	tPower	{ON,OFF}
	BrightSet	tBrightSet	{0...600} (lx)
Illuminometer	Power	tPower	{ON,OFF}
	CurrentBright	tBright	{0...1000} (lx)
Door	DoorStatus	tDoorStatus	{Open,Close}
	Power	tPower	{ON,OFF}
Phone	PhoneStat	tPhoneStat	{Received, Calling, Connected, Waiting}
DVD player	Power	tPower	{ON,OFF}
TV	Power	tPower	{ON,OFF}
	Input	tInput	{TV,DVD}
Blinds	Power	tPower	{ON,OFF}
	BlindsStat	tBlindsStat	{Open,Close}

air-conditioner, where the power is currently ON and the temperature setting is 23 degree.

Next, we introduce a *property formula* to construct the pre/post-conditions, which is a conjunction of Boolean formulas over properties.

Definition 3.3 (Property Formula): Let $P = \{p_1, p_2, ..., p_n\}$ be a given set of properties. A formula $c = f_{p_1} \wedge f_{p_2} \wedge ... \wedge f_{p_n}$, where f_{p_i} is any logical formula with respect to p_i, is called a *property formula* over P. $Cond_P$ denotes a set of all property formulas over P. For $c = f_{p_1} \wedge f_{p_2} \wedge ... \wedge f_{p_n}$, $\prod_{p_i}(c) = f_{p_i}$ is called a *projection* of c with respect to property p_i.

For a given state $\langle a_1, ..., a_n \rangle$, a property formula c is evaluated to be true or false, according to the (current) state. Let us consider the example of the air-conditioner. Then, c = [`Power=='ON'` $\wedge$ `TempSet` > 20] is a property formula, which is supposed to become true for a state `<ON, 23>`. Also, $\prod_{Power}(c)$ = [`Power=='ON'`], which is a projection of c onto `Power`.

Note in Definition 3.3 that f_{p_i} doesn't include any properties other than p_i, which might limit the expressive power of the property formula. However, we found that this was not a fatal limitation in modeling many practical appliances, including ten appliances introduced in Sect. 2.2. Specifically, within any single appliance we did not find such properties p_i and p_j that p_i and p_j are strictly dependent on each other, and that direct comparison and/or operation between p_i and p_j are required. Further discussion on the expressivity of the property formula will be left for our future work.

Definition 3.4 (Networked Appliance): A *networked appliance* d is defined as a 4-tuple $d = (P_d, M_d, Pre_d, Post_d)$, where

- $P_d = \{p_1, ..., p_n\}$ is a set of all *properties* of d.
- $M_d = \{m_1, ..., m_k\}$ is a set of all *methods* of d.
- $Pre_d : M_d \rightarrow Cond_{P_d}$ is a *pre-condition function* which maps each method $m_i \in M_d$ to a property formula over P_d. m_i can be executed only when $Pre_d(m_i)$ is true.
- $Post_d : M_d \rightarrow Cond_{P_d}$ is a *post-condition function* which maps each method $m_i \in M_d$ to a property formula over P_d. $Post_d(m_i)$ becomes true immediately after m is executed.

A property $p \in P_d$ (or a method $m \in M_d$) of an appliance d is denoted by $d.p$ (or $d.m$, respectively).

Table 2 shows methods of the ten appliances in our example. Each method is modeled by pre/post-conditions (R_e and W_e will be defined later). In the table, '*' denotes a *don't care* value, e.g., the condition `CurrentTemperature == *` becomes true as long as a certain value of the property is available. For instance, the air-conditioner has a method `setTemperature(tTempSet temp)`, indicating that "When the power is on, if the method is executed, the temperature is set to the value specified by `temp`".

To make such dynamics clearer, we define *semantics* of the appliance model.

Definition 3.5 (Appliance Semantics): Let $d = (P_d, M_d, Pre_d, Post_d)$ be an appliance and $s = \langle a_1, ..., a_n \rangle$ be a state over P_d. For a method $m \in M_d$, we say that m is *enabled* under s iff $Pre_d(m)$ is true for s. When m is enabled under s, m can be executed. If m is executed, s is changed to the *next state* $s' = \langle a'_1, ..., a'_n \rangle$ so that $Post_d(m)$ becomes true for s'. Properties that do not appear in $Post_d(m)$ keep their values unchanged.

Let us consider an execution of a method `setTemperature(25)` of the air-conditioner under state s_1 = `<ON,23>`. Then, the method is enabled under s_1 since the pre-condition `[Power=='ON']` is true. If the method is executed, s_1 moves to the next state s_2 = `<ON,25>`, as specified in the post-condition `[TempSet==25]`, where the formal parameter `temp` is substituted by 25.

3.3 Environment

Appliances in a HNS share a home space with each other. Hence, the appliances are tightly coupled with the *environment* of the home. For instance, the air-conditioner tries to keep a comfortable room temperature, which implicitly updates the temperature of the environment. Also, the thermometer refers to the current temperature of the environment. Thus, the air-conditioner and the thermometer are indirectly connected via the environment.

The environment of the home is an important factor in feature interaction analysis (cf. [17], [18]). In this paper, we formalize the environment as a *global object* which can be referred to or updated by all appliances in the HNS. Specifically, an environment object has a set of *global properties* such as temperature, brightness and sound volume.

When a method m of an appliance is executed, these environment properties are indirectly referred to or updated

Table 2 Appliance methods.

ApplianceName	ApplianceMethod	Pre_d	$Post_d$	R_e	W_e
AirConditioner	setPower(tPower onoff)		Power==onoff		
	setTemperature(tTempSet temp)	Power=='ON'	TempSet==temp		Temp
Thermometer	setPower(tPower onoff)		Power==onoff		
	getTemperature()	Power=='ON' ∧ CurrentTemp==*		Temp	
Speaker	setPower(tPower onoff)		Power==onoff		
	setInput(tInput spInput)	Power=='ON'	Input==spInput		
	setChannel(tChannel spChannel)	Power=='ON'	Channel==spChannel		
	setVolume(tVolumeSet spVolume)	Power=='ON'	VolumeSet==spVolume		Volume
TV	setPower(tPower onoff)		Power==onoff		
	setInput(tInput tvInput)	Power=='ON'	Input==tvInput		
DVD	setPower(tPower onoff)		Power==onoff		
Light	setPower(tPower onoff)		Power==onoff		
	setBrightness(tBrightSet lx)	Power=='ON'	BrightSet==lx		Bright
Illuminometer	setPower(tPower onoff)		Power==onoff		
	getBrightness()	Power=='ON' ∧ CurrentBright==*		Bright	
Door	getDoorStatus()	Power=='ON' ∧ DoorStatus==*			
Phone	ringing()	PhoneStat=='Recieved'	PhoneStat=='Calling'		Volume
	connected()	PhoneStat=='Calling'	PhoneStat=='Connected'		Volume
Blinds	setPower(tPower onoff)		Power==onoff		
	setGate(tBlindsStat gateStat)	Power=='ON'	BlindsStat==gateStat		Bright,Temp

by m. For the environment, we adopt a loose modeling such that we only care if m *reads* or *writes* some environment properties. This loose modeling is because the impact of a method to the environment properties are not as direct and explicit as the impact to the appliance properties[†]. Therefore, we cannot specify strict pre/post conditions with the environment properties before/after the execution of m.

Definition 3.6 (Environment): Let $D = \{d_1, d_2, ..., d_k\}$ be a set of all appliances deployed in the HNS. Also, let $M = \cup_{d_i \in D} M_{d_i}$ be a set of all methods of all appliances. Then, an environment e is defined as a tuple $e = (P_e, R_e, W_e)$, where

- P_e is a set of all environment properties.
- R_e is an *environment read* function $M \rightarrow 2^{P_e}$, which maps each method $m \in M$ into a set of environment properties that are read by m.
- W_e is the *environment write* function $M \rightarrow 2^{P_e}$, which maps each method $m \in M$ into a set of environment properties that are written by m.

The last two columns of Table 2 represent the environment read/write functions for every method. In our example, we assume the following environment properties:

`Temperature(Temp):` the degree of temperature of the room.

`Brightness(Bright):` the intensity of brightness in the room.

`Volume:` the sound volume in the room.

For instance, `Temperature` is designated in W_e(`AirConditioner.setTemperature(...)`), which implies that setting the temperature of air-conditioner can write (update) the current temperature degree of the room.

3.4 HNS and Integrated Services

We are now ready to formalize a HNS. A HNS consists of a set of appliances and an environment.

Definition 3.7 (Home Network System): A home network system is defined as $HNS = (D, e)$, where

- $D = \{d_1, d_2, ..., d_n\}$ is a set of appliances.
- $e = (P_e, R_e, W_e)$ is an environment where the HNS is deployed.

According to the definition, Tables 1 and 2 complete the formalization of our example HNS.

As mentioned in Sect. 2.1, an integrated service can be implemented by a set of invocations of APIs (i.e., appliance methods) with a control flow. In this paper, we define an integrated service as a *sequence* of appliance methods.

Definition 3.8 (HNS Integrated Service): Let $HNS = (D, e)$ be a given HNS. Then, an *integrated service* ss_i is defined as $ss_i = d_{i1}.m_{i1}\ ;\ d_{i2}.m_{i2}\ ;\ ...\ ;\ d_{ik}.m_{ik}$, where $d_{ij} \in D, m_{ij} \in M_{d_{ij}}$, and ';' is a sequential operator.

As described in Sect. 2.1, the integrated services are installed on a home server (see Fig. 1). Thus, all the methods in ss_i are supposed to be executed by the home server.

It is our design choice to define an integrated service by such a simple control flow without branch or loop. This is because the service designer could be a home user who cannot afford the complex service logic.

Indeed the HNS experts (e.g., vendors) may want to design and validate more sophisticated services with complex control flows (i.e., loops and branches), rather than simple sequences. In such a case, one can use the program analysis techniques (e.g., [19]) that "unfold" the loops and branches into a set of execution sequences. Thus, the proposed method can deal with each derived sequence as an integrated service.

Figure 2 shows implementations of the example service

[†]For instance, the temperature setting of an air-conditioner is not always equal to the temperature of a room.

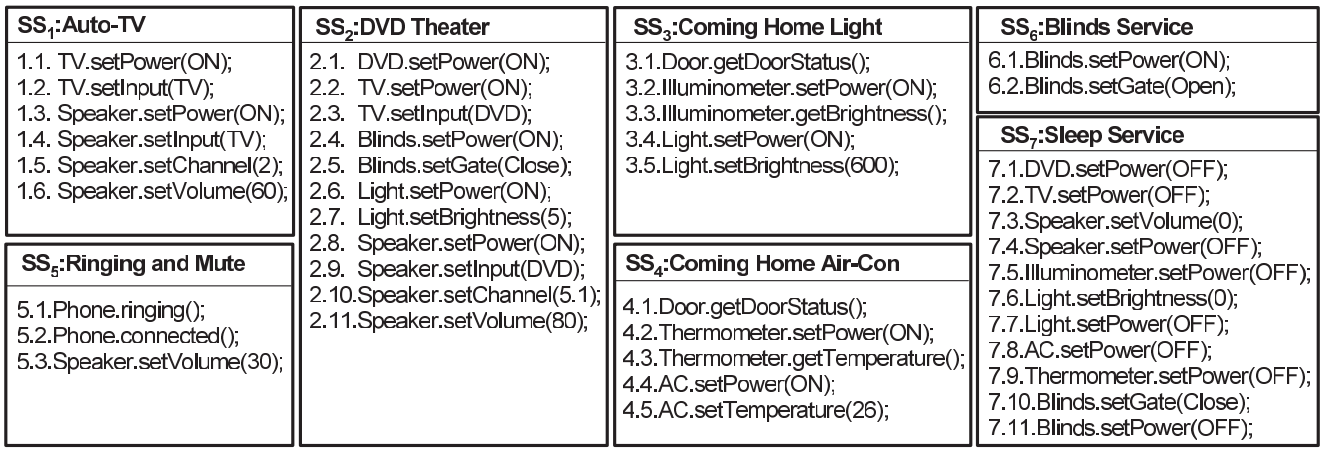

Fig. 2 Integrated services SS_1 to SS_7.

scenarios from SS_1 to SS_7 discussed in Sect. 2.2. For convenience, we put an index number for each method. Let us take SS_2: DVD Theater. First, the DVD player is turned on. Then, the TV is turned on and its input mode is set to DVD mode. Next, the blinds are closed and brightness of the light is minimized. Finally, the speaker is configured for the DVD setting. This sequence realizes the requirement of the DVD theater service. As also seen in other services, our model can express a quite reasonable range of practical integrated services despite its simple control flow.

4. Modeling Feature Interaction Problem

Even if every integrated service achieves its requirement, activating multiple services simultaneously may result in an unexpected conflict, which is called *feature interaction* (FI). Based on the proposed model, we formalize two kinds of FIs among HNS integrated services, specifically *appliance interactions* and *environment interactions*. Our basic idea is to capture an FI as a conflict of *appliance methods* that are incompatible with each other.

4.1 Appliance Interactions

When multiple integrated services simultaneously invoke incompatible methods m_1 and m_2 of a *common* appliance d, m_1 and m_2 conflicts. This causes an FI on the appliance d, which we formalize as an *appliance interaction*.

Definition 4.1 (Appliance Interactions): Let $HNS = (D, e)$ be a given HNS, and ss_i and ss_j be a pair of integrated services. Suppose that for an appliance $d \in D$, ss_i contains a method $d.m_i$ and ss_j contains a method $d.m_j$. We say that ss_i and ss_j cause an *appliance interaction* on d iff at least one of the following conditions is satisfied:

Condition D1: There exists an appliance property $p \in P_d$ such that $\prod_p Post(m_i) \wedge \prod_p Post(m_j) = \bot$ (unsatisfiable), or

Condition D2: There exists an appliance property $p \in P_d$ such that $\prod_p Post(m_i) \wedge \prod_p Pre(m_j) = \bot$ (unsatisfiable).

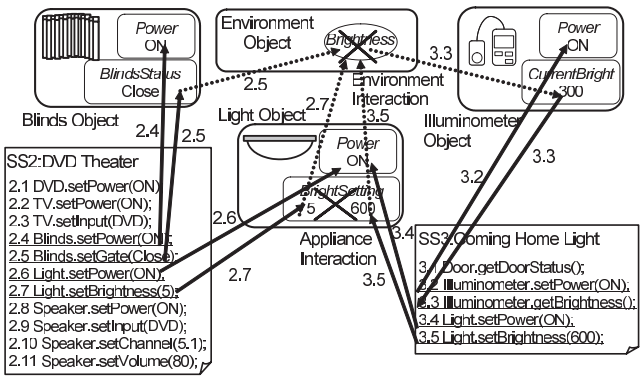

Fig. 3 Interactions between SS_2 and SS_3.

Condition D1 characterizes the appliance interaction as the unsatisfiability between two post-conditions. On the other hand, Condition D2 represents the unsatisfiability between a pre-condition and a post-condition.

Suppose that DVD-Theater (SS_2) and Coming Home Light (SS_3) in Fig. 2 are simultaneously executed. Figure 3 depicts the situation. The figure contains four objects commonly shared by the two services. An arrow represents an update (or a reference) to a property performed by the method invocation. As shown in Fig. 3, SS_2 invokes `Light.setBrightness(5)`, while SS_3 invokes `Light.setBrightness(600)`.

Thus, it can be seen a conflict in which value of brightness should be set to the light. This is exactly **FI-(a)** in Sect. 1. According to Table 2, **FI-(a)** is characterized by two unsatisfiable post-conditions; (`[BrightSet==5]` $\wedge$ `[BrightSet==600]`) $= \bot$ (unsatisfiable), as defined in Condition D1. Note that projection $\prod_p$ allows *property-wise checking* of the satisfiability.

Let us introduce another example with DVD Theater (SS_2) and Sleep (SS_7). `TV.setInput(DVD)` of SS_2 requires in the pre-condition that the TV is switched on

([power==ON]). However, TV.setPower(OFF) of SS_7 updates the value of the property power into OFF as defined in its post-condition, which *disables* TV.setInput(TV). As a result, SS_2 is suspended. This FI occurs since a pre-condition of SS_2 and a post-condition of SS_7 are unsatisfiable simultaneously. Condition D2 covers such cases.

Note that Condition D1 and D2 can be tested even if a HNS contains multiple appliances in the same kind. For instance, suppose that a HNS contains two lights Light1 and Light2, and that one service executes Light1.setPower(ON) and another service does Light2.setPower(OFF). In this case, these methods do not cause appliance interaction, since Light1.power and Light2.power are independent.

4.2 Environment Interactions

Even if methods m_1 and m_2 do not share the common appliance, FIs may occur indirectly via the environment. The environment interaction arises when m_1 and m_2 try to access common environment properties.

Definition 4.2 (Environment Interactions): Let $HNS = (D, e)$ be a given HNS, and ss_i and ss_j be a pair of integrated services.Suppose that for a pair of appliances $d, d' \in D$ $(d \neq d')$, ss_i contains a method $d.m_i$ and ss_j contains a method $d'.m_j$. We say that ss_i and ss_j cause an *environment interaction* iff at least one of the following conditions is satisfied:

Condition E1: $W_e(m_i) \cap W_e(m_j) \neq \phi$, or
Condition E2: $R_e(m_i) \cap W_e(m_j) \neq \phi$.

Condition E1 reflects a race condition between two "writes" on the common environment properties. Condition E2 specifies non-interchangeable "read" and "write" on the common environment properties.

Let us see the environment interaction among DVD-Theater (SS_2) and Coming Home Light (SS_3) using Fig. 3. Illuminometer.getBrightness() in SS_3 reads the environment property Brightness to obtain the current brightness of the room. On the other hand, Blinds.setGate(Close) in SS_2 makes the room darker, that is, the method writes environment property Brightness. If the two methods occur simultaneously, SS_3 may fail to set optimal brightness to the light. This is exactly **FI-(b)** in Sect. 1. According to Table 2, this FI occurs since the two methods access common environment property Brightness. That is, R_e(Illuminometer.getBrightness()) $\cap$ W_e(Blinds.setGate(Close)) = {Brightness} $\neq$ ϕ. Hence, Condition E2 applies to the FI.

Another interesting example is the FI among the Coming Home Air-Con (SS_4) and the Blinds (SS_6). Although no appliance is shared by the two services, the services cause environment interactions. Since opening the blinds could heat the room, air-conditioning may not work as expected. This FI can be also characterized by Condition E1.

The definition of the environment interactions would be strengthened by introducing the *direction of the effects* to the environment properties, so that only methods with the opposite environmental effects cause an interaction. However, even if the direction is the same, two methods may yield excessive effects beyond the user's intension. Thus, whether or not each instance of interaction is *desirable* heavily depends on *user's requirements*. Therefore, we here aim to define and detect the environment interaction in a *broad sense*, simply by using the read/write functions. We assume that the interpretation and the resolution of the interactions are performed after the detection process, which are beyond this paper.

5. Detecting Feature Interactions

Based on the formalization, we develop a method that detects FIs among given integrated services. There are two methods for the FI detection: *offline* and *online* methods. The offline detection is to detect all possible FIs for all pairs of given integrated services, assuming that the detection process is conducted before deploying the integrating services. On the other hand, the online detection is to conduct detection process during runtime, which dynamically manages any FI when it actually occurs.

5.1 Offline FI Detection

We start with the *offline* detection method. The offline detection is to identify all the *potential* FIs before actually deploying the services in the home server.

Offline FI Detection
Input: A home network system $HNS = (D, e)$, and a set of integrated services $ss_1, ss_2, ..., ss_n$.
Output: All possible pairs of appliance methods that cause appliance or environment interactions.
Procedure: For any pair of methods m and m' contained in ss_i and ss_j, respectively, evaluate Conditions D1 and D2 for appliance interactions, and Conditions E1 and E2 for environment interactions.

Proposition P1: For the given $HNS = (D, e)$, the offline FI detection can detect all appliance interactions and environment interactions between every pair of ss_i and ss_j $(i \neq j)$, which is independent of the contents of service scenarios or the combination of appliances.
Proof: According to Assumption A1, the environment object e is uniquely determined, and every appliance d commonly shares e. According to Assumption A2, for every appliance $d \in D$, there exists a real appliance in HNS corresponding to d.

By Definition 3.4, every method m is defined by a pre-condition and a post-condition. Hence, in the detection procedure, for a pair m and m' of methods in ss_i and ss_j respectively, it is possible to evaluate Conditions D1 and D2. According to Definition 4.1, satisfying either $D1$ or $D2$ leads to the appliance interaction. Note that the evaluation of D1 and D2 does not depend on specific appliances or methods.

Since D is a finite set and the number of service scenarios is finite, the same evaluation can be applied to every pairs of methods within every pair of services. Thus, all appliance interactions can be detected.

By Definition 3.6, the influence of each appliance method m to e is defined as $R_e(m)$ and $W_e(m)$. Hence, in the detection procedure, for a pair m and m' of methods in ss_i and ss_j respectively, it is possible to evaluate Conditions E1 and E2. According to Definition 4.2, satisfying either $E1$ or $E2$ leads to the environment interaction. By the same discussion, all environment interactions can be detected.

Finally, according to Assumption A3, we should note that we do not detect three-way interactions. Thus, the detection procedure conforms Proposition P1.

5.2 Online FI Detection

Even if the offline detection says that services ss_i and ss_j cause an FI, the FI does not actually occur unless ss_i and ss_j are not executed simultaneously.

Therefore, we propose the online FI detection method based on a *service life cycle* of the integrated services.

Definition 5.1 (Service Life Cycle): Let ss be any integrated service. The *life cycle* of ss is defined by a state transition machine L_{ss}, consisting of two states: `in operation` and `terminated`. When ss is deployed on the HNS, L_{ss} is generated and initialized at `terminated` state. On receiving an execution request of ss, ss executes its appliance methods and L_{ss} changes the state to `in operation`. When a termination request is received, ss stops its execution and the state moves back to `terminated` state. We say that ss is *terminated* (or *in operation*) iff L_{ss} is in `terminated` (or `in operation`, respectively) state.

We suppose that the service life cycle is managed within the home server, and that an execution (or termination) request is triggered by an event such as user's request, a sensor event, a timer event, etc.

The online FI detection aims to identify FIs during runtime just before the FIs occur. For this, we extend Definitions 4.1 and 4.2.

Definition 5.2 (Interactions During Runtime): Let $HNS = (D, e)$ be a given HNS, and ss_i and ss_j be a pair of integrated services. Then, we say that ss_i and ss_j cause an appliance (or environment) interaction *during runtime*, iff ss_i and ss_j cause an appliance interaction (or environment interaction, respectively), and both ss_i and ss_j are in operation.

To achieve the online detection, we need to check which service is in operation. To do this, we employ a special module called *FI detection module*. This module uses a database called *method pool* to identify the methods in operation. When a service is in operation, the methods of the service are registered in the method pool through the module. When the service terminates, the methods are removed from the pool. When a new service is activated during the method pool is not empty, checking FIs between each method of the new service and the ones in the pool is performed.

Online FI Detection

Input: A home network system $HNS = (D, e)$, and a set of integrated services $ss_1, ss_2, ..., ss_n$.

Output: All possible pairs of appliance methods that cause appliance or environment interactions during runtime.

Procedure:

Step 0: FI detection module $FIDM$ initializes the method pool MP to be empty.

Step 1: A Home Server HS accepts activation of a service $ss_i = m_{i1}; m_{i2}; ...; m_{ik}$.

Step 2: HS Consult $FIDM$, and $FIDM$ checks whether pooled methods in MP are.

- If MP is empty, go to Step 4.
- If MP is not empty, i.e., MP already contains methods $m'_{j1}, m'_{j2}, ..., m'_{jl}$, go to Step 3.

Step 3: For each pair of m_{ix} and m'_{jy} ($1 \leq x \leq k$, $1 \leq y \leq l$), $FIDM$ evaluates Conditions D1 and D2 (or E1 and E2) for appliance (or environment, respectively) interactions.

- If any interactions are found, arbitrary FI resolution scheme (we introduce some schemes in 6.3) is executed. The scheme decides which methods should be executed. Based on the result of FI resolution, FIDM registers and removes appliance methods in MP. Then, the HS executes the methods one-by-one, and go to Step 1.
- If no interaction is found, go to Step 4.

Step 4: $FIDM$ registers all $m_{ix} (1 \leq x \leq k)$ in MP. Then, HS executes the methods one-by-one. Go to Step 1.

5.3 Case Study

5.3.1 Offline Detection

We have conducted a case study with offline interaction detection. For this experiment, we have implemented a tool using Java (J2SE1.4.2), comprising about 1,500 lines of code with 5 classes. The tool provides four kinds of operations, checkD1(m, m'), checkD2(m, m'), checkE1(m, m'), checkE2(m, m'), which respectively checks Conditions D1, D2, E1, and E2 for a given pair of appliance methods m and m'. Using these four operations, the tool detects all FIs for given service scenarios SS and SS'.

In this case study, we took a HNS defined in Tables 1 and 2, and seven integrated services $SS_1, SS_2, ..., SS_7$ shown in Figure 2, as the input.

Table 3 (a) shows a total 43 appliance interactions detected, whereas Table 3 (b) enumerates 24 environment interactions. Each entry represents a set of pairs of methods causing FIs (each index corresponds to the one in Figure 2). It can be seen in Table 3 (a) that two appliance interactions explained in Sect. 4.1 are well detected as the method pairs (2.7, 3.5) and (2.3, 7.2). Also, two environment interactions

Table 3 Results of offline interaction detection.

(a) Appliance Interactions

	SS1	SS2	SS3	SS4	SS5	SS6	SS7
SS1		(1.2,2.3)(1.4,2.9) (1.5,2.10)(1.6,2.1			(1.6,5.3)		(1.1,7.2)(1.2,7.2)(1.3,7.4)(1.4,7.4)(1.5,7.4) (1.6,7.3)(1.6,7.4)
SS2			(2.7,3.5)		(2.11,5.3)	(2.5,6.2)	(2.1,7.1)(2.2,7.2)(2.3,7.2)(2.4,7.9)(2.5,7.8) (2.5,7.9)(2.6,7.7)(2.7,7.6)(2.7,7.7)(2.8,7.4) (2.9,7.4)(2.10,7.4)(2.11,7.3)(2.11,7.4)
SS3							(3.2,7.5)(3.3,7.5)(3.4,7.7)(3.5,7.6)(3.5,7.7)
SS4							(4.2,7.9)(4.3,7.9)(4.4,7.8)(4.5,7.8)
SS5							(5.3,7.3)(5.3,7.4)
SS6							(6.1,7.11)(6.2,7.10)(6.2,7.11)
SS7							

(b) Environment Interactions

	SS1	SS2	SS3	SS4	SS5	SS6	SS7
SS1					(1.6,5.1)(1.6,5.2)		
SS2			(2.5,3.3)(2.5,3.5) (2.7,3.3)	(2.5,4.3)(2.5,4.5)	(2.11,5.1)(2.11,5.2)	(2.7,6.2)	(2.5,7.6)(2.7,7.10)
SS3						(3.3,6.2)(3.5,6.2)	(3.3,7.6)(3.3,7.10) (3.5,7.10)
SS4						(4.3,6.2)(4.5,6.2)	(4.3,7.10)(4.5,7.10)
SS5							(5.1,7.3)(5.2,7.3)
SS6							(6.2,7.6)
SS7							

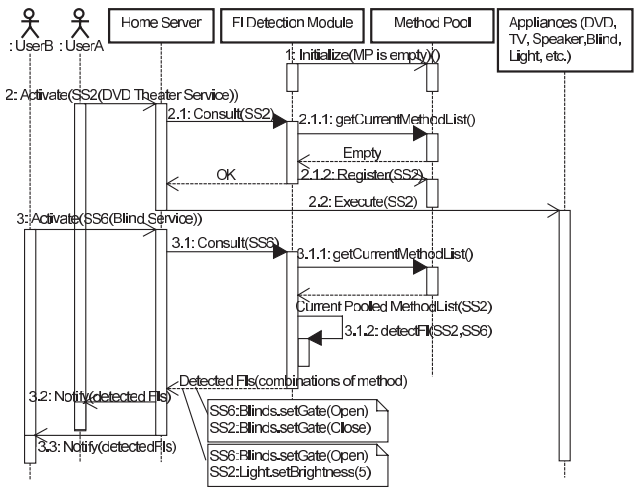

Fig. 4 Sequence diagram of online detection example.

illustrated in Sect. 4.2 are covered as (2.5, 3.3) and (4.5, 6.2) in Table 3 (b). Through a careful investigation, we confirm that all the detected FIs are reasonable and consistent against our formalization. The time taken for the tool to detect all the FIs among the seven services was 4.03 seconds, using a PC with Pentium III-M 933 MHz, 512 MB. This is efficient enough to conduct the offline detection.

5.3.2 Online Detection

Figure 4 shows an example of the online FI detection, where SS_2 and SS_6 are executed in this order by User A and User B. In the example, the home server communicates with the FI Detection Module which evaluates the conditions for FIs for given pairs of methods. The Fig. 4 shows a detailed sequence diagram from Method Pool Initialization until notification of detected FIs between SS2 and SS6.

First, SS_2 is activated by User A, and all the methods are registered to the empty method pool. Next, before the User A terminates SS_2, User B activates SS_6. Now, each method of SS_6 is examined against the ones in the pool by the detection module. The detection module evaluates the conditions, and successfully detects an appliance interaction (SS_2 : $Blinds.setGate(Close)$, SS_6 : $Blinds.setGate(Open)$), and an environment interaction (SS_2 : $Light.setBrightness(5)$, SS_6 : $Blinds.setGate(Open)$) during runtime. After these results of FI detection, the Home Server notifies them to users. Actually, the FIs detected by online detection vary depending on timing of the service activation and usage time of services. In this example, we imply a life-cycle of the services is expressed by the user.

For the performance evaluation, we have implemented the FIDM based on the proposed method, using the Java Web service (Apache AXIS2 + Tomcat 5.5). Experimental evaluation was done on a mid-range server (Pentium 4, 2.6 GHz). It was shown that the time taken for the FIDM to perform the detection algorithm (from Step 1 to Step 4) was around 80 milliseconds per a single execution, on average.

6. Evaluation

6.1 Performance

As described in the end of Sect. 5.3.1, the time needed for the off-line FI detection among seven practical services was just 4.03 second, by using the mid-class PC. We believe that the time is sufficiently small and feasible for the practical settings, since the off-line detection detects all potential FIs at once. In the online detection, FIDM poses a slight overhead ($\approx$ 80 msec.) to the execution of HNS integrated services as shown in Sect. 5.3.2. However, we consider it to be quite small compared to the total execution time of the integrated service.

6.2 Advantage and Limitation

We have presented FI detection methods on top of the solid HNS model. Since the proposed model does not depend on any specific platform or HNS protocols, one can conduct FI detection in early stages of service development efficiently.

Also, we proposed both offline and online detection methods. As seen in Sect. 5.3, the offline detection identifies all the potential FIs. Therefore, we consider that the offline method is quite useful to prepare in advance an appropriate *FI resolution scheme* (See Sect. 6.3) for every potential FI. On the other hand, the online detection should be used in actual service operations, since the system should not spare efforts FIs that do not actually occur. If the online method detects FIs, then the system dynamically dispatches an FI resolution scheme that has been prepared beforehand.

A limitation is that our model currently supports the sequential operator only for describing integrated services. This fact may limit the variety of services. However, as we allow more complex control flows, we need much more effort to assure the reliability of even a single integrated service. Thus, the ease of analysis and the expressive power of the model are in a tradeoff relation. We want to investigate this issue for the future research.

In this paper, our HNS model to detect feature interactions is built on a premise that the HNS is embedded in only one room. Actually, a HNS has multiple rooms. If the rooms are independent completely, that is, no environment property is affected by appliances allocated to different rooms, our model can correspond to the HNS. However, if any environment property is *written* by the method of appliances in the different rooms, our model should be extended to make the connection between the appliance (or environment property) and the location information.

6.3 Resolution of Feature Interaction

The proposed formalization of the HNS and integrated services allows us to take various approaches for *resolving* FIs. Here we present a brief sketch of resolution schemes. Further discussion on the interaction resolution schemes is left to our future research.

(a) Prioritize Services: Assign static priorities to services [17]. If a pair of service scenarios causes an interaction, then all conflicting methods in the service with lower priority are aborted.

(b) Prioritize Methods: Assign static priorities to methods. When a pair of methods conflict with each other, methods with a lower priority are aborted.

(c) Prioritize Users: Assign static priorities to users. A user with a higher priority can take precedence in executing services over the one with a lower priority.

(d) Compromise Services: Find a compromise between the conflicting services during runtime. In the service scenario, set a weight of *importance* to each method. For example, methods related to the DVD player, the TV, and the speaker are important for DVD Theater service, but the ones for the light and the blinds may be optional. When an interaction occurs, the service is compromised so that at least important methods are executed while optional methods are aborted.

(e) Compromise Methods: Find a compromise between the conflicting methods during runtime. For example, a conflict between `Speaker.setVolume(50)` and `Speaker.setVolume(10)` would be compromised to `Speaker.setVolume(30)`.

(f) Negotiate Among Users: Find a solution acceptable for users by conducting a negotiation. This approach is quite realistic as it is usually done manually in our daily life. A smarter approach will involve user agents which perform an automatic negotiation and resolution based on the user's policy and/or preference.

(g) Ask User: If the above resolution methods cannot derive any reasonable solution, this activity is chosen. When an FI is detected, ask the user(s) to determine manually how the interaction should be dealt with. This approach is not very elaborated but is some kinds of reasonable. Though it was taken in Step 3 of the proposed online detection, several options should be proposed to the user to support making decision.

6.4 Related Work

Kolberg et al. firstly addressed the feature interaction problem in the HNS [17], [20]. These authors regard each HNS component as a *resource*, and abstract the appliance operations as *resource locking*. Then, FIs are characterized as a resource competition, where different services try to lock a common resource in incompatible locking mode. Since all appliance operations are abstracted by nameless locking, the method achieves a light-weight runtime FI detection. However, the method cannot identify the concrete pair of appliance operations that actually cause FIs. The proposed method achieves finer-grained FI analysis in the sense that it can derive concrete appliance operations in conflict. Also, the proposed method has higher modeling fidelity in the sense that it can describe Kolberg's definition of FIs. Specifically, Condition D1 and E1 can capture Kolberg's MAI and STI interactions respectively, in more detailed level of abstraction. Also, Conditions D2 and E2 cover SAI and MTI. Thus, the proposed method can be used to implement their runtime interaction avoidance.

Matsuo and Pattara [21], [22] proposed an FI detection method for HNS services by means of the SPIN model checker. Taking our original definitions [9] (i.e., the appliance and environment interactions) into account, this method represents the conditions of FIs in LTL formulas, and verifies the formulas against HNS services described in PROMELA. Unlike ours, this method allows branches and loops in the service description, and can give complete proof within the model. However, this method is supposed to be used in only early stages of the development. Also it requires user's high expertise in LTL and PROMELA, which might be hard for practitioners. Our detection method is

much simpler, in the sense that it considers only APIs used in the services, but does not focus the control flow among the APIs. Because of the simplicity, it can be applied to a wide range of implementation languages, although the offline detection may yield *false negatives*. However, this problem can be complemented by the proposed online detection method.

Loke presented a modeling framework for HNS [23], where each appliance is a self-contained service component. The idea is to encapsulate the platform-specific issues in the service component, and then to construct integrated services as *workflows*. Although the concept for the modeling is similar to ours, the method does not address the feature interaction problem.

As far as is reported, explicit consideration of environmental factors in the control application was first introduced by Metzger [18]. Within the domain of embedded control systems, their approach captures the static structures of requirements and systems by dependency graphs, and conducts offline interaction detection for systems under development. Our method differs in targeting the HNS where the appliances and services can be dynamically added and modified. Hence, the proposed framework achieves *modularity* of every device (appliance) in an object-oriented fashion.

There have been methods that adopt some object-oriented approaches for the FI problem. Gibson et al. [24] presented a requirement model called *fair objects*. The model defines liveness properties that each object must satisfy eventually. When multiple objects are composed, the properties derive a specification to prevent some types of FIs (deadlock, disability of features). Prehofer et al. [25] exploited a concept of inheritance in object-oriented programming, and avoids FIs in incremental feature development.

These methods are useful for the purpose of *FI prevention*, in such a situation that developers can design, implement and modify all the objects in the system. Thus, the methods might be used for each vendor to circumvent FIs among features in individual appliance. However, every HNS appliance is a self-contained object (possibly under a multi-vendor environment). It is difficult to compromise requirements over multiple appliance objects, or to modify specification and implementation of the appliances to prevent FIs. Moreover, it is impossible to predict all possible combinations of appliances and scenarios within the integrated services. In such settings, we need a framework of *FI detection* for given integrated services, as we contributed in this work.

6.5 Applicability of Conventional Telephony-Based Methods

A number of methods for FIs in the *telephony domain* have been proposed so far [6]. However, we consider it not easy to apply these conventional methods directly to the HNS domain, due to the following two reasons.

Firstly, telephony services are provided on the *homogeneous* system, where all terminals (i.e., telephones) have the equivalent functionalities. Therefore, the conventional methods introduce state predicates and events commonly used for all terminals, and parameterize them to represent the actual state of every user. Thus, most conventional methods contain the *service specification* only, but not include any *terminal specification*. On the other hand, the HNS is a *heterogeneous* system, where different types and vendors of appliances are deployed. Hence, it is quite difficult to introduce the common predicates and events in the HNS. Therefore, in the proposed framework, we introduce the object model as *appliance specification*. Every actual appliance is specified by an appliance object model with properties, methods, and impacts to environment. This allows us to express heterogeneity of HNS quite naturally.

Secondly, in HNS we have to take account of not only individual terminals (i.e., appliances), but also their surrounding *environment*, as discussed in Sect. 3.3. Thus, the proposed environment interaction is the quite essential problem in HNS, but does not exist in the conventional telephony domain.

However, we found some techniques quite promising for *implementation* of the proposed method. For example, the approaches with logic programming (e.g., [7]) and/or structural analysis of rule-based methods would enable efficient pre/post-conditions checking of the appliance methods. A negotiating agent approach [8] would also help to implement an automatic interaction resolution for the scheme (h) in Sect. 6.3. Furthermore in [26], Kawaguchi et al. discussed the terminal assignment problem for the HNS, which addresses how to bind the appliance names and actual appliances. This method is quite promising to relax our Assumption A2 (see Sect. 2.3), which extends the proposed FI detection method for dynamic HNS.

7. Conclusion

In a smart home environment, various appliances are connected to a home network and provide users with value-added services such as appliance integration. In this paper, we have presented a framework for detecting feature interactions (FIs) in appliance integration services. We first formalized the HNS with object-oriented modeling. Then, the appliance interaction and the environment interaction were defined. Finally, we presented offline and online FI detection methods.

Several research directions present themselves. We are currently investigating efficient applications of the resolution schemes. Especially important is evaluating the feasibility of the suggested resolution schemes from several viewpoints: system-view, service-view, and user-view and so forth. Adaptation of the conventional techniques in telephony to the HNS integrated services is also an interesting topic for further study.

Acknowledgements

This research was partially supported by: the Japan Ministry

of Education, Science, Sports, and Culture, Grant-in-Aid for Young Scientists (B) (No.21700077, 20700027), and by JSPS and MAE under the Japan-France Integrated Action Program (SAKURA).

References

[1] Panasonic Electric Works Co., Ltd., "Lifinity," http://denko.panasonic.biz/Ebox/kahs/, 2008.
[2] T. Tamura, T. Togawa, M. Ogawa, and M. Yoda, "Fully automated health monitoring system in the home," Med. Eng. Phys., vol.20, no.8, pp.573–579, 1998.
[3] TOSHIBA Consumer Marketing Corp., "Toshiba home network – feminity," http://www3.toshiba.co.jp/feminity/feminity_eng/index.html, 2005.
[4] T. Yamazaki, "Beyond the smart home," Proc 2006 International Conference on Hybrid Information Technology (ICHIT '06), pp.350–355, Nov. 2006.
[5] Y. Tan and Digital Home Network Forum, Ubiquitous Technology: Home Network and Information Appliances, Ohmsha, 2004.
[6] M.D. Ryan, L.G. Bouma, E. Magill, et al., ed., Feature Interaction in Telecommunications, IOS Press, Amsterdam, 1992-2005.
[7] N. Gorse, L. Logrippo, and J. Sincennes, "Formal detection of feature interactions with logic programming and lotos," Journal of Software and System Modeling, vol.5, pp.121–134, April 2005.
[8] N.D. Griffeth and H. Velthuijsen, "The negotiating agents approach to runtime feature interaction resolution," Proc. Second Int'l Workshop Feature Interactions in Telecommunications Systems, Amsterdam, NL, pp.217–235, IOS Press, Amsterdam, May 1994.
[9] M. Nakamura, H. Igaki, and K. Matsumoto, "Feature interactions in integrated services of networked home appliances -an object-oriented approach-," Proc. Int'l. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05), Leicester, UK, pp.236–251, IOS Press, Amsterdam, June 2005.
[10] DLNA, "Digital living network alliance," http://www.dlna.org, 2007.
[11] ECHONET Consortium, http://www.echonet.gr.jp/english/index.htm, 1997-2004.
[12] OSGi Alliance, http://www.osgi.org/, 2006.
[13] X-10, http://www.x10pro.com/, 2006.
[14] HAVi, http://www.havi.org/, 2004.
[15] UPnP Forum, http://www.upnp.org/, 2007.
[16] J. Bourcier, A. Chazalet, M. Desertot, C. Escoffier, and C. Marin, "A dynamic-soa home control gateway," Proc. Int'l Conf. on Service Computing (SCC'06), pp.18–22, Sept. 2006.
[17] M. Kolberg, E.H. Magill, and M. Wilson, "Compatibility issues between services supporting networked appliances," IEEE Commun. Mag., vol.41, no.11, pp.136–147, Nov. 2003.
[18] A. Metzger, "Feature interactions in embedded control systems," Comput. Netw., vol.45, no.5, pp.625–644, 2004.
[19] K.K. Aggarwal, Y. Singh, and J.K. Chhabra, "A dynamic software metric and debugging tool," ACM SIGSOFT Software Engineering Notes, vol.28, no.2, pp.1–4, March 2003.
[20] M. Wilson, M. Kolberg, and E.H. Magill, "Considering side effects in service interactions in home automation - an online approach," in Feature Interactions in Software and Communication Systems IX, ed. L. du Bousquet and J.L. Richier, pp.172–187, IOS Press, Amsterdam, 2007.
[21] P. Leelaprute, T. Matsuo, T. Tsuchiya, and T. Kikuno, "Detecting feature interactions in home appliance networks," Proc. 9th Int'l Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2008), pp.895–903, Aug. 2008.
[22] T. Matsuo, P. Leelaprute, T. Tsuchiya, and T. Kikuno, "Verifying feature interactions in home network systems," IPSJ Journal, vol.49, no.6, pp.2129–2143, June 2008.
[23] S.W. Loke, "Service-oriented device ecology workflows," Proc. 1st Int'l Conf. on Service-Oriented Computing (ICSOC2003), Trento, Italy, pp.559–574, Springer-Verlag, Berlin, Dec. 2003.
[24] P. Gibson and D. Méry, "Fair objects," Proc. OT98 (COTSR), pp.1–16, Oxford, UK, April 1998.
[25] C. Prehofer, "An object-oriented approach to feature interaction," Feature Interactions in Telecommunications Networks IV, ed. P. Dini, R. Boutaba, and L. Logrippo, pp.313–325, IOS-Press, Amsterdam, June 1997.
[26] K. Kawaguchi and T. Ohta, "A study on a method for detecting feature interactions in home network," IEICE Technical Report, vol.107, no.88, pp.39–42, June 2007.

Hiroshi Igaki received the B.E. degree (2000) in Department of Electrical and Electronics Engineering from Kobe University, Japan, and the M.E. degree (2002) and D.E. degree (2005) in Information Science from Nara Institute of Science and Technology, Japan. In 2005, he worked for the Graduate School of Information Science at Nara Institute of Science and Technology, Japan. He joined Faculty of Mathematical Sciences and Information Engineering, Nanzan University, Japan. in 2006. He is currently an Assistant Professor of Graduate School of Engineering at Kobe University from 2007. His research interests include communication support in software development, web services and service-oriented architecture. He is a member of the IEEE, ACM, and IPSJ.

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