This paper was accepted for publication in the 10th International Conference on Feature Interactions (ICFI2009) in June 2009. The page number is not yet assigned for this version.

Considering Online Feature Interaction Detection and Resolution for Integrated Services in Home Network System

Masahide NAKAMURA [a,1], Hiroshi IGAKI [a], Yuhei YOSHIMURA [a] and Kousuke IKEGAMI [a]

[a] *Graduate School of Engineering, Kobe University, Japan*

Abstract. This paper presents an online detection and resolution method for feature interactions among integrated services in home network systems. To achieve reasonable online detection and resolution, we introduce three new concepts in this paper. Specifically, (a) *activation* which explicitly defines the execution lifetime of services, (b) *mandatory methods* which guarantees essential and optional operations in services, and (c) *suspend/resume mechanism* which allows lower-priority services to sleep temporarily and to wake up later when all conflicting services are terminated. A case study demonstrates the effectiveness of the proposed method.

Keywords. home network system, online resolution, activation, mandatory methods, suspend/resume mechanism

1. Introduction

The *home network system* (shortly, HNS) is a system comprised of networked home appliances and sensors, which realizes the next-generation *ubiquitous smart home* [10][11]. In the HNS, multiple appliances are orchestrated together to achieve value-added *integrated services* (or just simply called *services*). We here introduce some examples:

DVD Theater Service(DVD-T): This service allows a user to watch movies in a theater-like atmosphere. When activated, a DVD/HDD recorder is switched on, a TV is turned on in DVD mode, a curtain is closed, lights are darken, 5.1ch speakers are selected with appropriate sound volume, and the recorder plays back the movie.

Coming Home Service(CH): Integrating a door sensor and lights, the service supports a user to get into the house. When a door sensor notices that the user comes home, lights are automatically turned on while the user is entering the room.

Leaving Home Service(LH): Integrating the existing appliances, the service prevents a user from forgetting turning off appliances when leaving home. When the user presses a button at the entrance, all appliances are turned off to save energy.

[1]Corresponding Author: 1-1 Rokkodai-cho, Nada-ku, Kobe, Hyogo 657-8501, Japan; E-mail: masa-n@cs.kobe-u.ac.jp .

As is easily imagined, *feature interactions* (FIs, for short) occur among these services if they are used simultaneously. We describe some example scenarios.

FI Scenario 1 (DVD-T & CH) While user A is watching a movie with DVD-T, if user B comes home, CH turns on the lights. This disturbs the theater atmosphere.

FI Scenario 2 (CH & LH) User C comes home and presses the button of LH at the same time. It is non-deterministic whether or not the lights should be turned on.

FI Scenario 3 (LH & DVD-T) User D leaves home and presses the button of LH although user E is watching a movie. The TV and the DVD/HDD recorder are shutdown, which makes E angry.

In our previous work [7], we have proposed an object-oriented framework that formalizes FIs within the HNS integrated services. In the framework, we have defined two kinds of FIs, *appliance interaction* and *environment interaction*. Intuitively, the appliance interaction refers to a situation where two services share the same appliance in a conflicting way. For instance, in Scenario 3, DVD-T wants the TV and DVD/HDD to be active, whereas LH wants to shutdown them. We can also see the appliance interactions on some lights in Scenarios 1 and 2. On the other hand, the environment interaction occurs when two services, which do not necessarily share common appliances, conflict via environment properties. For instance, in Scenario 1, even if DVD-T and CH are using different lights, a conflict occurs on *brightness* in the room. Using the framework, we have then proposed an *offline* FI detection method, and some naive resolution schemes based on priorities over services.

However, on implementing the *online* FI detection and resolution mechanism in a real HNS [8][9], we have found that the previous framework could not handle some FI scenarios reasonably, since the following issues were not sufficiently considered.

P1: How do we detect FIs *during runtime*, considering timing of service execution?
P2: How do we define *essential* or *compromised* service operations in FI resolution?
P3: How do we deal with lower-priority services in FI resolution?

In this paper, we aim to propose a new online FI detection and resolution scheme for the HNS integrated services, taking the above P1-P3 into account. To achieve this, we extensively introduce the following three new notions: (a) *activation*, (b) *mandatory methods*, and (c) *suspend/resume mechanism*.

First, the activation defines the *execution lifetime* of services to cope with P1. Managing the activation for each service enables more precise FI detection during runtime. Second, the mandatory methods identify essential or optional operations in each service, which addresses P2. A service can run only when all mandatory methods are executable. In other word, even if some optional methods are suppressed by another higher-priority service, the service can continue to operate in a compromised mode. Finally concerning P3, the suspend/resume mechanism provides a fair and gentle treatment for lower-priority services. When an FI occurs, the mechanism allows lower-priority services to sleep temporarily, and to wake up after all conflicting services are terminated.

Integrating these new notions with the previous framework, we propose more efficient and reasonable online FI detection and resolution method for the HNS integrated services. To show the effectiveness we conduct a case study through practical services. It is shown that the proposed method achieves more reasonable FI detection and resolution, compared with the previous framework.

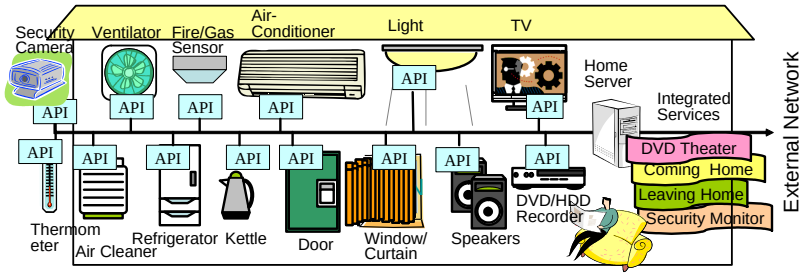

Figure 1. Home network system

2. Preliminaries

This section outlines the context and scope by reviewing our previous work briefly. More detailed description can be found in [7].

2.1. Home Network System (HNS)

As illustrated in Figure 1, a HNS consists of *networked home appliances* and a *home server*, connected to LAN at home. Each appliance typically has a set of *control APIs*, with which the user or external software agents can control the appliance. The home server works as a centralized application server managing appliances and services. It also plays a role of the gateway to the external network. The *integrated service* is an application that invokes the APIs of multiple appliances, according to a certain service logic. The services are installed and executed on the home server.

2.2. Object-Oriented Framework for FIs in HNS

In our previous work [7], we proposed an object-oriented framework to formalize the HNS. In the framework, the HNS is defined as a set of appliances and an environment. Each *appliance* is modeled as an *object* consisting of *attributes* (also called *properties*) and *methods*. The attributes are primary variables characterizing the state of the appliance, which are referred or updated by the methods. The methods correspond to the appliance APIs, each of which is abstracted by a pair of a *pre-condition* and a *post-condition*. The pre-condition $Pre(m)$ of a method m represents a guard condition over the attributes that must be satisfied *before* m is executed. The post-condition $Post(m)$ of a method m is a resultant condition over the attributes guaranteed *after* m is executed.

The *environment* is modeled as a global object that has a set of *environment attributes*, such as temperature, brightness, sound volume. The environment attributes can be read or written indirectly by the appliance methods. For simplicity, we assume in this paper that all the appliances in the HNS share a *single room* (thus, the environment is a singleton object). We also assume that an one-to-one mapping from every appliance object to a real appliance is already given.

Each integrated service is modeled by an object that uses multiple appliances. Every service is supposed to have `begin()` method, which invokes a sequence $\{m_1; m_2; ...; m_n;\}$ of appliance methods m_i's when activating the service. Optionally, a service can have `end()` method with $\{m_{e1}; m_{e2}; ...; m_{ek}\}$, which declares the post-processing for terminating the service.

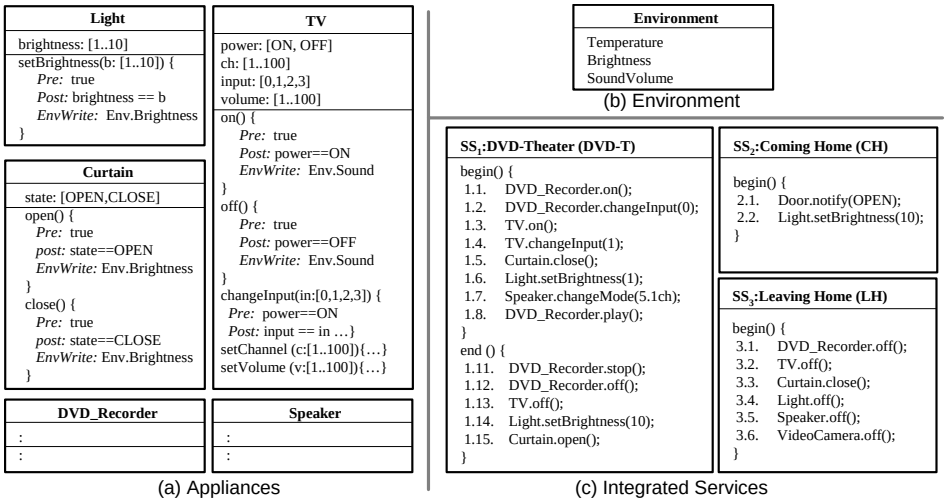

Figure 2. Object Oriented Model for the Example HNS (partly shown)

Figure 2 shows the models of (a) the appliances, (b) the environment, and (c) the three integrated services introduced in Section 1. The appliances are drawn in the similar description to the UML class diagram. $[A, B, C, ...]$ represents the enumerated type, and $[l..u]$ represents the integer type with the range from l to u. *Pre* and *Post* in each appliance method are respectively the pre- and post-conditions. *EnvWrite* means that the method may overwrite the value of the specified environment attribute. As shown in Figure 2(b), the environment has three attributes in this example. In Figure 2(c), a method `m()` of an appliance `A` is denoted by `A.m()`. For convenience, we assign an index number for each method. Every service has `begin()` method that achieves the scenario described in Section 1. Only DVD-T has `end()` method, which is supposed to be executed by the user when he/she wants to terminates the service.

2.3. Formalization of FIs and Offline Definition

Using the HNS model, we have defined two types of FIs in [7]. In the following, let s_1 and s_2 be given integrated services.

Definition 1 (Appliance Interaction) Let m_1 and m_2 be methods within the same appliance d. We say that m_1 *and* m_2 *are in conflict on the appliance*, denoted by $m_1 \asymp_{app} m_2$, iff m_1 and m_2 satisfy at least one of the following conditions:

Condition D1: $Post(m_1) \wedge Post(m_2) = \Phi$ (unsatisfiable)
Condition D2: $Post(m_1) \wedge Pre(m_2) = \Phi$ (unsatisfiable)

We say that s_1 and s_2 cause an *appliance interaction*, denoted by $s_1 \asymp_{app} s_2$, iff there exists a pair of methods (m_1, m_2) such that [$m_1 \asymp_{app} m_2$], [m_1 is involved in s_1] and [m_2 is involved in s_2].

Intuitively, Condition D1 means that two methods has *conflicting goals* (post-conditions) that cannot be satisfied simultaneously on the common appliance. A typical

example is `TV.on()`$\asymp_{app}$`TV.off()`. Condition D2 is the case where the execution of one method may disable another. The appliance interaction can occur when such conflicting methods are invoked by different services.

Definition 2 (Environment Interaction) Let m_1 and m_2 be appliance methods. We say that m_1 *and* m_2 *are in conflict on the environment*, denoted by $m_1 \asymp_{env} m_2$, iff m_1 and m_2 satisfy at least one of the following conditions:

Condition E1: $EnvWrite(m_1) \cap EnvWrite(m_2) \neq \phi$
Condition E2: $EnvWrite(m_1) \cap EnvRead(m_2) \neq \phi$

We say that s_1 and s_2 cause an *environment interaction*, denoted by $s_1 \asymp_{env} s_2$, iff there exists a pair of methods (m_1, m_2) such that [$m_1 \asymp_{env} m_2$], [m_1 is involved in s_1] and [m_2 is involved in s_2].

Condition E1 says that two methods try to write some common environment attributes simultaneously. A typical example is `AirConitioner.cooling()`$\asymp_{env}$`Heater.on()`. Condition E2 is the case where reading from and writing to a common environment attribute may cause a race condition.

In the following, we may simply use $\asymp$ to represent either $\asymp_{app}$ or $\asymp_{env}$ when no clear distinction is needed. Using the formalization, we can now conduct the offline FI detection as follows.

Offline FI detection:
Input: A pair of integrated services s_1 and s_2, a model of HNS consisting of appliances and an environment.
Output: Two sets $FI_a(s_1, s_2)$ and $FI_e(s_1, s_2)$ of pairs of conflicting methods s.t.

$$FI_a(s_1, s_2) = \{(m_1, m_2) | [m_1 \asymp_{app} m_2] \wedge [m_1 \text{ is in } s_1] \wedge [m_2 \text{ is in } s_2]\}$$
$$FI_e(s_1, s_2) = \{(m_1, m_2) | [m_1 \asymp_{env} m_2] \wedge [m_1 \text{ is in } s_1] \wedge [m_2 \text{ is in } s_2]\}$$

Procedure: Initialize $FI_a(s_1, s_2)$ and $FI_e(s_1, s_2)$ be empty. Let $\{m_{11}; m_{12}; ...; m_{1n};\}$ be a sequence of methods specified in s_1. Also, let $\{m_{21}; m_{22}; ...; m_{2k};\}$ be a sequence of methods specified in s_2. For every pair (m_{1i}, m_{2j}) $(1 \leq i \leq n, 1 \leq j \leq k)$, if $m_{1i} \asymp_{app} m_{2j}$, then put it in $FI_a(s_1, s_2)$. If $m_{1i} \asymp_{env} m_{2j}$, then put it in $FI_e(s_1, s_2)$. If both $FI_a(s_1, s_2)$ are $FI_e(s_1, s_2)$ empty, conclude that no FI occurs.

All FI Scenarios 1, 2, 3 in Section 1 can be exaplained and detected by the offline detection. For example, FI Scenario 1 is explained by conflicting methods on the light, i.e., `DVD-T.Light.setBrightness(1)`$\asymp_{app}$`CH.Light.setBrightness(10)`.

2.4. FI Resolution with Priority

A simple resolution scheme for FIs in HNS integrated services is to introduce (static) *priority* (denoted by $pri(s)$) for each service s. When $s_1 \asymp s_2$, if $pri(s_1) > pri(s_2)$, then the execution of s_1 takes precedence over that of s_2. For this, there are two naive approaches on how to deal with s_2.

FI Resolution 1 (AbortAll) Abort (or cancel) all the methods in s_2.

FI Resolution 2 (KeepAlive) Keep (or execute) non-conflicting methods in s_2, and abort (or cancel) only conflicting methods.

2.5. Scenarios in Questions

Based on the framework described above, we tried to conduct FI detection and resolution *during runtime*. However, we have faced with some scenarios that could not be explained reasonably by the previous framework only. Consider again the integrated services (DVD-T, CH and LH) and FI Scenarios 1, 2 and 3 introduced in Section 1.

Scenario Q1 (CH & LH) Let us consider FI Scenario 2 with slightly different setting. Suppose that user C executes LH to leave home. Immediately after that, C returns home to pick up something left behind, and then CH is activated. In this case, can we say that "CH and LH really cause the FI", as the offline detection says so? This question is related to the problem P1 in Section 1.

Scenario Q2 (DVD-T & CH) Assuming that $pri(\text{CH}) > pri(\text{DVD-T})$, we try to resolve the FI within FI Scenario 1. If we take the AbortAll resolution, all methods in DVD-T are aborted. This is quite relentless for A since only the light is disturbed. On the other hand, taking the KeepAlive resolution seems to be OK in this case, since A can keep watching the movie with compromising the light. However, if the DVD-recorder and the TV are occupied by another higher-priority service (say Security Monitor Service), then KeepAlive scheme w.r.t. DVD-T activates the curtain, the light and the speaker only without the TV and the DVD recorder. Is this meaningful for the DVD-T service? This question is related to the problem P2.

Scenario Q3 (LH & DVD-T) Consider FI Scenario 3 with $pri(\text{LH}) > pri(\text{DVD-T})$. According to the priority, all the appliances used in DVD-T are shutdown by LH. Then, questions may arise from E; "When do I start DVD-T again?", "Does nobody resume my interrupted DVD-T after D leaves home?". Both of the previous resolution schemes just abort low-priority methods in conflict, and forsake them even after the high-priority methods are terminated. More sophisticated schemes are needed to achieve *fairness* for lower-priority services. This question is related to the problem P3.

3. Proposed Method

3.1. Activation

The problem in Scenario Q1 arises because the previous framework lacked the notion of time. On receiving a request, a service executes the sequence of the appliance methods specified in `begin()` method. However, after all the methods are executed, it is quite ambiguous *until when the service is recognized to be active*. For this, we introduce the notion of *activation*, which explicitly defines the execution lifetime of a service.

Definition 3 (Activation) Let s be an integrated service. *Activation* of s is defined as an execution period (lifetime) from when s is activated with `begin()` method until s is terminated. Depending on when and how s is terminated, we define the following three types of activation.

(a) begin-end type: Activation continues until when the user explicitly terminates s with `end()` method.

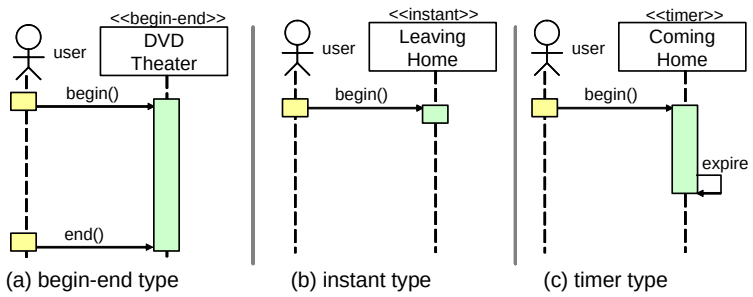

Figure 3. Three types of activation

(b) instant type: Activation ends immediately when all the appliance methods in s are executed.

(c) timer type: Activation continues for a pre-defined time. When the time expires, s autonomously terminates itself (with `end()` method if any).

We assume that exactly one of these types must be associated with every service. We say that s is *active* iff s is in activation. Also, we say s is *inactive* iff s is not active. An appliance method m in s is said to be active iff s is active.

Figure 3 shows UML sequence charts describing the three activation types associated with three services DVD-T, LH and CH. In the figure, a rectangle below each service represents the activation. Considering the nature of DVD-T, the service should be continued until when the user explicitly stops watching the movie. Thus, we suppose that DVD-T has the begin-end activation as shown in Figure 3(a). As for LH, the service can be recognized as "finished" immediately after all the appliances are shut down. So, we associate the instant type with LH, as shown in Figure 3(b). We assigned the timer-type activation for CH (Figure 3(c)). The light should be brightened for some short period while the user gets into the room.

Based on the activation, we extend the definition of FIs for online FI detection.

Definition 4 (Interaction During Runtime) We say that s_1 and s_2 cause an *interaction during runtime*, denoted by $s_1 \bowtie s_2$, iff [s_1 and s_2 are both active] and [$s_1 \asymp s_2$].

Using the new definition, we explain two scenarios with and without FIs for the same combination of services (CH and LH). Consider FI Scenario 2 in Section 1, where CH is first activated. Since the activation type of CH is the timer type, CH becomes active and turns on the light for a while. LH is then activated before CH is expired. This causes an FI during runtime since the activations of CH and LH overwrap. Let us consider Scenario Q1 in Section 2.5, where LH is first executed with the instant type. After all appliances are shutdown, LH is soon terminated and becomes inactive. Then CH is activated. In this case, no FI occurs during runtime since LH is no more active.

We now present an online FI detection method. In practice, the online detection should be performed *before* FIs actually occur. Hence, the detection is conducted between an inactive service going to be newly activated, and services already active within the HNS. For this, we deploy a server process, called *FI manager*, between the user and

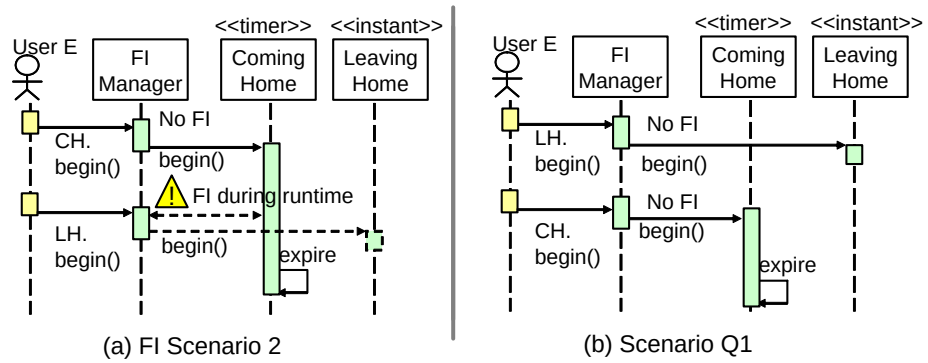

Figure 4. Online FI detection with FI manager

the services. The FI manager keeps monitoring the activation of every service, intercepts every service request from a user, and performs the online FI detection procedure.

Online FI detection (conducted by FI manager):
Input: An inactive service s_0 that is going to be activated, a set of already active services $AS = \{s_1, s_2, ..., s_n\}$, a model of HNS consisting of appliances and an environment.
Output: Two sets $FIR_a(s_0, AS)$ and $FIR_e(s_0, AS)$ of pairs of methods conflicting during runtime s.t.

$$FIR_a(s_0, AS) = \{(m_0, m) | [m_0 \asymp_{app} m] \wedge [m_0 \text{ is in } s_0] \wedge [m \text{ is in some } s_i \in AS]\}$$
$$FIR_e(s_0, AS) = \{(m_0, m) | [m_0 \asymp_{env} m] \wedge [m_0 \text{ is in } s_0] \wedge [m \text{ is in some } s_i \in AS]\}$$

Procedure: Initialize $FIR_a(s_0, AS)$ and $FIR_e(s_0, AS)$ to be empty. For every $s_i \in AS$ $(1 \leq i \leq n)$, calculate $FI_a(s_0, s_i)$ and put the result in $FIR_a(s_0, AS)$. Also, calculate $FI_e(s_0, s_i)$ and put the result in $FIR_e(s_0, AS)$. We just write $FIR(...)$ when no distinction between $FIR_a(...)$ and $FIR_e(...)$ is needed.

We assume that for every service request, the FI manager performs an *atomic transaction* of FI detection, FI resolution (presented later), and service execution. Even if multiple requests occur simultaneously, they are serialized and processed one-by-one.

Figure 4 shows how to detect FIs during runtime within the two example scenarios. In Figure 4(a), when E executes CH, the FI manager says that no FI occurs since $FIR(CH, \{\}) = \phi$. Next, when E executes LH, the FI manager detects an FI, since $FIR_a(LH, \{CH\}) = \{$ `(LH.Light.off(), CH.Light.setBrightness(10))` $\}$. If no resolution is applied, CH and LH actually cause an FI as shown in the dotted activation on LH. On the other hand, in Figure 4(b) when E executes CH following LH, no FI occurs since LH is no more active and $FIR(CH, \{\}) = \phi$.

3.2. Mandatory Methods

The problem in Scenario Q2 arises because all appliance methods within a service had the same weight of importance. For instance, for DVD-T, methods related to the HDD/DVD recorder and the TV are *mandatory*, because the user cannot watch movies without them. On the other hand, methods related to the curtain or the light are somehow *optional*. Even without them the user would enjoy the movie in a compromised setting.

Definition 5 (Mandatory Methods) Let s be an integrated service, and let m be an appliance method within s. We say that m is *mandatory* iff omission of m cannot achieve the essential requirement of s. We say that m is *optional* iff m is not mandatory.

For every service s, we assume that a set $Man(s)$ of mandatory methods within s must be *given* by the designer of s. Taking FIs during runtime into account, we then define the *executablity* of appliance methods.

Definition 6 (Enabling Condition of Method) Let $s.m$ be an appliance method in an inactive service s. Let AS be a set of services currently active, and let $s_a.m_a$ be any active method in some active service $s_a \in AS$. We say that $s.m$ is *enabled*, denoted by $enabled(s.m)$, iff one of the following conditions is satisfied.

Condition EC1: There exists no $s_a.m_a$ such that $(s.m, s_a.m_a) \in FIR(s, AS)$, or
Condition EC2: If there exists $s_a.m_a$ such that $(s.m, s_a.m_a) \in FIR(s, AS)$, then $pri(s) > pri(s_a)$.

Intuitively, $s.m$ can be executed only when [there is no conflicting method currently active], or [even if there is, then the priority of $s.m$ is higher than that of any conflicting method]. Then, we consider that a service s can be executed only when all of its mandatory methods are enabled.

Definition 7 (Enabling Condition of Service) Let s be an integrated service. We say that s is *enabled*, denoted by $enabled(s)$, iff $\forall m \in Man(s); enabled(m) = true$.

Note that evaluation of the enabling condition requires the online FI detection. So, we assume that the *FI manager evaluates* $enabled(s)$ *during runtime* on receiving a request of s. Execution of s may cause FIs with currently active services, and some lower-priority services are *suppressed* by s through the priority-based resolution.

Definition 8 (Suppress Relation) Let s_a be a service already active within the HNS, and let s be an inactive service going to be activated. We say that s_a *is suppressed* by s, denoted by $s_a \ll s$, iff [$pri(s_a) < pri(s)$] and [$s_a \bowtie s$ on executing s]. Similarly, we define $m_a \ll m$ for methods m_a in s_a and m in s.

Using the enabling condition and the suppress relation, we present a new scheme of online resolution.

FI Resolution 3 (KeepMandatory) Let s be an inactive service to be activated. If $enabled(s)$ is false, cancel s. If $enabled(s)$ is true, perform the following for every method m in s. If there exists s_a such that $s_a \ll s$, abort all conflicting methods m_a's in s_a such that $m_a \ll m$. For this, if $m_a \in Man(s_a)$, then abort the whole service s_a (with `end()` method if any). If m is mandatory, execute m. If m is optional, execute m as long as $enabled(m)$ is true.

Figure 5 shows the application of the KeepMandatory scheme to Scenario Q2 in Section 2.5. Within DVD-T, we assume that `Curtain.close()` and `Light.setBrightness(1)` are only optional, and others are all mandatory. As for CH, all methods are assumed to be mandatory. In Figure 5(a), suppose now that pri(DVD-T) $<$ pri(CH). User A

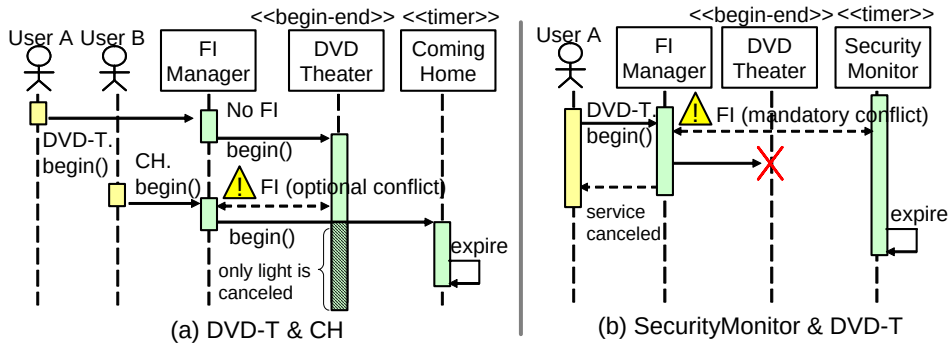

Figure 5. Online FI resolution with KeepMandatory scheme

first activates DVD-T, and then B comes home activating CH. The FI manager detects an FI between CH and DVD-T since `CH.Light.setBrightness(10)` $\asymp_{app}$ `DVD-T.Light.setBrightness(1)`. Because CH has a higher priority, the former method is enabled while the latter method is suppressed and aborted. However, as `DVD-T.Light.setBrightness(1)` is optional, DVD-T can continue the service with compromising the light setting. Note that the same resolution is performed for DVD-T even if CH is executed first. Next, consider a situation where the TV and the DVD recorder are occupied by SecurityMonitor Service, as shown in Figure 5(b). In this case, since mandatory methods of DVD-T are not enabled, the whole service is canceled.

3.3. Suspend/Resume Mechanism

The problem in Scenario Q3 (see Section 2.5) was that when $s \ll s'$ held, there was nothing for it but to abort s. To cope with this, we devise a new resolution scheme, called *SuspendResume*, by slightly extending the KeepMandatory scheme. Intuitively, even if $s \ll s'$ holds, we make s sleep temporally, and wake up later when s' is terminated.

FI Resolution 4 (SuspendResume) Let s be an inactive service to be activated. If $enabled(s)$ is false, cancel s. If $enabled(s)$ is true, perform the following for every method m in s. If there exists s_a such that $s_a \ll s$, suspend all conflicting methods m_a's in s_a such that $m_a \ll m$. For this, if $m_a \in Man(s_a)$, then suspend the whole service s_a (with `end()` method if any). If m is mandatory, execute m. If m is optional, execute m as long as $enabled(m)$ is true. When s is still terminated, if s_a is suspended and $enabled(s_a)$ is true, resume s_a by executing `begin()` method. If s_a is still running, resume suspended optional methods m_a's by executing them again, as long as $enabled(m_a)$ is true.

Depending on the nature of s, it would be good if we can choose either s should be suspended or aborted, when suppressed. For this, we assume that every service s has a boolean flag `resumable(s)`. If this flag is true, s can use the suspend/resume mechanism when suppressed. Otherwise, s is aborted.

To achieve the scheme, we define a *life cycle* of a service, which is described in a UML state chart in Figure 6. In the figure, a rectangle represents a state, a labeled arrow

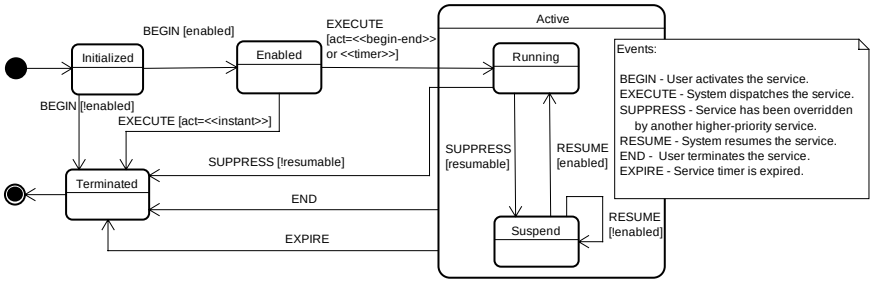

Figure 6. Service (Method) life cycle with suspend/resume

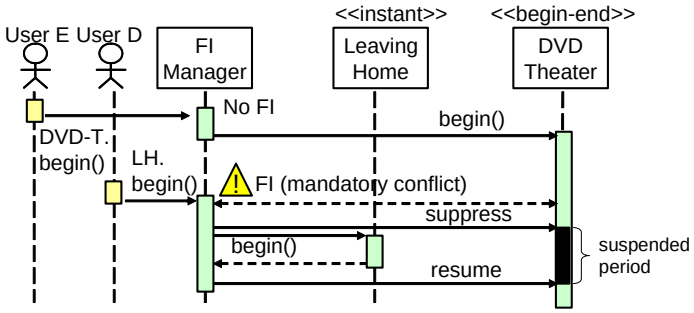

Figure 7. Online FI resolution with SuspendResume scheme

represents a transition in a form of `EVENT [condition]`. We assume that the *life cycle is managed and monitored by the FI manager*.

Every service s is instantiated at `Initialized` state. If a user activates s with `begin()` method, $enabled(s)$ is evaluated by the FI manager through online FI detection among mandatory methods. If $enabled(s)$ is false, the execution is canceled and s moves to `Terminated` state. If true, s moves to `Enabled` state and waits for dispatch. Then, the FI manager dispatches s to the home server. For this, if the activation type of s is instant (`[act=«instant»]`), s moves to `Terminated` state due to its definition. Otherwise, s moves to `Running` state. Now, if another higher-priority service s' is activated and any FI is detected, then the FI manager issues `SUPPRESS` event to s. If $resumable(s)$ is true, s is suspended at `Suspend` state. After all services suppressing s are terminated and $enabled(s)$ holds, s is resumed to `Running` state. Both `Running` and `Suspend` states belong to `Active` super-state. Thus, Hence, s can be terminated from either `Running` or `Suspend` state, when the user terminates s (`END`) or s expires (`EXPIRE`).

Figure 7 shows an application of the SuspendResume scheme to resolve the FI in Scenario Q3 (see Section 2.5). In this scenario, E first activates DVD-T to watch the movie. At this stage, no FI occurs. Then, D tries to activate LH. Since DVD-T is still active and $pri(\text{LH}) > pri(\text{DVD-T})$, DVD-T is suppressed by LH, i.e., DVD-T $\ll$ LH. The FI manager issues an event `SUPPRESS` to DVD-T, and then DVD-T is suspended. Next, LH is executed and all the appliances are shutdown. Since the activation type of LH is instant, LH is soon terminated. On confirming the termination of LH, the FI manager automatically resumes DVD-T by turning on related appliances again. Thus, E is happy to continue watching movies.

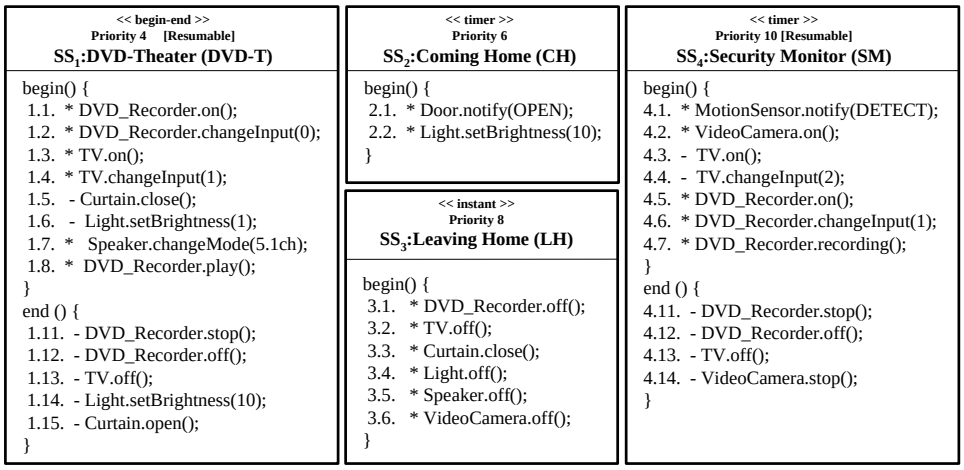

Figure 8. Integrated services used in case study

4. Case Study

To demonstrate the effectiveness of the proposed method, we conduct a case study in this section. In the case study, we use four integrated services in Figure 8: **DVD Theater (DVD-T)**, **Coming Home (CH)**, **Leaving Home (LH)**, and **Security Monitor (SM)**. The first three services are same as those introduced in Section 1. The Security Monitor service is defined as follows.

Security Monitor Service(SM): Integrating a motion sensor, a video camera, a TV and a DVD/HDD recorder, this service provides security monitoring. When the sensor notices that somebody breaks into the house garden at night, the camera shoots the view, the TV is turned on to display the camera view, and the DVD/HDD recorder is turned on to record the view. The recording is automatically terminated after pre-defined time.

As shown in Figure 8, each service has an activation type, a priority, and a resumable flag. Each method is either mandatory (labeled as $*$) or optional (labeled as $-$).

We apply the proposed online FI detection and resolution method to all *pairs* among the four services. Due to limited space, we examine only appliance interactions among them. Note that, in the online detection and resolution, we have to consider *execution order* of the services. That is, for every pair of services s_1 and s_2, we have to investigate FIs within two different execution order: (s_1, s_2) and (s_2, s_1). As for the resolution scheme, we adopt the SuspendResume scheme.

Table 1 shows FIs detected by the proposed online detection method. Each row represents a service firstly executed, whereas each column represents a service secondly activated. Each entry represents a set of conflicting methods causing FIs within the combination of service. It can be seen that some FIs depend on the execution order. For instance, when LH is the first service, no FI occurs. This is because LH has an instant activation. On the other hand, when LH is the second, it interacts with every service. When the first service has the timer-type activation, the occurrence of FIs becomes *conditional*

Table 1. FIs detected by proposed online detection

First Service \ Second Service	DVD-Theater (DVD-T)	Coming Home (CH)	Leaving Home (LH)	Security Monitor (SM)
DVD-T		(1.6,2.2)	(1.1,3.1)(1.2,3.1)(1.3,3.2)(1.4,3.2) (1.6,3.4)(1.7,3.5)(1.8,3.1)	(1.2,4.6)(1.4,4.4)(1.8,4.7)
CH	(2.2,1.6) *		(2.2,3.4) *	
LH				
SM	(4.6,1.2) (4.4,1.4) (4.7,1.8) *		(4.5,3.1) (4.6,3.1) (4.7,3.1) (4.3,3.2) (4.4,3.2) (4.2,3.6) *	

(entries with *). These FIs are detected only when the second service is activated before the first is expired.

In the following, we describe the interpretation of the FI scenarios, and how those FIs are resolved by the proposed Suspend/Resume resolution.

FI-1 (DVD-T, CH): While DVD-T is active, CH is executed. An FI occurs on the light. Since CH has a higher priority, the light setting of DVD-T is suppressed, and the light is brighten by CH. However, this does not suspend DVD-T, because the operation on the light is optional. The light automatically becomes dark again after CH expires, as the suspend/resume mechanism works for DVD-T.

FI-2 (DVD-T, LH): While DVD-T is active, LH is executed. FIs occur on all appliances that DVD-T uses. Since LH has a higher priority, DVD-T is suppressed by LH. As the mandatory methods are in conflict, DVD-T is suspended by executing `end()` method. Then LH shutdowns all appliances. As the activation of LH ends instantly, DVD-T is soon resumed, and the appliances are turned on again by `begin()` method.

FI-3 (DVD-T, SM): While DVD-T is active, SM is executed. FIs occur on the TV and the DVD recorder. As the mandatory methods are in conflict, DVD-T is suspended for a while. Then SM uses the TV and the recorder. When SM expires, DVD-T is soon resumed and all the appliances are turned on again.

FI-4 (CH, DVD-T): While CH is active, DVD-T is executed. An FI occurs on the light. Although DVD-T has a lower priority than CH, DVD-T is enabled since the light setting is not mandatory for DVD-T. When executed, the appliances except the light is turned on for watching movie.

FI-5 (CH, LH): While CH is active, LH is executed. An FI occurs on the light. Since LH has a higher priority, CH is suppressed by LH. For this, CH is not suspended but is immediately terminated, as the resumable flag is not assigned to CH.

FI-6 (SM, DVD-T): While SM is active, DVD-T is executed. FIs occur on the TV and the DVD recorder. As the mandatory resources are already in use by the higher-priority service SM, DVD-T is not enabled and thus canceled.

FI-7 (SM, LH): While SM is active, LH is executed. FIs occur on the TV and the DVD recorder. As the mandatory operations cannot be executed due to conflicts with the higher-priority service SM, LH is not enabled and thus canceled.

As we review the above results, all the FIs are detected and resolved in a quite natural and reasonable way, which does not contradict to our intuition. In our future work, we plan to increase the credibility by applying it to more services.

Table 2. FIs detected by previous offline detection

	Second Service			
	DVD-Theater (DVD-T)	**Coming Home (CH)**	**Leaving Home (LH)**	**Security Monitor (SM)**
DVD-T		(1.6,2.2)	(1.1,3.1)(1.2,3.1)(1.3,3.2)(1.4,3.2) (1.6,3.4)(1.7,3.5)(1.8,3.1)	(1.2,4.6)(1.4,4.4)(1.8,4.7)
CH	(2.2,1.6)		(2.2,3.4)	
LH	(3.1,1.1)(3.1,1.2)(3.2,1.3)(3.2,1.4) (3.4,1.6)(3.5,1.7)(3.1,1.8)	(3.4,2.2)		(3.1,4.5)(3.1,4.6)(3.1,4.7) (3.2,4.3)(3.2,4.4)(3.6,4.2)
SM	(4.6,1.2) (4.4,1.4) (4.7,1.8)		(4.5,3.1) (4.6,3.1) (4.7,3.1) (4.3,3.2) (4.4,3.2) (4.2,3.6)	

FIs that do not actually occur.
FIs that occur only in a special timing.

Table 3. Evaluation of resolution schemes w.r.t. treatment for suppressed services

Resolution Scheme	**Survivability**	**Functionality**	**Overhead**
AbortAll	Low	Aborted	Low
KeepAlive	Medium	Not Guaranteed	Low
KeepMandatory	Medium	Compromised or Aborted	Medium
SuspendResume	High	Guaranteed	High

5. Discussion

5.1. Advantage of Proposed Method

To evaluate the advantage of the proposed method, we first examine the FI detection results between the previous offline and the proposed online detection methods. Table 2 shows the pair of conflicting methods detected by the previous offline FI detection. Since the offline detection does not consider the activation, it always assumes that two services s_1 and s_2 are executed *simultaneously*. Thus, it can be seen, in Table 2, that for both combinations of (s_1, s_2) and (s_2, s_1) the same pairs of conflicting methods are detected. Compared to the result with the proposed online method (see Table 1), we can see that the previous offline method *over-detects* FIs which may not actually occur, as shown in the shaded entries in Table 2.

We then evaluate the four FI resolution schemes. Table 3 compares the four schemes with respect to treatment for lower-priority services (i.e., suppressed services). The metrics are (a) *survivability*, whether or not a service can continue to function when suppressed, (b) *functionality*, whether or not a service can achieve a functional requirement as a result of resolution, (c) *overhead*, how much effort needed for the FI manager for the resolution. The AbortAll scheme gives a quite simple solution, but gives no room for suppressed services to survive. The KeepAlive scheme is a best-effort approach in which non-conflicting methods can live. However, requirements of services may not be guaranteed as a result of resolution. The KeepMandatory scheme circumvents this problem by introducing the notion of mandatory methods. However, if any mandatory method is suppressed, the whole service has to be aborted. Finally, the SuspendResume scheme provides a way to recover for suppressed services using the suspend/resume mechanism. Thus, we consider that the SuspenedResume scheme is most effective and reasonable among the four, although it requires most expensive overhead in FI resolution.

5.2. Limitations

In prioritizing services, we have assigned a static priority for every service, which assigns the same priority value to all the appliance methods within the service. This is simple but

works well with the proposed method. However, we cannot say currently that the service-wise priority is the best, since there are many other ways of giving priority. For instance, we can use a *method-wise* priority, or assign a priority to a *user* who activates the service, or may introduce *dynamic priority* based on the usage of the service. We examine them in our future work. Note, however, that no matter which priority is adopted, the proposed method works as long as $pri(s)$ can be evaluated during runtime.

Another limitation is that all responsibilities with respect to the FIs are concentrated on the centralized FI manager, which may cause the low scalability. However, we consider that the scalability issue in the HNS is currently not so serious as in the telephony networks. As the HNS is deployed for every house locally, every FI manager just takes care of one house with a home server. In the future, the HNS will be inter-worked over multiple houses to achieve a ubiquitous community. In that case, we have to think new types of FIs as well as global coordination among remote FI managers.

5.3. Related Work

Kolberg et al. proposed an online FI detection and resolution method with the HNS [4][12]. They characterize FIs as competitions among resources (i.e., appliances), and presented an FI detection method based on a resource locking mechanism. As for the FI resolution, they basically adopt the AbortAll scheme only without any suspend/resume mechanism. Also, there is no explicit consideration of the three types of activations or mandatory methods. These are major differences from our method.

Pattara et al. presented a formal framework to detect FIs within the HNS, using the SPIN model checker [5]. Unlike ours, this method allows branches and loops in the service description, and can give complete proof within the model. However, this method basically provides the *offline FI detection* for early stages of the development. Also they do not address the FI resolution in the method.

Our approach, which deploys the FI manager in the HNS, is quite similar to the *Feature Manager based approaches* (e.g., [2][3][6]) in the conventional telephony services[2]. However, we consider that the activation management conducted by the manager would be quite different. The activation of a telephony service has been clearly defined within a *call* with subscribed *features*. No such explicit definition exists in the HNS integrated services, which motivated us to define the three types of activations. Also, some HNS services can be quite *long-life*, compared to a single call of telephony. Hence, on resolving FIs, we have to consider more carefully fairness and compensation for low-priority services, rather than just aborting them. The proposed SuspendResume scheme is one of the solutions for that.

6. Conclusion

In this paper, we have presented an online FI detection and resolution method for integrated services of the home network system. Introducing three new notions, i.e., the ac-

[2]More specifically, the proposed method can be categorized in the *feature manager – a-priori information* approaches (see Section 6.3 of [1]), in the sense that the proposed method uses the HNS model given in the design time for the online FI detection. Note, however, that the proposed method can detect and resolve FIs during runtime without a priori knowledge of potential interactions or pre-defined resolution matrix.

tivation, the mandatory methods, and the suspend/resume mechanism, on top of our previous framework, we could achieve more precise and reasonable FI detection and resolution within the context of FIs in the HNS. The case study demonstrated the effectiveness of the proposed method.

We are currently implementing the proposed method in a real home network system. Our primary future work is evaluation and feasibility studies on the implementation. Evaluation of other resolution schemes through usability testings with actual home users is also an interesting issue.

Acknowledgments

This research was partially supported by: [the Japan Ministry of Education, Science, Sports, and Culture, Grant-in-Aid for Young Scientists (B) (No.18700062, No.20700027)], [JSPS and MAE under the Japan-France Integrated Action Program(SAKURA)], and [Panasonic Electric Works Co., Ltd.].

References

[1] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: A critical review and considered forecast. *Computer Networks*, 41(1):115–141, January 2003.

[2] N. Fritsche. Runtime resolution of feature interactions in architectures with separated call and feature control. In *Proc. Int'l. Workshop on Feature Interactions in Telecommunication Systems III (FIW'95)*, pages 43–63, Kyoto, JP, June 1995. IOS Press, Amsterdam.

[3] Y. Jia and J. M. Atlee. Run-time management of feature interactions. In *Proc. 6th ICSE Workshop on Component-Based Software Engineering (CBSE'03)*, pages 115–134. IOS Press, Amsterdam, May 2003.

[4] M. Kolberg, E. H. Magill, and M. Wilson. Compatibility issues between services supporting networked appliances. *IEEE Communications Magazine*, 41(11):136–147, November 2003.

[5] P. Leelaprute, T. Matsuo, T. Tsuchiya, and T. Kikuno. Detecting feature interactions in home appliance networks. In *In Proc. of 9th Int'l Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD 2008)*, pages 895–903, August 2008.

[6] D. Marples and E. H. Magil. The use of rollback to prevent incorrect operation of features in intelligent network based systems. In *Proc. Int'l. Workshop on Feature Interactions in Telecommunication Systems V (FIW'98)*, pages 115–134, Lund, Sweden, September 1998. IOS Press, Amsterdam.

[7] M. Nakamura, H. Igaki, and K. Matsumoto. Feature interactions in integrated services of networked home appliances -an object-oriented approach-. In *Proc. Int'l. Conf. on Feature Interactions in Telecommunication Networks and Distributed Systems (ICFI'05)*, pages 236–251, Leicester, UK, June 2005. IOS Press, Amsterdam.

[8] M. Nakamura, A. Tanaka, H. Igaki, H. Tamada, and K. ichi Matsumoto. Adapting legacy home appliances to home network systems using web services. In *IEEE International Conference on Web Services (ICWS2006)*, pages 849–858. IEEE Computer Society Press, September 2006. Chicago, USA.

[9] M. Nakamura, A. Tanaka, H. Igaki, H. Tamada, and K. ichi Matsumoto. Constructing home network systems and integrated services using legacy home appliances and web services. *International Journal of Web Services Research*, 5(1):82–98, January 2008.

[10] Panasonic Electric Works Co., Ltd. Lifinity. `http://denko.panasonic.biz/Ebox/kahs/index.html`, 2008.

[11] Toshiba Consumer Marketing Corp. Toshiba home network – feminity. `http://www3.toshiba.co.jp/feminity/feminity_eng/index.html`, 2005.

[12] M. Wilson, M. Kolberg, and E. H. Magill. Considering side effects in service interactions in home automation - an online approach. In L. du Bousquet and J.-L. Richier, editors, *Feature Interactions in Software and Communication Systems IX*, pages 172–187. IOS Press, Amsterdam, 2007.