# Modeling and Analysis of Functionality in eHome Systems: Dynamic Rule-based Conflict Detection

Ibrahim Armac, Michael Kirchhof, and Liviana Manolescu
Department of Computer Science 3 (Software Engineering)
RWTH Aachen University of Technology, Ahornstr. 55, 52074 Aachen, Germany
{armac|kirchhof|livianam}@i3.informatik.rwth-aachen.de

## Abstract

*The domain of eHome systems is a special application area for pervasive computing. Many different kinds of devices are introduced to the home area to provide functionality for enhanced comfort or security. A similar level of heterogeneity can be found at the software level: many different vendors supply eHome systems with drivers and services, which intend to compute sensor information and trigger devices in the eHome. This multi-level heterogeneity leads to system faults in terms of deadlocks and unpredictable or disillusioning behavior. We call these error conditions* conflicts. *Pervasive systems, especially eHome systems, will only be useful and thus successful, if such conflicts can be handled properly. In this paper, we analyze eHome systems with respect to types of conflicts and discuss how conflicts can be detected. We will show that the dynamic conflict detection is reasonable and possible by a rule-based conflict detection. The detection is well-founded on a formal specification and is seamlessly integrated into the paradigm of component-based software construction.*

## 1 Introduction

In pervasive computing, one particular segment is gaining more importance due to recent developments in hardware technology. This is the area of *eHome systems*. eHome systems are built on top of integrable net-aware devices in households. Reasonable applications can be identified in the areas of comfort, security, infotainment, communication, and health care. Services of this nature are known as *value-added services* in a broad sense. Talking about services, we mean any piece of software, which is executed in a *networked* environment, making usage and administration of pervasive appliances easier. We call services, which are visible to the users, in all areas above *eHome services*. These eHome services can be autonomous or can be augmented by external services providing information database and services like weather forecast or traffic information.

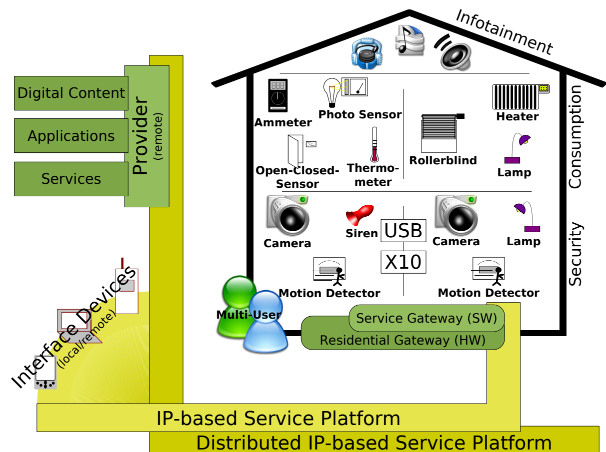


**Figure 1. eHome System Structure**

One very important fact in eHome Systems, in contrast to traditional distributed systems, is that the expected potential of market penetration is extremely high. Over time, many different devices, protocols, services, and providers will come into play. As one can clearly deduce from current problems in computer systems for the masses, it can not be expected, that all devices, services, and providers act cooperatively. This will result either in unexpected and astonishing system behavior or system faults during runtime. This will prevent such systems to be successful in the market. Thus, a way to handle inconsistent actions is needed.

The eHome system structure is illustrated in figure 1: The connected home on the right-hand side of the drawing is equipped with a *residential gateway*, a hardware device, which provides access to connecting infrastructure (e.g., X.10, EHS, Lon, Jini) and acts as runtime environment for the *service gateway*. The service gateway manages and runs *eHome services*. In our work, we focus on the domains of Security, Consumption, and Infotainment.

The services in these domains are based on certain equipment, as some examples are shown in the figure: an alarm system depends on cameras, motion detectors, and lamps or sirens. Monitoring and optimization of energy consumption can be realized by the use of ammeters, photo sensors, thermometers, the heating systems, and so on. Elements of the infotainment domain can be incorporated for audio-based and video-based interaction. The communication backbone is an IP-based platform, including a distributed extension. With this extension, internal and external communication can be handled equivalently. Beside direct interaction with devices in the house, interaction with the eHome system based on personal computers, PDAs, and mobile phones is realized. Service providers are connected to the systems via the distributed IP-based platform to provide digital content, applications, and services.

One often mentioned problem is the existence of many established home automation standards. To cope with this, we use an OSGi-compliant gateway [3] as the nerve center of our solution. The usage of the open service gateway enables an abstract, almost protocol-independent view onto the different home automation protocols used. We are more interested in the mechanisms of *composite services*, rather than in the mechanisms of *local communication and service development*. To ease the development of eHome services, we rely on the layered approach to OSGi-based service gateways PowerArchitecture [11], the rule-based declarative specification of eHome services PowerLogic [12], and the Distributed Services Framework (DSF) [10].

Conflicts inevitably occur in a multi-service environment, which can be illustrated by analyzing the following *integrative scenario*. The system provides basic services for controlling the illumination and the temperature in each location, the windows, the jalousies, the roller blinds, and the doors, as well as services controlling electric devices like oven, microwave, coffee maker, washing machine or iron. A multimedia gateway to which the TV, the DVD, the recorder, and the music equipment are connected is also available. The house has movement sensors in every room, open-close and glass breakage sensors at windows, as well as smoke and fire detectors. The system provides also a service for identifying if somebody is present or not in the house. Apart from this basic service infrastructure, five eHome services are installed in the eHome: LIFE STYLES, MUSIC FOLLOWS PERSON, SMART AGENDA, PREVENTION OF DANGEROUS SITUATIONS, and WAKE-UP. The LIFE STYLES *service* belongs to the area of comfort. It allows defining and grouping user personalized ambiance preferences into *mood profiles* or *life styles*. Each inhabitant of the house can specify illumination or temperature parameters, as well as the favorite music according to the mood. The MUSIC FOLLOWS PERSON *service* adds functionality to infotainment in an eHome: a music service that follows the person wherever he or she walks through the house. The SMART AGENDA *service* allows a centralized scheduling of a variety of tasks concerning the programming of devices and home appliances. Each user can plan the activation/deactivation of service functions whenever needed (e.g., programming of home appliances or activation of life styles). The agenda deals with both punctual and durative time events. The PREVENTION OF DANGEROUS SITUATIONS *service* aims at continuously monitoring the house and informing the inhabitants or the authorities of intrusion cases, fire, flood, or other dangerous situations. This is practically an extension of a classical security service with more functions beside surveillance. The WAKE-UP *service* targets the comfort enhancement of the inhabitants. Its purpose is to calculate an optimal wake-up time, taking into account various static and dynamic factors (e.g., the first appointment in the day, typical duration of breakfast or shower, information about traffic jams or blocked streets). This service does not only estimate the wake-up time, but also automatically turns on the heating system to reduce waste of energy and activates the wake-up mood for maximized comfort.

Having in mind such a complex scenario, the problem of conflict detection is set into a new light. It is obvious that, under the assumption that they are all running simultaneously, they might interfere in a conflicting way. The integrative scenario gives already an insight into possible *conflict* situations. Conflicts may occur within the rule-based services or among service components. For instance, by analyzing the user settings in the life styles profiles, it might come to cases when they are mutually exclusive for two people being simultaneously in the same location. This happens, for instance, to the MUSIC FOLLOWS PERSON service for two people in the same room. If all users had equal rights, there would be no conflict since the preferences of the last person to enter the room would always override the previous settings. But just imagine that each person would have an importance level. For example, the father's and mother's options are more important than those of children. In this case, if the child enters the room after his parent, the MUSIC FOLLOWS PERSON service can not switch to the child's preferences. Of course, it can be argued that such situations are only *social conflicts* and is not the responsibility of the eHome system to deal with them. Here naturally comes up the question: *to which extent* can a system recognize conflicts and, more important, *what kind of conflicts* can be detected?

Current situation is, that the complete functionality is coded into eHome services in a black-box style. The dependencies and preconditions are hidden in the implemented components, such that proper operation can not be guaranteed. In this paper, we will propose a solution to this *problem of conflict detection*. Once a conflict has been de-

tected, a conflict resolution strategy can be used to ensure proper service execution and transparent system behavior. To reach this goal, we profit from the well-defined software architecture for eHome systems PowerArchitecture, add a monitor for service actions, and introduce a solution to dynamic conflict detection in eHome systems. We will show, how conflicts can be formalized and automatically detected by a rule-based specification of conflicts.

# 2 Approach

Many software designers aim at developing internally consistent components and systems having an overall coherent behavior. Nevertheless, unwanted interactions or conflicts among the elements of a system are in many cases unavoidable. When put together, internally consistent components interact inevitably and oppose sometimes their behavior to that of others in a conflicting way. Also, consistent systems are conflict-prone when exposed to versatile environments and users. It can be affirmed that some general premises for conflict situations are (1) *concurrency* on the same, let's call it abstractly, entity and (2) *overlapping* of incompatible features, requirements, or behaviors. By refining the various causes which generate concurrency or overlappings in different application domains, as well as by giving various acceptances to the objects of a conflict, many domain-specific meanings of conflicts can be derived.

For eHome systems, the plurality of services simultaneously accessing and demanding for devices and appliances causes unwanted service interactions. Due to this, the objects of conflicts in eHomes are *concrete devices* and *home appliances*, as well as *environment variables* like temperature, illumination or sound level. These are concurrently accessed or influenced by service actions performed in different contexts and on behalf of various users. Overlappings of incompatible requirements usually occur because of additional restrictions imposed to service actions or contradictory configuration information. As users want to understand what happens in their system and why sometimes their actions do not have the consequences they expect, the conflict detection plays an important role in the eHome context.

## 2.1 Conflict Classification

A good *general* understanding of what a conflict is may not be enough for a successful detection process. That is because for each application domain, the meaning of this concept is different and needs to be precisely defined. In the eHome context, we identified several *conflict types* by applying an *inventory method*.

What we practically did was to think about as many different devices and household appliances as one could possibly have in his/her home. Thus, about forty of them have been identified. Around two thirds have an *actor* behavior, in the sense that they can be used for concrete activities and tasks. The remainder have a *sensor* behavior, providing essentially context and environmental information. The next step was to make an *inventory* of what can be done with each of them. In the following step, as many as possible *combinations* of such functionalities, *desirable smart features* or *mini scenarios* have been listed. The preferred composition form was that of *if-then* causalities. During the listing process, it has been tried to let the ideas' flow uncensored, accepting thus also some unreasonable combinations. About sixty of such use cases have resulted. After that, we tried to identify which conflicts could appear if *all* of them were available at the same time on the service gateway. By analyzing the discovered conflict situations, it turned out that they are relatively similar and can be easily classified into several types.

We use a self-defined notation to formalize the conflict types. This purpose-build language represents a modeling language with a small number of elements in order to ease the specification of conflicts. In contrast, using a more general language (e.g., first order logic) would be possible but would complicate the specification unnecessarily. Here, only the parts of the notation which are necessary to understand the examples will be explained "on-demand". There are four of such types which will be explained in the following. Due to limited space, we will apply the notation only for one type. The remaining ones will be explained in a non-formal description and an example.

**Type 1:** CONFLICT ON SUPER-STATE BLOCKED

**Description** This first type is mostly applicable to devices that require *exclusive* usage while they are performing a task and should be by no means interrupted by other actions.

**Formal Specification**

$$(\forall)R \in \Re^D,\ S \in \Sigma,\ a \in A^S\ if\ (\exists)S' \in \Sigma \backslash \{S\},\ b \in A^{S'}\ s.t.\ S'.b \prec S.a \wedge R(*) \stackrel{S'.b[B+]}{\longrightarrow} R(i) \wedge \neg[R(i) \stackrel{S.a[B-]}{\longrightarrow} R(*)] \Rightarrow S.a \longrightarrow \times \longleftarrow S'.b$$

**Explanation** The above listed formula describes the first conflict type using the aforementioned notation. $R \in \Re^D$ describes a resource $R$ belonging to the set $\Re^D$ of all resources in form of physical devices and appliances in the eHome. $S \in \Sigma$ describes an eHome service running on the service gateway where $\Sigma$ is the set of all services running on the service gateway. With $a \in A^S$, we mean an action $a$ of the basic services $S$ from the set of all its conflict relevant actions (that have a direct impact on the state of a resource). The notation $S'.b \prec S.a$ means that the action $b$ of service $S'$ chronological precedes the action $a$ of service $S$. Fur-

thermore, $R(*) \stackrel{S'.b[B+]}{\longrightarrow} R(i)$ stands for the transition $S'.b$ of type $B+$ changing the state of resource $R$ from any state $*$ to state $i$. Thereby, a transition of type $B+$ blocks a resource from being accessed by other services. Respectively, $B-$ unblocks a previously blocked resource. Last but not least, the notation $S.a \longrightarrow \times \longleftarrow S'.b$ symbolizes that the two actions $S.a$ and $S'.b$ are in conflict. The readers should be familiar with the remaining symbols in the formula so that is now understandable with the following explanation. A service action $S.a$ produces a conflict with respect to a resource $R$ of device type, if there is a service $S'$ which previously executed an action $S'.b$ inducing a transition of type $B+$ (blocking) with respect to that resource and the current action is not generating an unblocking transition or does not belong to the same service.

**Example** An eloquent example for this conflict type is that of a multimedia device which is scheduled to record two different shows on overlapping intervals. Consider the situation where the device starts recording the news from 20:00 to 20:15 and, thus, is blocked. If another task tries to record a favorite movie from 20:05 to 21:30, a conflict of type 1 will occur because the video channel is already blocked by the first task.

**Type 2:** CONFLICT WITH SUPER-SERVICE

**Description** This conflict situation occurs among services that use *shareable* resources (whose state can be modified) but have been assigned priorities or importance levels and, consequently, have a restricted action domain. For this conflict type, transition types for shareable resources are relevant. A given state of these resources can be modified by services having enough access rights such as higher priority. For shareable resources, we differentiate between "positive" ($S+$) and "negative" ($S-$) transition types. A service action $S.a$ produces a conflict with respect to a resource $R$ of device type, if there is another service which previously executed an action inducing a transition of type $S+$ or $S-$ with respect to that resource and the current service has been granted a lower priority than its predecessor. The resource is released from the conflict state either by a service with a higher priority or automatically at the deactivation of the latest service that accessed it.

**Example** A typical case for such a conflict type is that of the PREVENTION OF DANGEROUS SITUATIONS and LIFE STYLES services described in section 1. Assuming that the first high-priority service lets down the roller blinds for trapping the intruder inside and this action is configured as a transition of type $S-$ for the roller blinds, the activation of a life styles profile (low-priority) that lifts up the blinds will be considered a conflict.

**Type 3:** CONFLICT ON ENVIRONMENT VARIABLE

**Description** Some service actions do not only change the status of a device but also influence *environment variables* like temperature, illumination or sound level of the location where the resource can be found. Essentially, these variables are of quantitative nature and their value can be increased (transition type $V+$) or decreased (transition type $V-$) by executing service actions. Conflicts on environment variables occur if some services "fight" with other higher priority services for changing their value. A service action $S.a$ produces a conflict with respect to a resource $R$ of type environment variable, if there is another service $S'$ which previously generated a transition of type $S+$ or $S-$ on a device resource $R'$ and of type $V+$ or $V-$ on an environment variable $R$ and the current service action induces on the variable $R$ also a transition of type $V+$ or $V-$ but this service has been granted a lower priority than its predecessor. The resource is released from the conflict state as explained for conflict type 2.

**Example** A situation when this conflict type occurs is, for instance, that of a LIFE STYLES service, requiring to change the temperature in a room when activating a mood profile and an optimized HEATING SYSTEM CONTROL service. The former tries to set up a temperature of 25°C, whereas the latter suggests an optimally calculated temperature of 22°C. In the assumption that the second service has a higher priority than the first, the activation of the life styles profile creates a conflict situation.

**Type 4:** CONFLICT WITH SUPER-USER

**Description** This conflict category is typical for the situation when actions belonging to the *same* service are performed by *users* with different priorities and reflect upon the same resource. Unlike the other previous three cases, this type takes into consideration the influence of the human factor behind service actions. The general assumption is that the users are assigned some importance levels or priorities. Even if this may not be always justified, there are some cases when ordering user rights is needed, for instance, for restricting the actions of little children on devices when parents are not at home, or for distinguishing between authorized family members and unauthorized visitors. A service action $S.a$ performed by an user $U_i$ generates a conflict with respect to a resource $R$ of device type, if there is another user $U_j$ who has previously executed an action $S.b$ belonging to the same service and affecting the same resource $R$ and the current user has been granted a lower priority than his predecessor. The resource is released from the conflict state either by a user with a higher priority or automatically at the deactivation of the latest service that controls it.

**Example** One typical example for this conflict class is that of a multi-user service like MUSIC FOLLOWS PERSON. In this case, if two users enter simultaneously the same room, since the audio channel can play only one stream at a time, a conflict will occur. Moreover, supposing that priority levels are granted and that the system plays first the choice of the user having the highest priority, a conflict will occur when trying to play the choice of the second person.

## 2.2 Completeness of the Types

The above introduced types are meant to facilitate and clarify the meaning of conflicts in the eHome context. It can not be claimed that this classification is exhaustive and covers all the possible conflict types, but we assume that it is general enough to cover most of the cases listed by applying the inventory method. One situation that is not considered is the concurrent use of resources by services with different priorities acting in the name of users having also different priority levels. This case has been left aside because of the reason that it is not easy to decide who is more important: the service or the user in the name of which the action has been performed. Naturally, it can be said that every service is an agent acting in the name of an user and the user should be more important than the service. However, there are also cases when the services should be granted more importance than their users, especially when the service actions have been performed by mistake or from ignorance. Consider the situation of a home owner and a guest both using the PREVENTION OF DANGEROUS SITUATIONS service, the home owner having a higher priority than the guest. In this case, it is reasonable that the home owner should be able to activate, deactivate, and reconfigure this service offhand. Thus, the home owner has more power than the PREVENTION OF DANGEROUS SITUATIONS service. In contrast, the guest should use this critical service only in emergency cases so that he has to authorize adequately and confirm his actions to avoid unmeant or unauthorized activities. In this case, the service is more important than the guest user.

## 2.3 Conflict Detection Strategy

An important aspect for the conflict detection is to make clear *when* and *where* to perform it, as this might have a significant impact on the accuracy, validity and usefulness of the results, as well as on the computational effort dedicated to it. Depending on the moment in time when conflicts are identified, two different techniques can be used: a *static* and/or a *dynamic* one. The *static* detection is done, mostly, in the design and specification stage and analyzes the elements of a system and their interactions based on a formal description of their behavior. It can also be performed on a running system, but is essentially an independent task, which uses a snapshot of the system at a certain moment and performs the detection on this system "image". The *dynamic* alternative is a process running in parallel to the system and continuously taking information from the system environment, working thus on an up-to-date image of the system. The dynamic detection monitors practically what happens *now and here* and reasons based on the state of art of the system.

Our *initial idea* for a conflict detection approach was the static analysis. This was inanimated and influenced by the *rule-based paradigm* used for implementing eHome services in the eHome Group [12]. As rules express the behavior of these services, the idea was to analyze what happens within the rule sets, if there are overlappings and contradictions and if they generate conflicts. However challenging and interesting this idea might sound, this approach turned out to be inadequate for the eHome context, mainly because of the fact that *these rules are not a sufficient and complete formal specification for performing the static conflict detection*. In fact, the lack of such a specification for the services developed in the eHome Group is strongest *pragmatic argument* against the static alternative. The dynamic approach has a winning cause, considering also that *not all* eHome services are developed using the rule-based paradigm but might be implemented also using usual imperative programming concepts [12].

## 2.4 Dynamic Conflict Detection

The idea behind the dynamic conflict detection starts with a *first assumption* that each resource type can be modeled by an $\omega$-automaton [5] (figure 2). For instance, a lamp is an $\omega$-automaton with two states: *on* and *off* and three transitions: *set_intensity_level*, *turn_on* and *turn_off*. Similarly, an audio device is an $\omega$-automaton with four states: *started*, *playing*, *paused* and *stopped*. The audio device switches from "started" to "playing" with the *play* transition, with *pause* and *restart* between "playing" and "paused" and goes from "playing" to "stopped" using the *stop* transition. The transition *set_volume_level* does not change the state of this resource type. These two examples are kept small and simple, but for other resource types the $\omega$-automata can reach even higher degrees of complexity.

A *second assumption* is that each basic service controlling a resource type provides a one-to-one mapping to the transitions in the $\omega$-automaton associated to the resource. This means, for instance, in the case of the resource type "lamp", that the basic service behind it provides at least three methods - "turn_on", "turn_off" and "set_intensity_level" mapping the transitions in the $\omega$-automaton. This assumption puts restrictions on basic services in the sense that the mapping functions have to be *elementary* in order to mirror just *one* transition. For ex-

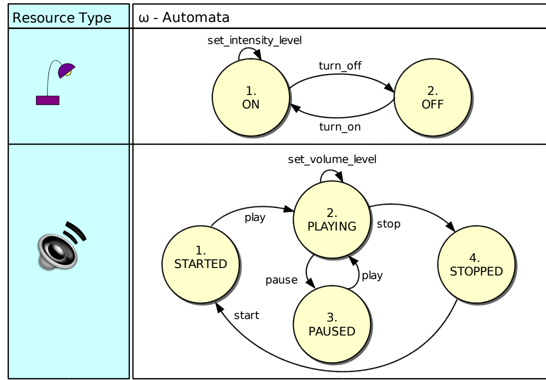


**Figure 2. Representation of resource types as $\omega$-automata**

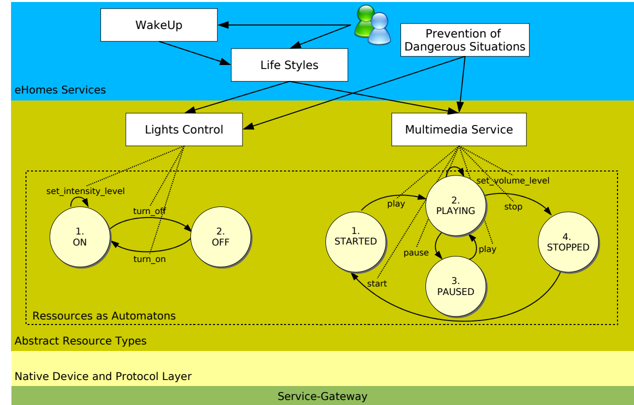


**Figure 3. Conflict detection integrated into** `PowerArchitecture`

ample, a "toggle" function, that switches the lamp "on" and "off" alternatively, is *not* an elementary function, because it illustrates a combination of two other functions and can be mapped to a succession of "turn_on" and "turn_off" transitions.

For the dynamic conflict detection, it is *not* relevant to have the whole information illustrated by the $\omega$-automaton. Only the transitions are important. Moreover, not even the transitions themselves are important, but the *type* the transitions belong to. That is because transitions might induce *state changes* and these can lead to conflict situations when differently ranked services produce such changes. Transition types are more relevant than transitions themselves, because they express in an abstract way properties of transitions like: what kind of resource they refer to, if they are blocking or not for the resource, if they induce a shareable state and so on. Furthermore, no complete $\omega$-automaton for each resource type is available in reality. Only the service methods and their mapping to the transitions of an $\omega$-automaton are available and observable.

For the understanding of the dynamic approach it is important to clarify the *reasons* behind conflicts. The main cause is the fact that functionalities provided by basic services are used by a *plurality of services* in a *variety of contexts*, and in the name of *different users*. This multiple usage, combined with a limited availability of resources and with additional restrictions imposed to services and users lead to conflicts. An example of multi-dependencies between users, services, and resource types is illustrated in figure 3 based on the `PowerArchitecture` described in [11]. This architecture decouples the *service layer* from the *native device and protocol layer* through the introduction of an *abstract device layer* between them. The services on the top level use, thus, abstract device functionalities provided by the middle layer instead of concrete device functionalities.

In figure 3 the native device and protocol layer from the original architecture is not shown in detail, the abstract device layer has been split up into resource types and $\omega$-automaton-representation of the basic services and the service layer is reflected by the eHome services. On this layered architecture, it can be seen that eHome services act in the name of several users and use functionalities provided by basic services inducing, thus, state transitions on the associated resources. All services are running in parallel and request, in the worst case, simultaneous access to resources and even try to execute contradictory actions. For instance, the PREVENTION OF DANGEROUS SITUATIONS service detects an intruder at night and consequently uses the MULTIMEDIA service to raise an alarm, and turns on all the lights in the house. Shortly after, a preprogrammed "night" mood is activated by the LIFE STYLES service. The night mood turns off all the lights in the house and stops the sound system. These actions cancel the effects of the PREVENTION OF DANGEROUS SITUATIONS service and, thus, induce a conflict. This is just one possible conflict situation, but in a real eHome system, the area of such conflicts is much larger. However numerous the conflicts may be, the reasons behind their occurrence are essentially the same and point to *concurrency of services on resources*.

## 3 Implementation

For the detection of the conflict types explained in the previous section, a flexible and scalable software architecture has been developed (figure 4). This consists in the following main elements: (1) a *service settings* component, that allows adding extra-semantic to services and their methods and (2) an *interceptor* service, that keeps track of the method calls and forwards detailed information to the (3) *conflict monitor*, a rule-based service that performs the

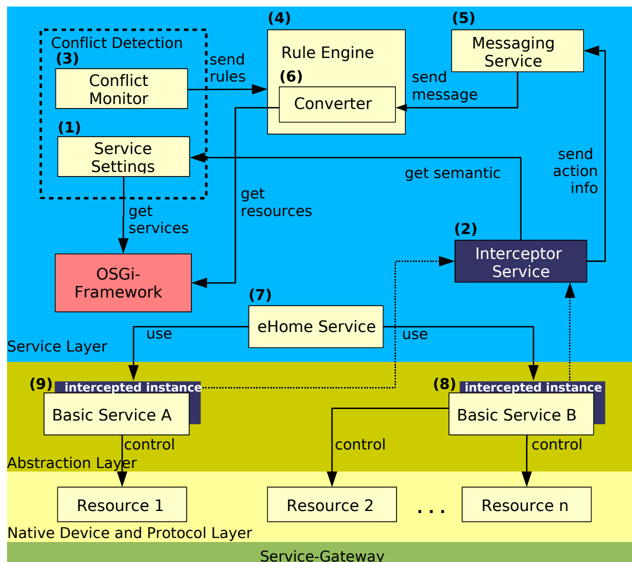


**Figure 4. Conflict detection architecture: global view**

conflict detection process according to the specification introduced in section 2.1. Apart from them, some other already available components are used, the most important of them being (4) the *rule engine* [12], that evaluates and triggers the rule set of the conflict monitor, and the (5) *messaging service*, that enables the communication between the interceptor and the conflict monitor. The rule engine provides a (6) *converter*, that automatically translates the information on the communication channel into the data structures required by the conflict monitor. Interesting for the conflict detection process are especially the (7) *eHome services* because they incorporate compositional logic and make use of functionalities offered by (8) *basic services*. The last mentioned services stay behind concrete physical devices and home appliances (designated here by the general term of *resource*).

The general assumption of our approach is that most of the eHome devices are accessible and controllable through software. Behind them are the basic services whose functionalities can be used by the eHome services in composition rules. For the conflict detection process, it is relevant to keep track of the end methods executed by eHome services, because these have a direct impact on the state of the physical resources. For this purpose the *interceptor* service has been developed. Each eHome service making *use* of other basic services may *get a reference* of the interceptor and execute method calls in the context of *intercepted service instances*, instead of actual ones. For the rule-based services, the usage of the interceptor happens automatically since they all use a generic method provided by the rule engine to call methods provided by the basic services. The mechanism used by the interceptor is that of creating instances of Java dynamic proxy classes. Unlike a regular instance, each intercepted instance has enhanced capabilities, being able, on the one side, to *get* additional *semantic* about the invoked service method from the service settings component and, on the other side, to *forward action*-related *information* through the messaging service to the specialized converter in the rule engine and thus, indirectly, to the conflict monitor. The messaging service is a communication channel were messages are published, from where interested subscribers like the converter can *get information* addressed to them. The converter has the function of transforming the received interception messages into the data structures (in our case Jess [7] facts) necessary for the conflict detection monitor. The converter ensures also that the conflict module is aware of the dynamics of all devices and home appliances registered as resources into the system. Thus, the conflict monitor is aware of both the *current* status of the eHome devices and of the *desired* status induced by the intercepted method calls. Because of its rule-based implementation, the conflict monitor makes use of the rule engine by sending its rule set to it. Further on, the rule engine overtakes the responsibility of evaluating and triggering the conflict rules whenever needed.

As far as the concrete implementation is concerned, we would like to explain in more detail the three main components of the conflict detection architecture.

### 3.1 The `ServiceSettings` Component

One of the main impediments for the dynamic conflict detection is the lack of semantic information about the installed eHome services. After all, that is not surprising since eHome services are nothing but components and they follow principles of *component-based software engineering* (CBSE). By definition, a component is an opaque implementation of functionality, subject to third-party composition and conformant with a component model. The *opaqueness* of components is caused on the one side by third-party providers, which see software as black boxes and want to protect their intellectual property from discloser, and, on the other side, by inherited concepts from the programming world like abstraction and information hiding. The lack of transparency is exactly the obstacle in the way of a successful dynamic conflict detection. Without appropriate information about services, their methods and the way they reflect upon resources, no conflicts could be discovered. The `ServiceSettings` component was developed especially for compensating the opacity hiding the semantics of services. The aim of this component is to discover the installed basic services and offer a flexible solution for specifying the semantics of their actions. The main assumption

is that basic services are so modularly designed that each method invocation produces only one transition on the automaton of the associated resource. This is a reasonable premise since functionalities of basic services do not contain any logic in themselves, but just map simple functions of devices like start, stop, turn-on, turn-off etc. The only constraint that basic services have to fulfil in order to be taken into consideration by the settings component is to *group the conflict-relevant actions into a predefined interface*. This restriction is an architectural constraint for the basic services to be detected and automatically integrated in the conflict detection process.

### 3.2 The `Interceptor` Service

The interceptor approach is not new in software engineering. The *Chain of Responsibility* behavioral design pattern [8] suggests a similar practice. A variant of this pattern has been described in [15] under service actions and configuration patterns. Its intent is to enhance the flexibility and extensibility of a software system by letting applications add to the base system's functionality and also dynamically change its subsequent behavior. In the context of our software architecture, the interceptor facilitates the integration of an arbitrary number of basic services into the conflict detection process, if they respect some service registration conventions. At the same time, by forwarding the service method invocations to the conflict monitor and letting it take over the execution control, it changes the subsequent behavior of the system. The interceptor service provides essentially two methods for interacting with the available services: `newInstance`, that creates the intercepted service instances and `invoke`, that is automatically called when a regular service method is invoked. This latter method catches practically the service method calls, enriches them with the configuration information provided by the `ServiceSetting` component and forwards the invocation intention to the `ConflictMonitor` for further analysis.

### 3.3 The `ConflictMonitor` Service

The `ConflictMonitor` is the central component of the conflict detection process. Its goal is to analyse the method calls that are about to be performed and according to their semantics, the conflict definitions, and the status of the involved resources decide whether a conflict will occur. This service has a global view on the installed services and the resources distributed in the eHome environment, that turns it into a permanent monitor of the system state. The service itself is a rule-based component containing *initialisation* rules (for the first execution of a method upon a device), *detection* rules (according to the identified conflict types) and *execution* rules for all the other cases when a conflict situation does not occur. The rules have a similar left-hand side structure that compares the current status of a resource with the intended status expressed through the interception messages. The conflict rules differ on the right-hand side in that the execution of the service method is simply not performed, the user being notified that a conflict occurred. As an example the detection rule for conflict type 1 is illustrated in listing 1.

```
1  (defrule conflict_type_1
2   "detects conflict of type 1"
3   ;for any resource fact
4   ?res <-(resource
5      (resource_id ?res_id)
6      (resource_type ?res_type)
7      (transition_type ?trans_type)
8      (service_id ?service)
9    )
10  ;if interception event exists
11  ?iev <-(interception_event_device
12     (resource_id ?res_id)
13     (new_transition_name ?new_transition)
14     (new_transition_type ?new_trans_type)
15     (service_id ?new_service)
16     (service_priority ?priority)
17     (timestamp ?ts)
18  )
19  ;if valid conflict conditions
20  ;1.  resource type device
21  (test
22    (compareConst ?res_type RESTYPE_DEVICE))
23  ;2.  previous transition B+
24  (test (compareConst ?trans_type TT_B_PLS))
25  ;3.  new transition is not B- from same serv.
26  (test
27   (not
28    (and
29     (compareConst ?new_trans_type TT_B_MNS)
30     (eq ?service ?new_service)
31  )))
32
33  =>
34  ;send user notification event
35  (bind ?desc (new java.lang.String))
36  (bind ?desc (str-cat (hrDate ?ts) ":
37       conflict on " ?res_id))
38  (bind ?props (new java.util.Properties))
39  (?props setProperty "type" "conflict-msg")
40  (?props setProperty "description" ?desc)
41  ; send message
42  (?powerLogic sendTextMessage
43       "notification-Event"
44       "conflict-message"
45       ?props
46  )
47  (printout t "Jess notification event sent")
48  ;delete the interception event
49  (retract ?iev)
```

**Listing 1. Detection rule for conflict type 1**

This rule has been developed according to the formal specification presented in section 2.1. The rule tests for every device resource (lines 4-9), if there is a service action (expressed through an interception event in lines 11-18) about to be performed. A resource fact is automatically joined with the interception event through the $?res_id$ attribute. If the previous transition was of type B+ and the current transition is not of type B- belonging to the same service, then a conflict of type 1 occurs. The user receives in this case a notification (`sendTextMessage`) that can be visualized on the graphical interface of the `ConflictMonitor`. For the remaining conflict types similar rules have been defined.

The conflict detection module has been tested on the integrative scenario described in section 1. For realistic testing conditions, a component providing basic services for controlling almost all usual home appliances has been developed. The implementation of this component did not go into protocol and device access details, since the purpose was just to provide such services to the eHome services on the upper level. For each conflict type we developed several test cases involving two or more eHome services. The results confirmed the viability of our detection strategy. Nevertheless, more elaborate tests on an smart home demonstrator are part of further work.

# 4 Related Work

This section presents related research on the topic of conflict detection covering the areas of policy-based management systems, multi-agent systems, and telecommunication systems.

## 4.1 Policy-based Management Systems

Research in this field spans both the area of *static* [13, 14] and *dynamic* [4] detection. Similar to the rules of the eHome services, policies are pieces of information able to modify the behavior of a system without coding the behavior into manager objects. A policy has a subject and a target. The *subject* specifies the human or automated managers to which the policy applies. The *target* defines the objects on which the policy actions may be performed. Management policies can express either *authorisation* (A) or *obligation* (O). By associating a *mode*, it can be distinguished between positive (O+) and negative (O-) obligations, and positive (A+) and negative (A-) authorizations. Policy conflicts arise when two or more policies with modalities of positive and negative signs refer to the same subjects, actions and targets. The static approaches detect conflicts by performing a syntactic analysis of policies at specification time. The dynamic approach uses similar policy types and enhances the policy specification with *temporal* characteristics that allow the runtime detection. The conflict detection algorithms use a *conflict database*, where definitions of conflict types are stored. Each time a new policy is added to the system, a computation process is started and, based on the available definitions, runtime conflicts are detected.

Unlike policies, the eHome rules do not have associated types and their structure does not allow the explicit definition of subjects and targets. For these reasons, the policies approach can not be directly applied to eHome services. Nevertheless, the idea of policy types has been adopted and applied to defining the typology of transitions on the resource's $\omega$-automata. Similar to the policy modes, the overlapping of transitions having incompatible types might lead to conflict situations. From the dynamic approach, we borrowed the idea of conflict definitions. In contrast to the discussed approach, our solution does not use a conflict database, but formulates the conflict definitions as *rules* evaluated at runtime by the conflict monitor.

## 4.2 Multi-Agent Systems

A *multi-agent* system can be seen as a group of entities interacting to achieve individual or collective goals. To facilitate the conflict detection in such systems, a representation of the dependencies among agents' *goals*, *plans*, and *beliefs* is presented in [2]. Conflicts occur at all three levels with respect to the resources involved in the agents' activities. Of special interest are *plan* conflicts [1]. The agent plan space denotes the scheduling of resources and is represented with help of E-PERT diagrams (Extended Project Estimation and Review Technique). By analyzing these diagrams, it can be easily determined when and what resources are shared by multiple agents. As a resource can be only exclusively used by an agent, plan conflicts occur whenever two ore more agents schedule the same resource at the same time.

The eHome services can be also regarded as agents acting on the resources in the smart home environment. While the agent-based approach presumes the exclusive use of any resource, our approach affirms that a resource can be exclusively used only after a blocking (B+) transition. In all the other cases the state of a resource is modifiable depending on user and service priorities. The actions of eHome services are similar to agent plans, since both describe an execution logic. But the succession of service actions is not at all linear and their durations can not be estimated beforehand. For this reason, the approach of the E-PERT tables is inappropriate for our work.

## 4.3 Telecommunications Systems

Related approaches regarding conflict analysis at specification time can be found in the area of telecommunications

systems. Such approaches introduce formal specification languages for describing telecommunication features and describe methods for automatically detecting feature conflicts at an early stage [6] or composing these features such that no conflicts occur [9].

The approaches in this area benefit from the powerful theoretical background offered by methods of formal specification, verification, and validation of software. They also target a domain where standards for system features are available and can be easily translated into a new formalism. However, they do not find a direct application in the eHome field. Even if eHome services could be regarded as a collection of features, the diversity of services and their higher complexity makes formalization attempts based on these two approaches inadequate. One could apply the main underlying principles, but not exactly the same notations or the formalism that they use. For this reason, we have introduced a notation of our own and concentrated on the meaning of conflicts in the particular context of eHome systems.

## 5 Summary and Outlook

In this paper, we motivated the conflict detection for pervasive systems, especially for eHome systems. We showed how the different kinds of conflicts can be formalized and that this formal description can be transformed into rules. These rules were used to dynamically detect conflicts during runtime. The detected concrete conflicts have been dealt with a simple but reasonable conflict resolution strategy. The proposed approach has been completed by the seamless integration into the component-based paradigm. Thus, a sophisticated and powerful approach to conflict detection and conflict resolution conform to existing standards has been developed.

The proposed approach can be the basis for further work. First, the integration during runtime can be changed, if the triggering of actions is handled solely in the rule engine. Thus, the intercepting of basic services would be no longer necessary. Second, also the design time and development time of eHome services can be augmented by conflict detection. This will require research in the area of static conflict detection to compute potential conflicts already in this early stages. Similarly, the conflict free composition of services can be taken into account, too. Third, the conflict resolution can be extended to handle different situations properly. Finally, the considered dimension of eHome systems can be extended from the local home to other multiple environments in order to reflect a person's private and business life. In such a wide-spread system, the types and number of conflicts may increase. A proper handling of conflicts will then respect the principle of the least astonishment.

## References


[1] K. S. Barber and T. H. Liu. Conflict detection during plan integration based on E-PERT diagrams. In *AGENTS '00: Proceedings of the fourth international conference on Autonomous agents*, pages 106–107. ACM Press, 2000.

[2] K. S. Barber, T. H. Liu, A. Goel, and C. E. Martin. Conflict representation and classification in a domain-independent conflict management framework. In *AGENTS '99: Proceedings of the third annual conference on Autonomous Agents*, pages 346–347. ACM Press, 1999.

[3] K. Chen and L. Gong. *Programming Open Service Gateways with Java Embedded Server Technology*. Addison-Wesley Professional, 2001.

[4] N. Dunlop, J. Indulska, and K. Raymond. Dynamic conflict detection in policy-based management systems. In *EDOC '02: Proceedings of the Sixth International ENTERPRISE DISTRIBUTED OBJECT COMPUTING Conference (EDOC'02)*, pages 15–26. IEEE Computer Society, 2002.

[5] B. Farwer. Omega-Automata. In *Automata, Logics, and Infinite Games*, volume 2500 of *LNCS*, pages 3–21. Springer, 2002.

[6] A. Felty and K. Namjoshi. Feature specification and automated conflict detection. *ACM Transactions on Software Engineering Methodologies*, 12(1):3–27, 2003.

[7] E. Friedman-Hill. *Jess in Action*. Manning Publications Co., July 2003.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[9] J. Hay and J. Atlee. Composing features and resolving interactions. In *SIGSOFT '00/FSE-8: Proceedings of the 8th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 110–119. ACM Press, 2000.

[10] M. Kirchhof. Distributed and Heterogeneous eHome Systems in Volatile Environments. In S. Weerawarana, editor, *Proceedings of Forum at $2^{nd}$ International Conference on Service Oriented Computing (ICSOC 2004)*, number RA221 W0411-084 in IBM Research Report, pages 123–131. IBM, 2004. Refereed Papers.

[11] M. Kirchhof and S. Linz. Component-based Development of Web-enabled eHome Services. *Personal and Ubiquitous Computing Journal*, 9(5):323–332, 2005.

[12] M. Kirchhof and P. Stinauer. Service Composition for eHome Systems: A Rule-based Approach. In S. K. Mostéfaoui and Z. Maamar, editors, *Ubiquitous Computing - Proceedings of the $2^{nd}$ International Workshop on Ubiquitous Computing (IWUC 2005)*, pages 28–38. INSTICC Press, 2005.

[13] E. C. Lupu and M. Sloman. Conflicts in Policy-Based Distributed Systems Management. *IEEE Transactions on Software Engineering*, 25(6):852–869, 1999.

[14] A. Majidian. A model for static conflict analysis of management policies. *BT Technology Journal*, 17(2):53–59, 1999.

[15] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*, volume 2. John Wiley & Sons, Inc., 2000.