Dependency Management in Smart Homes

Daniel Retkowitz and Sven Kulle

Department of Computer Science 3 (Software Engineering)
RWTH Aachen University
Ahornstr. 55, 52074 Aachen, Germany
retkowitz@i3.informatik.rwth-aachen.de
svekul@i3.informatik.rwth-aachen.de

Abstract. In future smart homes functionality will be provided to the inhabitants by software services decoupled from the underlying hardware devices. While this will enhance flexibility and will allow to provide cross-functionalities across multiple devices it will also lead to resource conflicts. Future devices will provide basic functionalities which are used by separate higher level services. Each person will use a number of different services and each environment can be inhabited by multiple users at the same time. All respective services have to be executed based on a limited number of devices, which will result in resource conflicts. In this paper we describe how we extended our existing dependency management approach for smart home services with a mechanism for monitoring service bindings and handling access control based on priority groups.

1 Introduction

Today computing is associated with desktop or laptop computers. Research in the field of ubiquitous computing aims at integrating small computing devices into almost anything that surrounds us in our everyday life. While being unaware of the individual devices, users will be supported in different ways by software services running on such devices. Smart Homes, or *eHomes* as we call them, are smart environments based on the ubiquitous computing paradigm especially focusing on home environments, e. g. private residential buildings.

Component-based software engineering allows to reuse existing software easily and facilitates dynamic composition. These characteristics are especially important for eHome systems. To make low-cost eHomes available to a broad market, ready-made components are used that are composed dynamically according to the user's needs and the current context. Typically an underlying middleware infrastructure is used to support the development of service components. By making use of middleware technology, the service development effort can be greatly reduced as the developers can focus on the services' application logic instead of implementing infrastructure functionality, e. g. life cycle and dependency management, time and again for each service. This way service development is simplified and development costs are reduced. In addition, more consistency is gained by moving cross-functionality to the middleware layer.

The idea of ubiquitous computing implies a separation of application functionality and the devices used for realizing this functionality. Todays consumer electronics are typically based on a tight coupling of functionality and hardware. Either the functionality is directly implemented in hardware or it is implemented as an embedded system, i. e. a specific software hardware composition. Ubiquitous computing will lead to more general purpose devices and less highly integrated devices incorporating very specific functionalities. Thereby the direct relationship between functionality and the corresponding hardware used for its realization will disappear. While this will lead to positive effects in general, it will nevertheless create problems with respect to resource usage. Each eHome user can use a number of different services and different users can share the same environment within an eHome. The numerous services on the one hand and the limited number of hardware devices on the other hand will result in a disproportion. Therefore services usually will have to share the resources available in the respective environments. But in general not all resources may be used by multiple services at the same time. If e. g. a speaker system is used by lots of services at the same time, the user would only hear a chaotic noise. On the other hand, if one service exclusively uses the speaker system for a longer period of time, all other services depending on the speaker system will be blocked and can only proceed after the speaker system is released. This may not be reasonable in many cases.

To prevent such situations of resource conflicts a dynamic dependency management and resource allocation mechanism is needed to find a feasible trade-off between getting exclusive usage access to devices and sharing resources with other services. In this paper we will present our approach to tackle this issue and we will describe how we manage the service bindings at runtime with respect to concurrent use of resources. This approach is based on a fine-grained notion of bindings and corresponding usage relations, which enables interleaved resource access. Furthermore, our approach includes a priority management mechanism for service bindings, which is used to solve resource conflicts by prioritizing certain services over others. This is especially useful in case of security related services.

The paper is structured as follows. In Section 2 we describe previous work in our project that is used as basis for the dynamic dependency management introduced in this paper. The following Section 3 constitutes the main part of the paper in which we describe the extensions we made regarding binding types, priorities, and the application of both at system runtime. In Section 4 we explain the mechanisms used to implement the extensions. Furthermore we give an overview on some related work in Section 5. Finally, in Section 6, we conclude the paper with a summary and an outlook.

2 eHome Systems

Inhabitants of future smart home environments will use services from different domains as comfort, entertainment, communication, security, health care, or time and energy saving. To facilitate services in these areas a powerful infrastructure is needed to support development and later execution of services. Important

challenges we addressed in previous work are handling the dynamics that occur in smart environments and enabling adaptivity and interoperability of heterogeneous services. In the following we give a brief description of our project and the underlying system architecture.

2.1 System Architecture

In our prototype realization, we use the OSGi Service Platform as a component-based service architecture for the implementation of eHome services [1]. In OSGi software components are called bundles which are deployed onto a service platform. OSGi provides different capabilities needed to build and run a component-based software system. Most importantly OSGi offers a concept for modularization, which is only supported insufficiently in pure Java. Furthermore it offers life-cycle management, allowing to add or remove bundles at system runtime. A service registry allows to find and use registered services from other bundles.

While OSGi offers these important capabilities it provides only limited support for a dynamic and context-aware dependency management, which is needed for services in the area of smart environments. For eHomes we need a more sophisticated model to handle special requirements like which service instance is assigned to which room and how many devices of a certain type are available in a room and are ready to use. Furthermore the user requirements change often and the eHome system has to be reconfigured accordingly. These characteristic requirements affect the configuration process of such systems.

Our system is based on a three-layered architecture of eHome services. The upper layer consists of so-called top-level services. These are application services that provide their functionality directly to the user. To provide this functionality they typically rely on driver or integrating services. Driver services build up the bottom layer. They provide access and control of available hardware devices in the eHome. Integrating services are used in an optional intermediate layer which allows to provide different steps of abstraction to connect top-level services to driver services.

2.2 Dynamic Dependency Management

Any service which requires a certain functionality needs a corresponding service providing that functionality. This relationship between services constitutes a *dependency*. At runtime, when service instances are created, these dependencies have to be fulfilled to allow the execution of the service instances. Dependencies are fulfilled by creating a *binding* between service instances. We call this process *configuration*. The purpose of the configuration process is to create a service composition that matches the user requirements and the available device environment on the one hand and tries to meet all service dependencies if possible on the other hand. The service specification affects the dependencies to other services. For each dependency the composition behavior can be influenced by so-called *binding policies*. These policies define whether a dependency is to be fulfilled automatically or manually and whether to bind as many services as possible

or to bind only the minimum requirements. Furthermore, *binding constraints* impose restrictions on the service matching and thereby imply more specific service dependencies. Binding constraints are defined in the service specification for each required functionality. This way dependencies that relate to the current context of the environment can be realized.

We developed a prototype tool called *eHome Tool Suite* to support dynamic dependency management of eHome services as described above. Besides an editor for service specification the eHome Tool Suite also comprises a graphical editor for managing the runtime phase of the eHome system. Runtime management is implemented according to the so-called *SCD Process* [2] which consists of the three phases specification, configuration, and deployment. The specification can be modified by the user at any time using the graphical editor, e. g. moving services from one location to another or by manually adjusting service bindings. Based on this and the selected services' specifications the configuration of the system is generated or adjusted by the dependency management system described above. Finally the (modified) system configuration is deployed to the OSGi runtime environment in the deployment phase. This includes loading the corresponding bundles, creating service instances, and setting the service references according to the configured bindings. In contrast to other approaches aiming at a fully automatic system management, we provide the eHome Tool Suite as a means to visualize the current system state and to apply manual modifications to this state. In our view such means will be essential in future eHome systems to keep the users in control of their environments.

2.3 Example Scenario

In the rest of this section we will describe an example scenario to illustrate the problems evolving from the disproportion of used services and available resources.

Peter lives in an eHome and uses a web-enabled speaker system allowing to play different audio streams from network resources. The speakers are placed in the different rooms of Peter's apartment. His speaker system is used by several services, e. g. an alarm service for intrusion detection, a music service, a wake-up service, and a TV service. All these services depend on the speaker system.

Coming home from work, Peter's personal music service starts to play his favorite music. While walking through his apartment the music service is following automatically to his current location. After a while Peter wants to watch the news on TV. Because the music service has bound all speakers in the living room, he has to perform several manual reconfiguration steps such that the music service is stopped and the TV gets connected to the speakers. He also configures his wake-up service to wake him up at 6.00 am the next morning because he has an important meeting that day. After watching TV for a while Peter falls asleep in the living room. In the morning Peter wakes up at 6.30 am, fortunately not yet too late, wondering why he did not notice the wake-up call. The wake-up service could not use the speakers because they are still in use by the TV service and therefore could not be used by the wake-up service.

This problem also applies to other categories of resources. Considering a general heating service which keeps the temperature of a room at a certain level depending on the time of the year and the day we can easily imagine conflicts with personalized services relying on heating functionality. The wake-up service e. g. should be able to increase the bathroom temperature shortly before waking up the user. Therefore it needs to withdraw resources from the heating service.

These examples show that there will be a lot of standard services in future eHomes which make use of common resources. Even more services have to be taken into account if several users live in an eHome together. Especially an alarm service should be active at all times without requiring a constant manual reconfiguration of the system. It should always get access to resources that are currently used by other services due to the high security relevance.

3 Dependency Management

As we have seen in Section 2 the eHome prototype deals with common dynamic situations, like the dynamic reconfiguration of a user's music service to adapt to his movement. The reason for the occurrence of resource conflicts lies in the limited number of available resources. At some point, required resources will be unavailable which will lead to conflicts. But services do not need required resources during their entire runtime. If services would share resources efficiently during the time they are unused a lot of conflicts could be avoided. Based on this idea we present a solution to address this problem in the following.

3.1 Different Binding Types

The service developer enriches each service with a specification consisting of functional and non-functional meta-data, e. g. the service dependencies. This information is required for the (re-)configuration of an eHome system. Figure 1 illustrates the specification and configuration of the music service scenario and a wake-up service. On the left hand side the specification is shown. Both services require the functionality *Audio Output* provided by the speaker driver service. On the right hand side both services are installed in the living room. The music service is not usable and marked as invalid due to unavailable speakers. On the contrary the wake-up service is valid and can be used. Furthermore the wake-up service can use the speakers and the corresponding bindings respectively at any point in time until it gets undeployed. One resulting problem is the permanent locking of the speakers even though the wake-up service uses them only for a few minutes per day, e. g. in the morning when the user wants to be wakened. This problem does not only apply to the wake-up service. Most services do not use their assigned resources permanently. Nevertheless services cannot use resources bound by other services whether they are actually in use at the moment or not.

Since this is not a preferable solution, we extended our existing dynamic binding concept to allow a shared resource usage. Instead of only having bindings reserving resources permanently, we suggest an additional type of bindings which

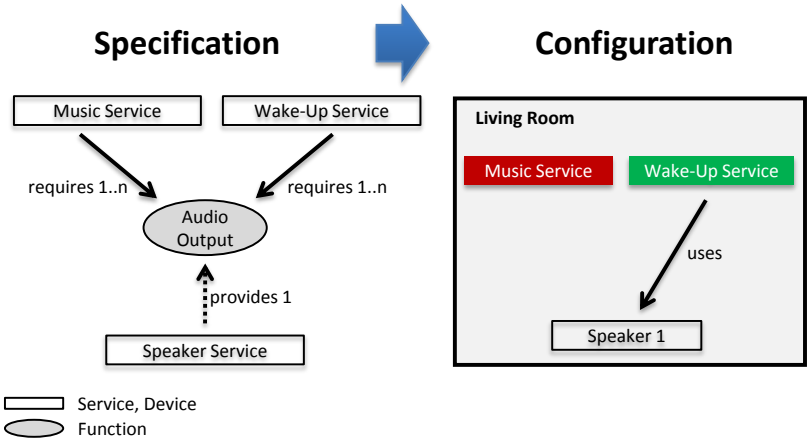

Fig. 1. Exemplary specification and configuration of eHome services

only locks a resource during the time it is actually using it. Bindings, allowing the shared use of resources are called *concurrent bindings* while bindings, only allowing the exclusive use of resources are called *exclusive bindings*. The extended dynamic binding concept is based on three main ideas: (1) Concurrent bindings get established independently of the actual availability of a resource. This is a substantial difference to exclusive bindings. As a consequence the availability of a service is computed differently. Unlike exclusive bindings concurrent bindings consume a provided functionality only while the binding is actually used. (2) If a service actually tries to use a concurrent binding the service framework must examine whether it is currently available or not. This binding check is performed by a so-called interceptor component. (3) Services trying to use a concurrently bound resource which is momentarily unavailable get notified about the failed use attempt. They are notified again when the resource becomes available again.

The three concepts mentioned above are implemented by the two newly introduced binding types. Exclusive bindings grant an independent and permanent access to the resource. But if no providing service is available no binding can be established. In that case the corresponding service is invalid if the binding is not specified as optional. In Figure 1 this applies to the music service which is invalid and cannot be used. Concurrent bindings always can be established but are not permanently usable, therefore each single attempt to use a concurrent binding is monitored and can either proceed or fail.

As mentioned above concurrently bound resources are locked temporarily depending on the actual use by other services. One important question is how to determine the time frame a concurrent binding is used. An idea coming first to mind is that the use of a binding begins with a call of a method and ends when the execution of the method is finished. But regarding the music service as an example, this would imply that after pressing the play button the bound speakers become instantly available to other services. This is because the music service is

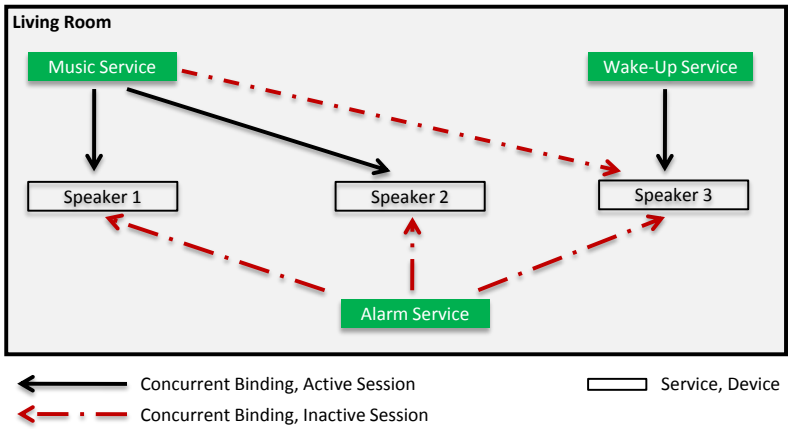

Fig. 2. Runtime configuration based on the extended dynamic binding concept

calling control operations on other services and resources. In case of the speaker system the music service would pass certain parameters like the URL of a music streaming resource. Most top-level services have a controlling character like the music service. This means the actual playback of the music does not involve the controlling service itself and the time period during which the binding is actually used by a method call is very small. In most cases the resource is required for a much longer period of time. In case of the music service pushing the button is only a point in time but playing the song is not.

To overcome these difficulties, the dynamic binding concept has been extended to include a session concept. An active session indicates that the concurrent binding is used and therefore the availability of the resource is limited for other services. On the contrary, an inactive session indicates that the concurrent binding is not used and therefore the resource is available to other services. In previous work we introduced a concept allowing to map service functionalities onto a domain-specific ontology to enable semantic matching and service adaptation [3]. This concept is further used for semantical tagging of methods and in this context we use it for *begin session* and *end session* annotations. This way the service developer can express that with pressing the play button a session begins and the speakers are used by the music service until the user presses the stop button and the session ends. If the user pauses the music or changes the track then the session status remains unchanged. This also means that a resource can only be used if the session is already active or some method is called which starts a session. If the user e. g. pushes the next track button while no music is played there is no active session and nothing will happen.

Figure 2 shows a runtime configuration of three eHome services based on the extended dynamic binding concept. For the time being all three services have bound speakers and are therefore valid. The wake-up service is bound concurrently to the third speaker and also uses this speaker at the moment.

All three speakers are bound to the music service but only speakers one and two are used by the service since the third speaker is already occupied by the wake-up service. The alarm service is also valid and has bound all speakers in the environment. At the moment the service does not need the resources, therefore the respective sessions are not active. However, with this concept problems can still arise. Some types of services, e. g. affecting the users' safety as the alarm service in our example scenario, are not executable under certain circumstances. Figure 2 shows a configuration where the alarm service could not access any speaker at all which is not acceptable in an intrusion situation when the alarm service needs to access the speakers to raise an alarm signal. Therefore we use a simple but effective priority concept for concurrent bindings.

3.2 Priority Concept

The priority concept is based on a ranking of the installed eHome services. If required this ranking can be used to determine which service is allowed to use a shared resource. In Figure 2 the alarm service should have the highest priority so that the service can use all the speakers if needed. It is not reasonable to assign a priority number to each eHome service at development time. eHomes are dynamic systems and at specification time it is not known which services are installed later on that have to share common resources at runtime. Therefore no ranking can be specified beforehand. Instead we have to regard the specification of a function and the respective providing and requiring services in the service setup of a specific eHome system. At runtime the user can create an individual ranking and assign to each providing functionality of a service a corresponding priority list consisting of the installed services requiring this specific functionality. This is shown at the top of Figure 3. The leftmost service has the highest priority and the rightmost service has the smallest one. For practical reasons the services can be arranged in priority groups. Services within the same group also have the same priority. This means it is not determined which service is of higher priority and thus no service can withdraw resources from other services of the same group. If a higher priority service accesses a resource in use by a lower priority service, the resource is withdrawn from the lower priority service and reassigned to the higher priority service. The lower priority service then receives a notification and the corresponding session is closed. This procedure is based on the assumption that resources are fully interruptable and hence the withdrawal of concurrently bound resources is always possible. Anyway, the concurrent bindings remain established while resources are reassigned. When a required resource becomes available again a waiting service is notified and can continue using the resource.

4 Implementation

This section briefly describes the implementation of the extended dynamic binding concept. As mentioned before an interceptor component is used to monitor concurrent bindings at runtime. This component consists of two parts: The first

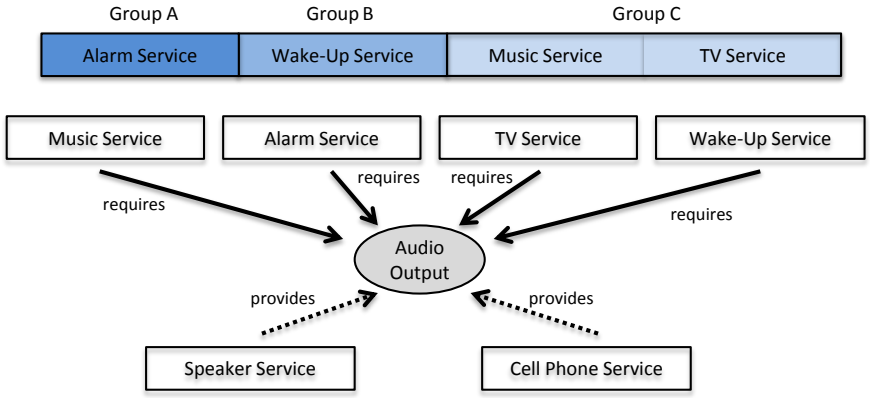

Fig. 3. Connection between services, functions, and priorities

part is responsible for detecting the actual use of a binding. The second part deals with access control, session handling, and the priority management.

Resources get accessed through method calls, e. g. `playMusic()` for the music service. After such a call is detected the eHome framework must decide whether the method call should be allowed or rejected. As explained above the use of a dynamic binding depends on the resource's actual usability. This check is related to the second part of the interceptor. According to the interceptor's result the method call can proceed or is rejected. In the second case, the service needs to be notified about the invalid access to be able to react correctly. For that purpose we use Java's exception mechanism. Similar to other frameworks like e. g. iPOJO [4] the eHome services mostly implement pure application logic. In general the service developer does not need to know much about the eHome framework to be able to develop services. In this case the developer must only be aware of the fact that some binding may not be usable and an exception is thrown. But this is a general requirement anyway in ubiquitous computing and especially eHome systems. Due to the dynamics in such environments, required resources may not be available or accessible at all times.

To monitor concurrent bindings we use aspect-oriented programming [5]. In our case the aspect-oriented language AspectJ is used, which is a seamless extension of the Java programming language. AspectJ enables a clean modularization of cross-cutting concerns such as error checking, logging, monitoring etc. We use especially the load-time weaving mechanism of AspectJ to implement the interception mechanism. This feature enables code injection into bundles that are already loaded and running.

In line 1 of Listing 1.1 the pointcut *functionMethods* is defined, which is responsible for the method call detection. Each call of a method within the *ehome.interfaces* package will be intercepted. These calls correspond to using a service binding. Pointcuts only match specific points in the control flow of a program, which are called join points. To actually inject code or implement

```
1  pointcut functionMethods(EhService usedSrv) : target(usedSrv)
       && call(public * ehome.interfaces.*.*(..));
2
3  Object around(EhService usedSrv) throws
       BindingUnusableException : functionMethods(usedSrv) {
4    ...
5    String rString = interceptor.intercept(usingSrv, usedSrv,
         methodSig, interfaceName);
6
7    if (...) { // if binding usage is allowed
8      ...
9      proceed(usedSrv);
10   } else { // binding is not usable, raise an exception
11     int hashCode = Integer.valueOf((rString.split(";")[1]));
12     throw new BindingUnusableException(usingSrv.toString()+",
           "+usedSrv.toString(), hashCode);
13   }
14 }
```

Listing 1.1. Aspect intercepting service communication

cross-cutting concerns an advice is used. If a join point is reached corresponding advices are executed. AspectJ supports different kinds of advices. Line 3 shows an around advice which interrupts execution at respective join points and executes the advice instead of the original method call. The given parameter *usedSrv* is the service which requests to use the binding. In line 5 the interceptor component checks if the service *usingSrv* is allowed to use the binding. If access is granted, line 9 redirects to the original method call. Otherwise the around advice throws an exception in line 12 to notify the calling service about the invalid access.

The second part of the interceptor component is called from within the advice discussed above and is modeled as a Fujaba story diagram. Fujaba is a UML-based development tool which allows to generate Java code from UML diagrams [6]. Besides the data model so called story diagrams, which are a combination of UML activity and collaboration diagrams, are used to model the application logic. This way executable Java code can be directly generated from the model. Since the whole story diagram modeling the interceptor component is quite complex, we will discuss a simplified view depicted in Figure 4. The interception process consists of four important steps. First the binding in question must be determined. The interception process only continues if the binding is a concurrent one, otherwise it returns with *proceed* which means the original method call will be executed by the around advice. The next steps depend on the state of the session and the binding's current usability. If the session is already established (cf. diamond *Active Session?*) then the method call is valid and can be allowed to execute. But before that, the interceptor checks if the session is to be closed. Like explained in Section 3 it is possible to semantically annotate methods with an

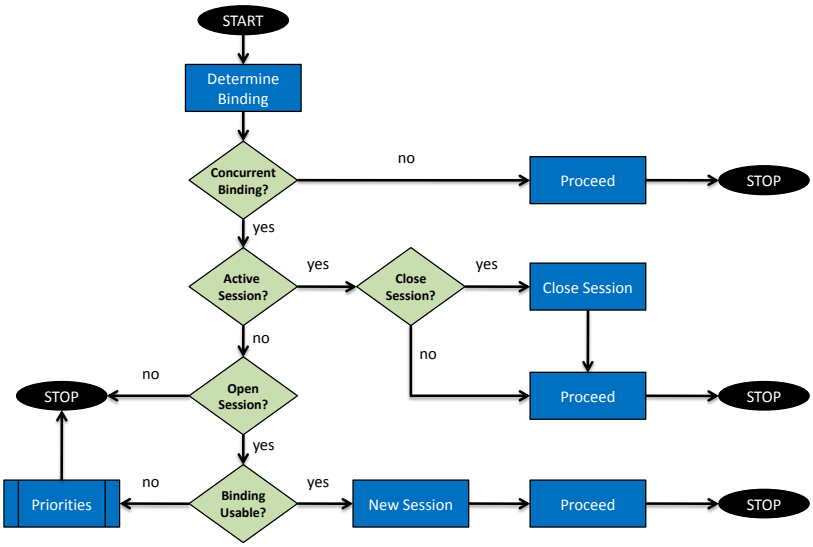

Fig. 4. Evaluation of intercepted service communication

end session tag. This annotation is read by the interceptor. If a service requests to use a resource and the session is not active then the interceptor must check if a session is to be opened (cf. diamond *Open Session?*). If this is not the case the around advice rejects the method call. Otherwise a usability check is performed (cf. diamond *Binding Usable?*). The interceptor counts the active sessions and exclusive bindings to determine if a binding can be used. If no more active sessions are allowed, service priorities are taken into account to check if some active session has to be withdrawn from a service with lower priority. If the binding is still not usable the method call will be rejected. Otherwise a new session is opened and the method call is executed.

The described mechanism has been implemented and integrated into our prototype system. So far, we only performed a qualitative validation based on the *eHomeSimulator*, a software tool we developed to simulate different smart environments [7]. The results show that our approach actually allows to resolve resource conflicts. However, a quantitative evaluation is still pending.

5 Related Work

In service-oriented architectures, applications are composed from several services which often appear or disappear dynamically. Dependency management is therefore a key aspect and a lot of research is going on in this field.

A number of approaches are based on the OSGi Service Platform which also provides dependency management. Dependencies between bundles and dependencies between services are handled differently in OSGi. The first ones are package

dependencies and are related to the OSGi module layer. The bundle developer specifies these dependencies in the bundle manifest. The OSGi framework resolves these bundle dependencies and only if all constraints are satisfied the bundle can be loaded. Service dependencies on the other hand are related to the service layer. An OSGi service is a normal Java object registered at the service registry under one or more Java interfaces. In general, a service can use the service registry to search for required services registered by other bundles. But this type of dependency management does not support automatic resolution.

Since release 4 of the OSGi Service Platform, the Declarative Services specification is available which evolved from the Service Binder project [8]. It separates two important responsibilities: Implementing the application logic on the one hand and dependency management on the other hand. This allows service developers to focus on the application logic while dependency management is outsourced into a special framework bundle. Similar to our approach bundles get enriched with meta-data descriptors of their provided and required functionalities. In contrast to Declarative Services, our dependency management allows to bind services depending on context information, e. g. bind only services within the living room. Further on, OSGi does not provide a priority concept, which is needed in the domain of eHomes as we have shown.

In [4] Escoffier et al. propose a service-oriented component model to simplify OSGi application development. Like with Declarative Services the application logic is separated from the non-functional requirements. In iPOJO each service is encapsulated inside a container, which is used to inject non-functional requirements to manage e. g. service bindings or the service lifecycle. Each container is composed by handlers managing these non-functional requirements. If a required service becomes available, the appropriate handler directly injects the needed objects. If a required service disappears and cannot be replaced, the depending services become invalid. Since iPOJO employs a decentralized composition approach, each container has its own dependency manager. Thus no global view is available and features like our resource management based on priority groups cannot be supported.

In [9] Bottaro et al. discuss several requirements for software architectures in home environments. The requirement of *service continuity* addresses rebinding problems due to stateful services. If a resource has an internal state and is replaced by another resource at runtime, the state information has to be transferred to the new resource. The authors propose to handle state transfer on the application level since an automatic approach is only possible in specific applications. This is also the case in our approach. However, the problem of how to handle the rebinding of services in the first place while allowing shared resource usage is not addressed in [9].

In [10] the problem of disconnections regarding component bindings in distributed environments is addressed. The authors argue that top-level components are rarely usable if disconnections occur frequently. The proposed solution is to activate and deactivate required interfaces according to their current availability and to allow component execution even if some required interfaces are deactivated.

In that case the top-level component may still be usable though with restricted functionality. In contrast to our approach, the component developer is required to take care of testing the interface status before invoking methods. Our mechanism does not require such tests. However, the developer has to handle events in case of failed use attempts in our approach.

In [11] the authors describe an approach called COMITY for runtime conflict detection in pervasive computing environments based on the PCOM component model. The presented approach analyzes the effects that pervasive applications take on the environments they are executed in and how this affects other applications and users. The proposed system consists of a conflict manager component connected to a database that stores a context model representing the current state of the environment. A second database stores conflict specifications which determine what kind of situations are considered to be conflicting. Based on this, the conflict manager detects conflicting situations at runtime. In contrast to our approach COMITY does not focus on conflicts regarding resource usage but rather on conflicts resulting from different user interests. We do not address these conflicts as we believe they will require manual resolution by the users in most real-life scenarios.

In previous work at our department an approach to rule-based conflict detection has been developed [12]. The presented conflict detection mechanism assumes that each resource in the system is specified in form of an ω-automaton describing its behavior. Together with a set of rules which formalize the different conflict types a monitor component can detect conflicting situations at runtime. This approach is similar to COMITY but it is based on a different infrastructure. It requires services to provide a semantic specification describing their behavior and it forces top-level services to be realized in a rule-based approach. We do not impose such strong requirements on service development, instead service developers can focus on the core task of implementing application logic.

6 Summary and Outlook

In this paper we described our approach to support dependency management for eHome systems dealing specifically with the resource constraints arising in ubiquitous computing scenarios. The general idea of our solution is to monitor and manage service communication based on the current state of the system configuration. All this is realized with minimal impact on the service implementation, so that the service development process is not affected unnecessarily. We introduced a session concept based on tagging service methods. Together with the interception of service communication this information is used to manage the utilization of bindings at runtime. In addition to that we allow to define priority groups at runtime to allow an automatic resolution of conflicts.

There are several issues which are to be addressed in future work. Up to now, we tested our approach using a testbed fully implemented in software. We still need to perform a quantitative evaluation in a larger scenario to analyze the scalability of our implementation. Furthermore, it is still an open question

how to simplify the definition of priority groups to support automatic conflict resolution. In real-world systems we also need a simple but useful mechanism for solving conflicts manually at runtime without bothering the users with permanent requests for interaction.

References

1. The OSGi Alliance: OSGi Service Platform Core Specification. `http://www.osgi.org/Specifications/HomePage\#Release4` (April 2007) Release 4.1.
2. Retkowitz, D., Stegelmann, M.: Dynamic Adaptability for Smart Environments. In Meier, R., Terzis, S., eds.: Distributed Applications and Interoperable Systems, 8^{th} IFIP WG 6.1 International Conference (DAIS 2008). Volume 5053 of LNCS., Springer (2008) 154–167
3. Retkowitz, D., Pienkos, M.: Ontology-based Configuration of Adaptive Smart Homes. In Taïani, F., Cerqueira, R., eds.: Proceedings of the 7^{th} Workshop on Reflective and Adaptive Middleware (ARM'08) held at the 9^{th} International Middleware Conference, ACM (2008) 11–16
4. Escoffier, C., Hall, R.S., Lalanda, P.: iPOJO: an Extensible Service-Oriented Component Framework. IEEE International Conference on Services Computing (SCC 2007) (July 2007) 474–481
5. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-Oriented Programming. In: European Conference on Object-Oriented Programming (ECOOP). Volume 1241 of LNCS., Springer (June 1997)
6. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels, G., Rozenberg, G., eds.: Proceedings of the 6^{th} International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, Germany. LNCS, Springer (November 1998) 296–309
7. Armac, I., Retkowitz, D.: Simulation of Smart Environments. In: Proceedings of the IEEE International Conference on Pervasive Services 2007 (ICPS'07), IEEE (2007) 257–266
8. Cervantes, H., Hall, R.S.: Automating Service Dependency Management in a Service-Oriented Component Model. In Crnkovic, I., Schmidt, H., Stafford, J., Wallnau, K., eds.: Proceedings of the 6^{th} ICSE Workshop on Component-Based Software Engineering (CBSE6). (May 2003) 379–382
9. Bottaro, A., Gérodolle, A., Lalanda, P.: Pervasive Service Composition in the Home Network. In: 21^{st} International Conference on Advanced Information Networking and Applications (AINA'07). (May 2007) 596–603
10. Hoareau, D., Mahéo, Y.: Constraint-Based Deployment of Distributed Components in a Dynamic Network. In Grass, W., Sick, B., Waldschmidt, K., eds.: Architecture of Computing Systems - ARCS 2006. Volume 3894 of LNCS., Springer (2006) 450–464
11. Tuttlies, V., Schiele, G., Becker, C.: COMITY – Conflict Avoidance in Pervasive Computing Environments. In Meersman, R., Tari, Z., Herrero, P., eds.: Proceedings of the 2^{nd} International Workshop on Pervasive Systems (PerSys '07). Volume 4806 of LNCS., Springer (2007) 763–772
12. Armac, I., Kirchhof, M., Manolescu, L.: Modeling and Analysis of Functionality in eHome Systems: Dynamic Rule-based Conflict Detection. In: Proceedings of the 13^{th} Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06), Washington, DC, USA, IEEE (2006) 219–228