Dynamic Adaptability for Smart Environments

Daniel Retkowitz and Mark Stegelmann

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

Abstract. Software reuse and hardware integration are key factors to offer flexible, low-cost smart environments. Until now, we have been using a static process called the SCD-process to allow a tool-supported realization of such smart environments. The SCD-process is comprised of three different phases: specification, configuration, and deployment. As an initially specified environment is expected to change during runtime and the user may wish to influence certain aspects of the configuration, the static process had to be adapted. This paper describes a new process that supports continuous specification activities and allows for an automated adaptation of the smart home's configuration based on a model-driven approach. We enriched the specification of services with binding policies and constraints to allow for a flexible reconfiguration and a service-specific adaptation. The new configuration mechanism facilitates dynamic reconfiguration based on context information and the extended service specification. In addition, we present a visual tool, which is used to assist the developer and the end-user.

1 Introduction

Ambient intelligence, ubiquitous and pervasive computing, are some of the more recent topics in computer science. Approaches in these fields aim at creating so called *smart environments* by separating computing from today's desktop PCs to make applications and their functionalities available anywhere, independent of PC hardware [1]. This way, users can access services wherever they are. Any available device should be usable to realize service functionality. Devices installed in the user's current environment, together with mobile or wearable devices may be used by the software.

Related to home environments, ambient intelligence is realized by so called *eHomes* or *smart homes*. We refer to *eHome systems* as a combination of devices and software running in such an eHome. This software is running on a residential gateway that controls all home appliances. Top-level services are applications that offer certain functionalities to the user. Top-level services are based on integrating services that reduce the level of abstraction down to device driver services, which control actual hardware devices. In today's homes, a lot of appliances are available, but in general, these appliances are not interconnected.

To facilitate comprehensive services based on multiple appliances, that offer complex functionalities, it is necessary to develop flexible and adaptive software. To achieve this goal at low overall costs, the eHome software has to be built from standard components, that are automatically composed according to the user's needs and the individual home environment.

The goal of our research is to enable such low-cost eHome systems by composing eHome services from reusable software components. In our prototypes these software components are developed according to the OSGi component model [2]. We currently employ the Equinox framework of the Eclipse platform [3] as an OSGi runtime environment for our service bundles, as the components in OSGi are called. The customization of the eHome software is achieved later on by composition of the services in a process of specification, configuration, and deployment. This process is called the *SCD-process* [4].

In [4], a tool called *eHomeConfigurator* is presented, which enables the SCD-process by employing a model-driven approach based on Fujaba. Fujaba is a tool for specifying a software's data model and application logic using different UML diagrams [5]. Furthermore, it allows to generate Java source code from such a specification. In this paper, we present a redesigned SCD-process, which is capable of handling the dynamics in smart environments and incorporates several major improvements. We also present a newly developed tool based on the Ecliplse platform mentioned above.

The rest of the paper is structured as follows. In Section 2 we describe a scenario to illustrate the need for a support of dynamic changes in smart environments. Next, in Section 3, we explain the different concepts which form the basis of the new continuous SCD-process. After that, in Section 4, we discuss the realization details and we show how we implemented the new concepts. Then we present the current state of our new tool in Section 5. In the following Section 6, a short overview of related work is given. Finally, in Section 7, we give a conclusion and point out some open problems and future work.

2 Scenario

Before discussing the details of our system architecture, we will take a closer look at an example scenario. We will see that changes appear frequently in smart environments, which imposes certain requirements on the SCD-process.

John is coming home from work. At home, he sits down in his living room. Since he has a music service selected in his personal profile, his favorite music starts playing when he enters the room. After some time he walks into the kitchen to prepare some food. The music in the living room stops playing when John leaves the room. Once he enters the kitchen, his new location is detected by the eHome and the music service starts playing John's music again in his new location using the speakers integrated in the kitchen wall. The music service resumes playing from the last position where John was listening in the living room. A few minutes later John's wife Mary comes home, too. When she walks through the living room, her personal video conferencing service notifies her about an incoming call from Anne. She takes the call and a live picture of Anne appears on the TV. She can hear Anne's voice from the speakers in the living room. Anne talks about their last joint vacation and wants to show Mary some of the pictures she took. Mary's video conferencing service is capable of presenting different media data. Using her PDA, Mary selects the TV to display Anne's picture presentation. The pictures appear on the TV and the live picture of Anne is reduced and displayed in the lower corner of the screen. Mary wants John, who is still in the kitchen, to see the pictures too. So she adds the display in the kitchen to be used for media data output of her video conferencing service. Now the picture presentation is also displayed in the kitchen. After Mary has seen several pictures she decides to create prints of some of them. So she picks up her PDA again and connects the printer in the living room to her video conferencing service. The printer is also capable of processing visual media data and starts printing the selected pictures.

This example scenario shows some standard situations we assume in future smart environments. To design a software system that is sufficiently flexible and adaptable to support such scenarios, we had to come up with a novel development process. In the next section we will describe our approach based on a modified SCD-process in more detail.

3 System Architecture

The SCD-process as described in [4] aims at reducing development costs per eHome by increasing the amount of possible service reuse. Thus services are developed once and enriched with a specification. This allows for a later automatic integration of the services into different eHome configurations. The service specification describes which functionality each service provides to other services and which functionality is required to do so.

3.1 Service Layers

We distinguish between three types of services: driver, integrating, and top-level services. Driver services represent low-level driver software needed to access the different hardware devices. Top-level services are applications that offer functionality to the user. So called integrating services may be used to add multiple layers of abstraction to the basic, driver-based hardware access. In many cases the functionality that is required by a top-level service does not directly match a functionality provided by a driver service, because both services are on very different layers of abstraction. In such cases adequate integrating services have to be found to adapt both layers to each other.

In Figure 1 the three types of services are shown by the example of the video conferencing scenario from Section 2. In Figure 1(a) the Video Conferencing top-level service is depicted, which requires several functionalities as Audio Input, Audio Output, etc. to operate properly. For each required functionality the cardinality shows how many instances of this functionality are required at least

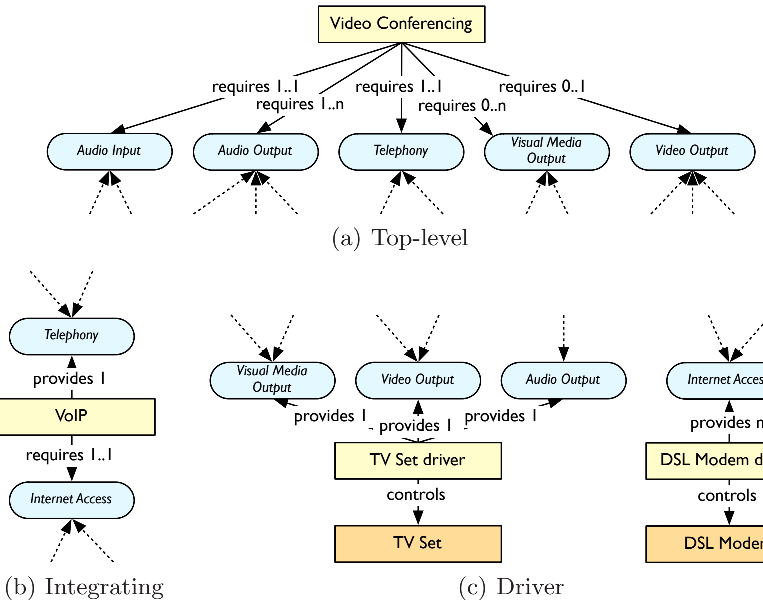


Fig. 1. Service Layers

and at most. Figure 1(b) shows an integrating service. In this example the depicted VoIP service is used for voice over IP telephony. Accordingly it provides Telephony functionality and furthermore requires Internet Access, since network access is required. Finally, in Figure 1(c) two driver services are depicted. A DSL Modem Driver which provides Internet Access and a TV Set Driver which offers Visual Media Output, Video Output, and Audio Output.

Any service used in the eHome system is specified as indicated above. This way, services can be composed by the system later on in the configuration phase according to their specified functionalities. In the described example the Video Conferencing service may use the VoIP service to fulfill its Telephony requirement. The VoIP service in turn may use the DSL Modem Driver to fulfill its Internet Access requirement.

3.2 Process Requirements

An automatic support for the SCD-process is one of the main requirements for the application of eHome services. The user will not accept a system that requires permanent interaction. Most of the tasks have to be performed automatically. Ambient intelligence implies that the environment acts and reacts automatically according to context changes. Nevertheless, the user permanently wants to be in control of the situation, i. e. the system must not act in a way the user does not expect. In case the system makes a decision the user does not agree with, it should be possible to manually apply changes so that the user has means to influence the system's behavior. Especially when considering home environments, these are key issues for the acceptance of eHome systems.

3.3 Continuous SCD-Process

As we have seen in Section 2, the nature of dynamics at runtime of an eHome system can be quite diverse. To cope with the described dynamics, we redesigned the SCD-process described in [4] to meet the new requirements discussed above. Our approach focuses on considering runtime changes concerning user movement through different locations and device mobility. Whenever changes occur in the eHome environment, the different phases of the SCD-process have to be re-executed to adapt the software to the new situation. Any change of the user's location or desires or any change of available devices implies corresponding changes in the specification and hence also the configuration and the deployment. We refer to the new adapted process as *continuous SCD-process*.

To facilitate an automated adaption of the eHome system to context changes, the availability of certain sensor devices that allow to detect these changes is required. The scenario presented above requires e. g. some means of automated person and device detection. The demonstration environments described in [4] and [6], which we use as testbeds, provide these capabilities e. g. by means of video cameras or remote controls for the different users to log in. We assume that appropriate technologies will be available for future eHomes at low prices.

In Figure 2, the new overall eHome process is illustrated. On the left-hand side of the figure the service-specific part is depicted. This part consists of the service development phase and the phase of service specification. These phases are performed by a software developer. The resulting service components can be used in any eHome based on our framework.

The right-hand side of Figure 2 depicts the eHome-specific part of the overall process, i. e. the SCD-process which represents the runtime phase of the system. For each eHome the SCD-process demands a specification of floor plans,

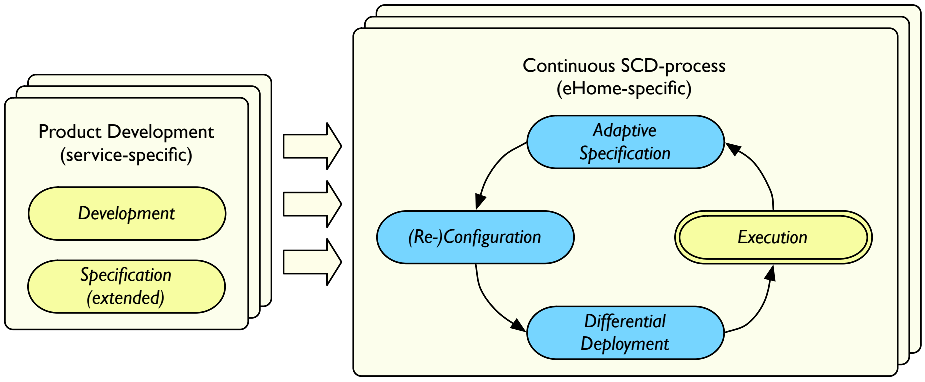


Fig. 2. Overall process incorporating the new continuous SCD-process

the desired top-level services, and the available devices. This is the specification phase of the SCD-process. The division into two independent specification phases, the product-specific service specification and the eHome-specific specification, allows to minimize the expert knowledge required by the end-user. After the eHome-specific specification phase the configuration phase follows next. In this phase any changes of the specification are processed and a recomposition of services according to their provided and required functionalities is performed. Depending on the service specification the composition is performed automatically or manually. In case a service requires manual binding, so far the user has to create the appropriate bindings in the graphical service representation of the visual specification tool described in Section 5 in order to start the service. In the future, this way of interaction could be extended e. g. by offering a graphical user interface for PDAs such that users can carry PDAs as remote controls for their current environments. Furthermore composition constraints are checked to keep the configuration valid. In Section 4, further details on the configuration algorithm are presented. Finally, when the eHome software has been configured, the configuration has to be deployed. In the deployment phase service instances are created or destroyed according to the configuration and the bindings between these instances are registered.

4 Realization

As described in [7] we pursue a model-driven approach to realize eHome systems. Our data model, which is partially shown in Figure 3, is specified as UML class diagram. The runtime behavior of our framework is described using so called UML story diagrams to express the configuration logic. All defined classes and methods are translated to compilable Java source code via Fujaba. Thus no actual configuration code has to be written by hand.

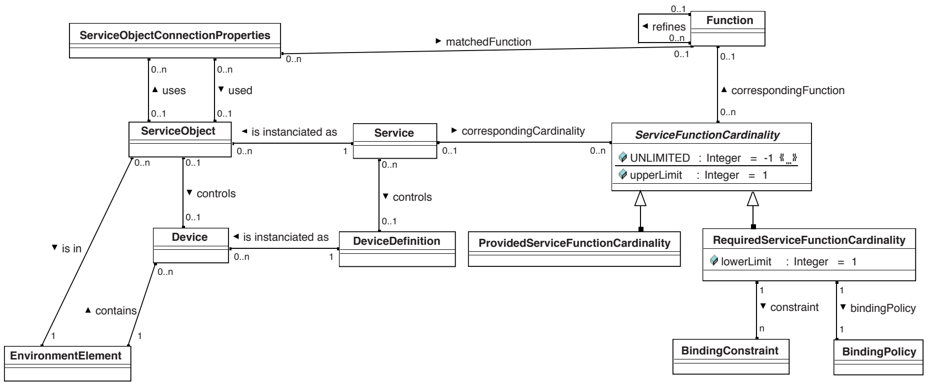


Fig. 3. Part of the data model UML class diagram

4.1 Data Model

In the data model, `Services` are used to represent the eHome-independent service specifications. These services are linked through `ServiceFunctionCardinality` objects to `Functions`. The `RequiredServiceFunctionCardinality` objects express that the service requires some functionality. `ProvidedServiceFunctionCardinality` objects respectively mean that the service provides some functionality. Furthermore, lower and upper cardinality bounds are stored, whereas lower bounds are only used for required functionalities. A driver service additionally `controls` a specific `DeviceDefinition`.

During the eHome-specific configuration phase the specified required and provided functionalities are used to match services. This way the needed service runtime instances called `ServiceObjects` can be determined. To connect two of these `ServiceObjects`, `ServiceObjectConnectionProperties` are used. Top-level `ServiceObjects`, selected by the user, are contained in `EnvironmentElements` thereby describing the `ServiceObject`'s location. *Context-aware* [8] or so-called *personal services*, adapt to the user's context, e. g. his surroundings. Such `ServiceObjects` thus are always associated to their user's current location.

4.2 Dynamic Service Composition

The continuous nature of the new process requires some way to determine which match a connection between two `ServiceObjects` is based on. This information is stored in the model by the `matchedFunction` relation from `ServiceObjectConnectionProperties` to `Function`. Thus it is feasible to extend and respectively reduce prior compositions of `ServiceObjects` according to the service's specification.

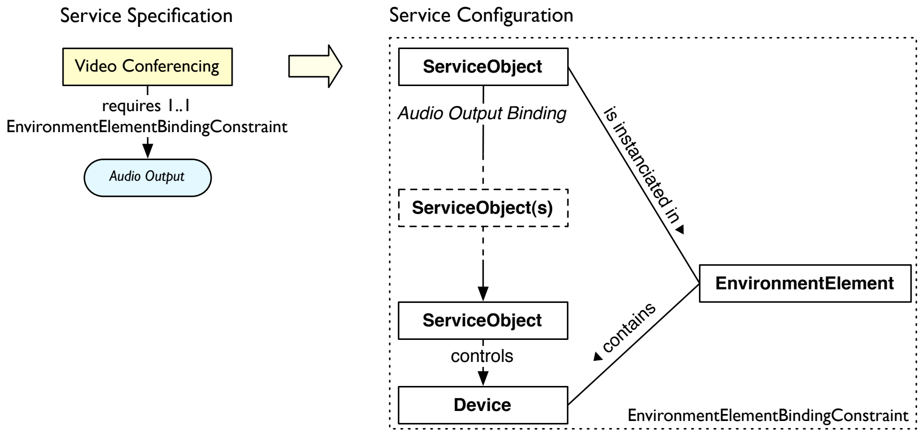


Fig. 4. Abstract visualization of a binding constraint

Binding Policies. To specify for each required functionality if it should be bound automatically or manually by the user, so-called binding policies were introduced.

A *binding policy* constitutes a strategy concerning the establishment of bindings for a specific `Function` required by a `Service`. We offer three types of policies covering different configuration strategies. The *automatic binding policy* manages all bindings to services providing the required function automatically. The *manual binding policy*, in contrast, only allows manual binding modifications. To automatically establish bindings until the lower cardinality limit is reached and allow manual interaction beyond that point, an *automatic mandatories policy* can be selected.

Binding Constraints. To implement a flexible concept of fine-grained, context-aware services, binding constraints were introduced. We call a *binding constraint* a declarative description of a graph pattern that has to be matched in the configuration in order to establish the binding. If the configuration graph conforms to the pattern the binding constraint *is satisfied*.

Realizing a constraint for personal services is straightforward. The data model contains all information necessary for such a pattern. An abstract visualization of the constraint is shown in Figure 4. The binding constraint is satisfied if all `Device`s that are used via the Audio Output binding are from the user's current location. As constraints are specified for each required `Function` separately the developer may choose to omit this restriction for some functionality that does not have to be chosen from the user's current location. Telephony support e. g. typically does not have to be located in the same room as the user, even for personal services. This kind of supporting services are usually not bound to a specific location.

Binding constraints are a fundamental concept within our approach. They can be used to impose various effects on the dynamic composition mechanism. We will extend this concept in the future to support further context-aware features.

4.3 Adaptive Configuration

To incorporate environment specification changes as discussed in Section 3 the possibility to choose from different binding policies was introduced. As shown in Figure 3, each `BindingPolicy` object is related to a certain `RequiredService-FunctionCardinality`. These objects in turn are bound to a `Function`. Thus this information can be easily derived by the binding policies' implementations.

In Figure 5 an excerpt of the method `addBinding(ServiceObject)` is shown. This method is responsible for the automatic creation of bindings between `Ser-viceObject`s and is used by the automatic binding policy and by the automatic mandatories policy. The particular method fragment depicted creates bindings to existing service compositions that may provide the specified `function`. In the topmost activity, being a so-called *for-each activity*, candidate `ServiceObjects` meeting this requirement are determined. For this purpose, first those `Services` that according to their specification may offer `function` are ascertained. Then

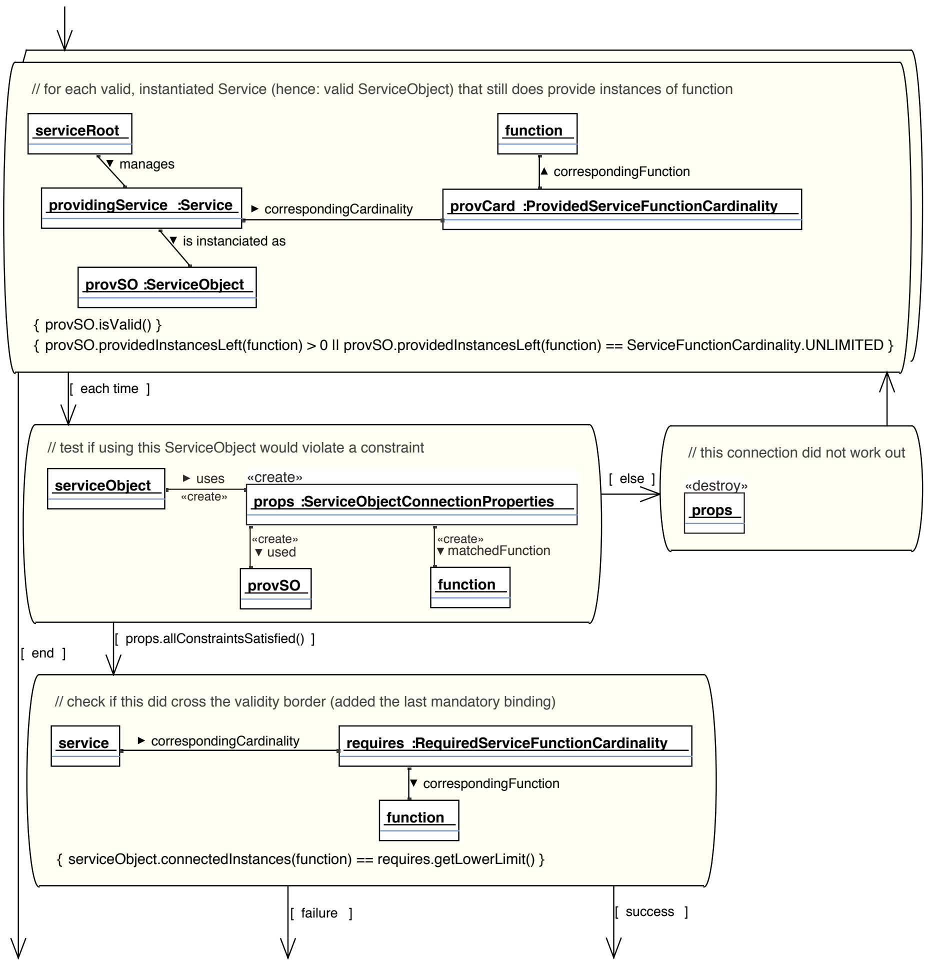

// for each valid, instantiated Service (hence: valid ServiceObject) that still does provide instances of function
serviceRoot
▼ manages
function
▲ correspondingFunction
providingService :Service
▶ correspondingCardinality
provCard :ProvidedServiceFunctionCardinality
▼ is instanciated as
provSO :ServiceObject
{ provSO.isValid() }
{ provSO.providedInstancesLeft(function) > 0 || provSO.providedInstancesLeft(function) == ServiceFunctionCardinality.UNLIMITED }
[each time]
// test if using this ServiceObject would violate a constraint
serviceObject
▶ uses
«create»
«create»
props :ServiceObjectConnectionProperties
«create»
▼ used
«create»
▼ matchedFunction
provSO
function
[else]
// this connection did not work out
«destroy»
props
[props.allConstraintsSatisfied()]
[end]
// check if this did cross the validity border (added the last mandatory binding)
service
▶ correspondingCardinality
requires :RequiredServiceFunctionCardinality
▼ correspondingFunction
function
{ serviceObject.connectedInstances(function) == requires.getLowerLimit() }
[failure]
[success]

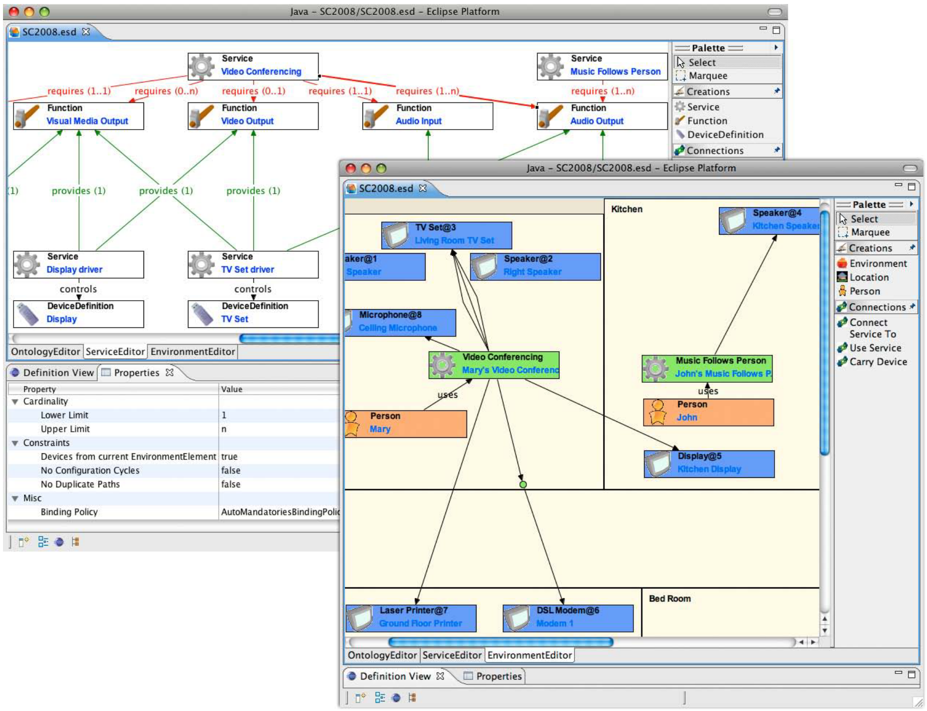
Fig. 5. Activity diagram fragment of `addBinding(ServiceObject)`

each `provSO` object that is an object of such a service is examined if it is valid and is able to provide an instance of the required `function`. As some functionality required by a `ServiceObject` may be unavailable at times, a `ServiceObject` does not always have to be valid. We define a `ServiceObject` to be *valid* if all mandatory required functionality is provided by `ServiceObject`s that are valid themselves and all binding constraints for connections between the `ServiceObject`s are satisfied. To check the latter, the binding has to be established, as `BindingConstraints` are defined for existing configuration graphs. Thus for each of the found `provSO`s a `ServiceObjectConnectionProperties` object is created in the left activity below. This object indicates that `provSO` is used to provide `function`. If at least one binding constraint is violated the binding is destroyed in the activity to the right and the next `provSO` is determined in the topmost activity. Execution continues with the bottommost activity if all constraints are satisfied. If the connection did add the last mandatory binding the `ServiceObject`'s validity may have changed. The validity is thus reevaluated. Else the method ends as a binding was added by the policy. If none of the existing `ServiceObjects` qualifies as provider for the required `Function` the left arrow marked with *end* is followed. The method continues by creating new `ServiceObjects` of `Services` that may provide the function.

5 Tool Support

To provide tool support for the continuous SCD-process we created a visual specification tool (cf. Figure 6). We chose the Eclipse Graphical Editing Framework (GEF) for implementing the user interface. GEF [9] is a framework focusing on providing an easy way to build Eclipse-based graphical editors for existing models. As Fujaba is able to generate code for our model and the configuration logic, GEF was a natural choice for realizing the tool.

The *Service Editor* shown in the left screenshot of the figure is used by the service developer. Here services, functions, device definitions, and their relations can be modeled as described in Section 4. In the right screenshot the



Property	Value
Cardinality	
Lower Limit	1
Upper Limit	n
Constraints	
Devices from current EnvironmentElement	true
No Configuration Cycles	false
No Duplicate Paths	false
Misc	
Binding Policy	AutoMandatoriesBindingPolic

Fig. 6. Screenshots of the visual specification tool

Environment Editor is depicted. It is used for the specification phase of the continuous SCD-process. Environments and locations can be created to specify floor plans. Afterwards, the hardware devices may be placed and desired top-level services can be selected and associated to the prior defined locations. The tool palette of the Service Editor allows users to specify what service objects should be used as personal services and what devices they currently carry. To manually link service objects to each other or to some devices the user may also use the tool palette. After each change in the editor the configuration code is executed to reconfigure the runtime graph accordingly. The amount of required user interaction can be reduced if person and device detection and localization is available. Corresponding context changes can then be automatically detected. For the future we plan to integrate user profiles, such that desired services can be automatically inferred from the profile data.

The *Service Editor* depicts the service specifications for the scenario detailed in Section 2. Each service is specified along with its required and provided functions. In addition, driver services are linked to their respective devices. Binding policies and binding constraints can be chosen for each required functionality. This is shown in the properties view visible at the bottom of the left screenshot.

The right screenshot visualizes the configuration at the end of the example scenario. Mary resides in the living room. John is in the kitchen. The composition is a result of the successive environment changes and the user-generated specification changes described.

6 Related Work

As mentioned before there is a lot of research activity going on in the field of ambient intelligence and the related topics. Some of this research focuses on smart environments. Nevertheless there are numerous other areas of application. Related to software engineering, the concepts of software components and services are addressed frequently.

In [10], Cervantes and Hall discuss the concepts of service and component orientation and a service-oriented component model which is used in their project, called *Gravity*. The authors' initial goal, which is detailed in [11], is to provide an automatic service dependency management framework for user-oriented applications. As these applications are composed of services that continuously arrive or depart during runtime, the applications constantly have to be reassembled. Gravity allows to eliminate dependency management code needed to deal with such compositional issues using a tool called *Service Binder*. The Service Binder prototype is realized for the OSGi framework. Using XML descriptors similar to our service specifications each OSGi component is enriched with meta-data descriptions of its required and provided services. Cardinalities and static or dynamic policies can be specified to define the runtime reconfiguration behavior. The continuous SCD-process, we described in this paper, also aims at dynamic recomposition. Yet, the composition behavior defined by binding policies and binding constraints differs from the policies of the Gravity project. In contrast to our approach, no means of user interaction are provided by Gravity. This is an important aspect of our approach as the user has to be in control of the system even if the process should be executed as automated as possible. For each component instance the Service Binder creates instance managers that locally try to maintain the instance's validity according to the respective instance descriptor. In contrast, our graph-based approach relies on a centralized composition mechanism that may leverage local as well as global context information. This includes locations of persons, services, and devices. Context-restrictions based on the model may be described as binding constraints.

In [12], Botarro and Gérodolle describe several extensions to the original Service Binder addressing some of its limitations, like service selection ambiguity for equivalent service providers, support for context-awareness, and remote distribution of services. The *Extended Service Binder* introduces service provider rankings based on dynamic properties and transparent service access to remote services to the original concept. At the moment, we are exploring approaches that may help reducing selection ambiguity based on semantic service description.

In [13,14], Broens et al. propose a middleware called Context-Aware Component Infrastructure (CACI) that allows transparent binding management for personalized mobile component-based applications. The authors distinguish between context producing entities (e. g. GPS receivers, RFID beacons) and context consuming entities (e. g. context-aware applications). Bindings between these are called context bindings, and they are established dynamically and maintained at runtime. Using the CACI Component Description Language (CCDL), developers may define which context bindings are required by their application. Every context binding is specified with several parameters including a binding policy. The policy may be set to be either dynamic, semi-dynamic, or static. A dynamic policy indicates that the binding is to be updated if *better* context producers become available at runtime. Semi-static context bindings are only replaced if the context producer gets unavailable. Static bindings are only bound once. Compared to our binding specification approach CCDL lacks the possibility to specify if a context producing entity should be bound automatically or if user interaction is desired. In fact, no manual modifications can be applied by the user. No tool is provided for visualization and interaction.

In [15], the authors present an approach to behavioral service composition that is based on semantic web services. The user's needs are specified as so called abstract user tasks. Abstract user tasks do not refer to actual component instances. To realize these tasks, a matching algorithm is applied to compose semantic web services, which implement a certain behavior. Both abstract user tasks and semantic web services are specified as OWL-S processes. These process definitions are modeled as finite state automata, which are used by the matching algorithm to reconstruct the abstract user tasks based on available service behavior. In our approach, we do not consider service behavior to perform the matching. Instead we focus on context information, user interaction, and especially the reconfigurability of service compositions. In the approach discussed above, no reconfiguration issues are addressed.

7 Conclusion and Outlook

Smart homes require flexible and adaptive software composed from standard components. To offer smart home software to end-users at a reasonable price, the individual eHome-specific software is composed from these components at runtime. Service composition is a complex task, which has to be solved by a service gateway capable of managing, running, and adapting compositions automatically. If the end-user is bothered with technical configuration tasks in everyday life, smart homes will not be accepted by the general public. However, users want to be in control of their environments. Therefore, complementary means of user interaction have to be offered.

In this paper we described a dynamic process for composing standard service components, such that the resulting compositions meet the individual requirements for specific eHomes. The continuous SCD-process is capable of handling the dynamics occurring during runtime of eHome systems. To automate service composition, a detailed service specification has to be provided. We achieve this by specifying binding policies and binding constraints. This allows the service developer to define an adequate composition behavior for each specific service. We described a dynamic reconfiguration mechanism and how we implemented the according algorithm in a model-driven approach. Finally, we gave a short overview of a new tool supporting the eHome development process.

Currently, we are adapting the final deployment phase of the SCD-process to connect our tool to the eHomeSimulator [6], which is a virtual eHome environment we use as a testbed. This enables us to simulate environments containing numerous different devices and to evaluate more complex scenarios and the dynamic behavior at runtime. So far, preliminary tests indicate that the adaptive recomposition at runtime does not produce any significant performance overhead in comparison to the prior static approach used in [4]. In the near future, we will carry out a more extensive performance analysis to evaluate complex scenarios with a larger number of simultaneous users. We currently also work on further extensions of the service specification. Especially the specification of service functionalities using semantic labels is to be extended, to better support the matching algorithm and to allow for a more flexible service composition in heterogeneous environments. Other future extensions could be automated support for conflict resolution, optimization of the global configuration with respect to resource usage or other parameters, and also support for service versioning and updating at runtime.

References

1. Weiser, M.: The Computer for the 21^{st} Century. Scientific American 265(3), 66–75 (1991)
2. The OSGi Alliance: OSGi Service Platform Core Specification. Release 4 (August 2005), `http://www.osgi.org/osgi_technology/download_specs.asp#Release4`
3. des Rivières, J., Wiegand, J.: Eclipse: A platform for integrating development tools. IBM Systems Journal 43(2), 371–383 (2004)

4. Norbisrath, U., Mosler, C.: Functionality Configuration for eHome Systems. In: Proceedings of the 16th International Conference on Computer Science and Software Engineering, CASCON 2006, ACM Digital Library (2006)
5. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In: Ehrig, H., Engels, G., Kreowski, H.-J., Rozenberg, G. (eds.) TAGT 1998. LNCS, vol. 1764, pp. 296–309. Springer, Heidelberg (2000)
6. Armac, I., Retkowitz, D.: Simulation of Smart Environments. In: Proceedings of the IEEE International Conference on Pervasive Services 2007 (ICPS 2007), pp. 257–266. IEEE Press, Los Alamitos (2007)
7. Norbisrath, U., Armac, I., Retkowitz, D., Salumaa, P.: Modeling eHome systems. In: MPAC 2006: Proceedings of the 4th International Workshop on Middleware for Pervasive and Ad-Hoc Computing (MPAC 2006), 6 pages. ACM Press, New York (2006)
8. Abowd, G.D., Dey, A.K., Brown, P.J., Davies, N., Smith, M., Steggles, P.: Towards a Better Understanding of Context and Context-Awareness. In: Gellersen, H.-W. (ed.) HUC 1999. LNCS, vol. 1707, pp. 304–307. Springer, Heidelberg (1999)
9. Moore, B., Dean, D., Gerber, A., Wagenknecht, G., Vanderheyden, P.: Eclipse Development using the Graphical Editing Framework and the Eclipse Modeling Framework, 1st edn. IBM (Redbooks) (February 2004)
10. 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 6th ICSE Workshop on Component-Based Software Engineering (CBSE6), pp. 379–382 (May 2003)
11. Hall, R.S., Cervantes, H.: Gravity: supporting dynamically available services in client-side applications. In: ESEC/FSE-11: Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 379–382. ACM Press, New York (2003)
12. Bottaro, A., Gérodolle, A.: Extended Service Binder: Dynamic Service Availability Management in Ambient Intelligence. In: FRCSS 2006: Future Research Challenges for Software and Service (April 2006)
13. Broens, T.H.F., van Halteren, A.T., van Sinderen, M.J.: Infrastructural Support for Dynamic Context Bindings. In: Havinga, P., Lijding, M., Meratnia, N., Wegdam, M. (eds.) EuroSSC 2006. LNCS, vol. 4272, pp. 82–97. Springer, Heidelberg (2006)
14. Broens, T.H.F., Quartel, D.A.C., van Sinderen, M.J.: Towards a Context Binding Transparency. In: Pras, A., van Sinderen, M. (eds.) EUNICE 2007. LNCS, vol. 4606, pp. 9–16. Springer, Heidelberg (2007)
15. Mokhtar, S.B., Georgantas, N., Issarny, V.: Ad Hoc Composition of User Tasks in Pervasive Computing Environments. In: Gschwind, T., Aßmann, U., Nierstrasz, O. (eds.) SC 2005. LNCS, vol. 3628, pp. 31–46. Springer, Heidelberg (2005)