Towards Automatic Deployment in eHome Systems: Description Language and Tool Support

Michael Kirchhof, Ulrich Norbisrath, and Christof Skrzypczyk

Department of Computer Science III, Aachen University of Technology,
Ahornstr. 55, 52074 Aachen, Germany,
{kirchhof|uno|christof}@i3.informatik.rwth-aachen.de

Abstract. eHome systems are essentially component-based systems. One of the main reasons preventing a wide application of eHome systems in practice is the effort needed to interconnect all appliances, necessary controller and infrastructure components to benefit from derived value-added services. In the area of software engineering, this problem is addressed by configuration management and software deployment. In this paper, we introduce a language, which forms a basis for describing coarse-grained and abstract scenario defaults up to complete deployment information to carry out the actual installation in an eHome. We present a tool supporting the automatic transformation of an abstract input document into a complete deployment document adapted to a specific eHome environment. The tool is based on the description language.

1 Introduction

Home automation promises new comfort and useful services in everyday life. These services will become manifest in ubiquitous appliances. From users' point of view, services should be at least as easy in use. From developers' point of view, different *appliances and technologies* exist, which have to be integrated.

Connecting home area networks with communication and data networks provides potential for many service ideas. So far, remote control of services is most popular. Furthermore, users may access their eHome using different communication and data networks. Appliances can be controlled from any place (e.g., the office) using a browser and the Internet. The owner of an eHome may determine and change state of the alarm equipment with the Wireless Application Protocol (WAP [1]) and mobile communication networks using a mobile phone. In case of an alarm, the security equipment sends a multimedia message (e.g., MMS [2] or eMail) to the owner of an eHome, who will receive the message with the mobile phone. But eHome Services may also interact on their behalf with other data and communication network services making value-added services possible.

Decoupling of services from the underlying infrastructure is based on abstraction from the infrastructure itself. An eHome Service should rely on the types of devices, instead of specific devices and their proprietary implementation. For example, an alarm system should be able to integrate any motion detector or device, which is able to detect motion (e.g., cameras), instead of being tightly bound to a vendor-specific motion detector. Taking the back-end systems into account, an eHome Service is not only a

piece of software executed on the service gateway. An *eHome Service* is the *wholeness of services a user experiences*.

Home automation aims at mass markets. Therefore, a technical background can not be expected from the users. Mainly users will expect trivial *plug-and-play* from home automation appliances as they do from their legacy pendants. This leads to the demand for *self-configuring* and *maintenance-free* systems. Home automation can not require technicians come to the user's home to integrate any kind of devices to home networks. Hence, there is a need for automatic configuration management and software deployment. Current approaches deal with manual configuration management and software deployment [3,4]. The solution has to ease the realization, the configuration, and the deployment of distributed eHome systems, which do not impose any further burdens to users.

To ease and automate this *configuration and deployment process*, we will introduce a language, which forms a basis for describing coarse-grained and abstract scenario defaults up to complete deployment information to carry out the actual installation in an eHome. Furthermore, we will introduce an ontology and tools to support the automatic transformation of a very abstract input document into the complete deployment document adapted to a specific eHome environment.

This paper is structured as follows: In the following section, the scenario of our eHome system is described. In section 3, we will discuss related languages and frameworks capable of describing and processing various aspects of component-based systems in the face of automation. In section 4, we will introduce our proposed description language and in section 5, the tool support is described. Finally, we give a summary and a conclusion of this paper.

2 Scenario

The scenario 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 communication infrastructures via different protocols (e.g., X.10, EHS, Lon, Jini) and acts as a runtime environment for the *service gateway*. The service gateway manages and runs certain software components. 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, or the heating systems. 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.

eHome systems are built on top of integrable net-aware devices in households. Applicable devices, communication techniques, and infrastructures vary in several dimen-

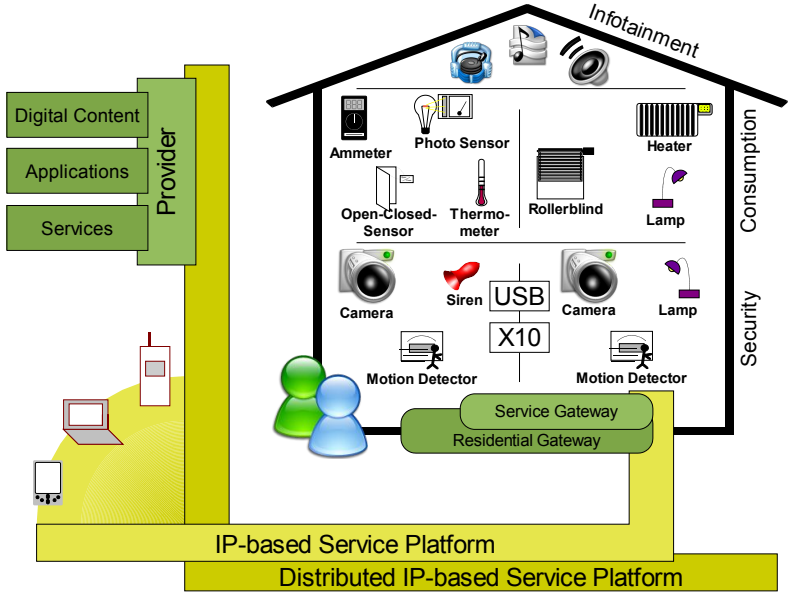

Fig. 1. Scenario

sions: Devices vary with respect to interfaces, features, locations, and range. Important differences in communication are protocols, bridges, name spaces, and through-put. The applied infrastructure is centralized, because current development in gateway technology provides sufficient computational power for eHomes and allows to reduce the complexity of system design in contrast to decentralized or mixed approaches. Distribution aspects of gateway technology and service execution are to be observed in a later phase. Last but not least, the integration of external service providers is an unanswered question. Until now, just hardware-specific problem fields have been dealt with. Hence, suitable and reasonable models and structures for this new application domain still need to be developed.

In contrast to traditional distributed systems, one very important fact in eHome Systems is that the expected potential of market penetration is extremely high (in terms of million households), but the variety in terms of the underlying ontology is relatively small. To solve the configuration and deployment problem in eHome systems, we introduce configuration documents on three levels. We describe transformations on these documents, as well as tools supporting the different levels. Further improvement with respect to the automatic handling of such documents should be possible by the incorporation of knowledge-based systems. Resulting from the expected high penetration and the extremely large amount of expected variants, a manual adaption of systems for each customer is neither possible nor affordable. Thus, there must be a possibility to automate this adaption process. On the one hand the configuration has to be automated and on the other hand the deployment has to be automated. To ease the configuration and

deployment task, we rely on the layered approach to OSGi-based gateways described in [5]. To abstract from communication details in terms of provider-ehome-connectivity, we make use of the Distributed Services Framework (DSF) [6].

3 Related Work

As we want to define a language capable of describing both configuration and deployment information, we studied other frameworks offering such descriptions or forming a base for automatable deployment of components. The most important ones and their benefits are described below.

xADL 2.0 [7] is an XML-based description language for software architectures. Inheriting XML's schema-based mechanisms, it is flexible and highly extensible. The default schemes for xADL 2.0 based on xArch [8] provide definitions for elements typically found in architecture description languages (ADL). The creation, management, and manipulation of xADL 2.0 documents and schemes is well supported by tools from the xADL 2.0 community. However, there are no tools supporting automatic configuration or deployment processes.

The *Openwings* [9,10] consortium was founded as an open community. Its objective is to specify a framework for component-based systems independent of database, architecture, and operating system. Openwings defines its own component model and component programming model. It has a strong focus on the application in loosely coupled distributed networks. In Openwings, every component has to be started separately or via batch support. There is no mechanism defined to automate the instantiation process of a complete component-based system.

RIO [11] is an extension of the Jini-framework [12]. It adds a component model and a component programming model to the Jini-framework. In RIO, components are called Jini Service Beans (JSB). They are described in an XML-based notation called *Operational String*. Operational Strings may include further Operational Strings and Service Bean Elements. Service Bean Elements include all information necessary to install a JSB. Hence, it is sufficient to specify Operational Strings to instantiate a complete component-based system by one operation.

The *Open Services Gateway Initiative* (OSGi) [13,14] specifies a set of software application interfaces (APIs) for building open-service gateways on top of *residential gateways* [5]. The residential gateway describes a universal appliance that interfaces with internal home networks and external communication and data networks. Similar to a data network gateway, a residential gateway is equipped with interfaces to different physical media and provides conversion between protocols. It represents a concept, which lets different networks communicate transparently. This leads to a wide support of various standardized technologies. In the OSGi context components are called *bundles*. Each bundle may provide its own configuration[1] user interface (UI) utilizing the (specified) HTTP-service. Additionally, for configuration data get- and set-methods

[1] The meaning of configuration in OSGi in this context differs from our understanding of configuration. In OSGi configuration is the setting of properties. In our context a configuration is a document, which also includes the collection and combination of components. For a more precise definition see [15].

are provided in order to be directly accessible from other bundles and components. The configuration data is isolated within the bundle. A third way for the realization of configuration tasks is implementing the interface ManagedService. Bundles can be remotely configured by the module Configuration Admin, which is specified by OSGi, too. Configuration data are encapsulated in Property objects. The Configuration Admin acts as a mediator. With that, basic configuration in terms of adjustments of properties can be realized. OSGi is widely used in white goods and has the potential to be established as a standard framework for household appliances.

The OSGi gateway (see figure 2) resides within a Java runtime environment, which offers the well-known features of Java [16]. The core component is specified as the *OSGi service framework*, which acts as a container for service implementations. This environment includes a Java runtime with life cycle management, persistent data storage, version management and a service registry. Services are Java objects implementing a concisely defined interface. Services are packaged within *bundles*, which register zero, one or more services within the framework's service registry. Bundles contain services implemented in the Java programming language, a manifest file describing import and export aspects, and additional implementation-specific libraries. Bundles can be deployed and undeployed during runtime, while the information in the manifest file ensures the integrity of the system. Security aspects are handled as well. As the figure shows, a bundle is not restricted to rely on functionality offered by the framework, it can profit from every layer below, i.e. native functionality offered by the operating system and the hardware. This stands in contrast to layered approaches in software engineering, but allows the realization of bundles for any protocol. To ensure a minimal common set of functionality, certain bundles are standardized (e.g., the Log bundle for logging and the HTTP bundle for user interface purposes).

Central to the OSGi specification [17] is the *service framework* with a service registry. Components register their services at the service registry where other applications may retrieve and use them. Based on the concept of residential gateways [18,19], the open services gateway specification describes an approach that permits coexistence of and integration with multiple network and device access technologies. In addition, components may be added implementing new technologies as these emerge. Interaction is enabled via Java interfaces, without relying on proxy objects. While other systems -like Jini- are decentralized, the OSGi approach is a centralized system, which simplifies the maintenance of the system to the disadvantage of distribution aspects.

To make eHome systems affordable for the masses far-reaching automation is needed. A requirement for tool support to achieve the automatisms is the existence of an extensible language describing abstract scenario requirements up to complete deployment configurations. Hence, it is highly desirable to combine the extensibility and basic properties of xADL 2.0 with the possibility of automatic deployment of RIO's operational strings. Furthermore, the language should allow the usage of frameworks like Openwings with a strong focus on distribution. To support OSGi as an established eHome framework, the necessary information for the deployment on this platform have to be integrated and a tool to support automatic deployment has to be developed.

Whereas the specification languages mentioned above (like xADL 2.0 or RIO's Operational Strings) are intended to describe software components and their dependencies,

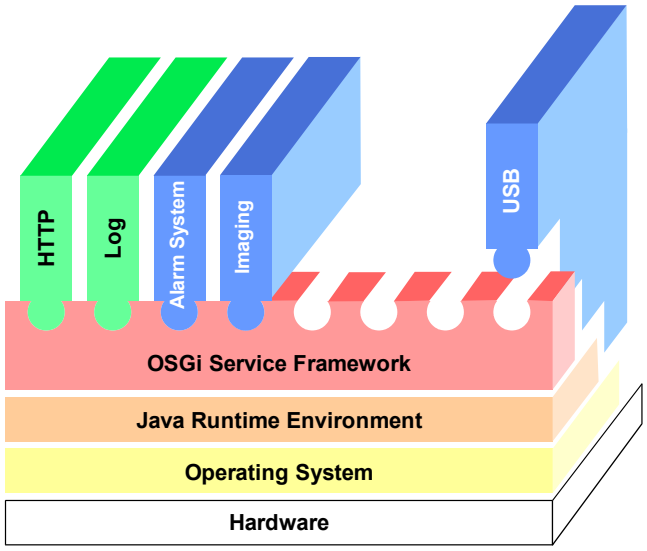

Fig. 2. OSGi System Structure

our goal is to cover also the physical architecture of eHome systems. This allows us to extend the simplification of the configuration process beyond software issues and discloses automatic generation of complete system solutions including all involved products.

4 Description Language

Our approach is process-oriented. So, we examine the life cycle of a newly ordered service. For the relations between configuration document levels, the supporting tools, and the knowledge base compare figure 3. The customer (i.e., the eHome owner) triggers this process, by requesting the provider portal to display available services. After authentication and authorization, the suitable services are identified. Of course there is a large amount of general knowledge (like platform specific drivers for appliances, interface descriptions, controller components, and dependencies), which has to be acquired and specified by the provider or its subprovider in beforehand to allow the determination of these services. We distinguish three different levels of a configuration document: If the customer decides to subscribe to one of the offered services, this choice, which we will call *scenario configuration document* (1), is transferred to the provider. Then, this document is enriched with necessary information to do the actual software installation at the customer's home. The tool supporting this step is called *Deployment Producer* and the resulting document is called the *deployment configuration document* (2). After installation of the necessary hardware, a tool called *Runtime Instancer* instantiates the deployment configuration and brings the functionality specified in the scenario configuration document to life. While the service is maintained, the Runtime Instancer is

used to save the current configuration, deployment data, and status information of the eHome in an *runtime configuration document* (3). After a phase of usage, maintenance, or installing further services the service is stopped and deactivated via the Runtime Instancer. We emphasize that this process should also hold for upgrading a service. Upgrading is either extending a service with new devices or adding services, which could depend on already installed services. Hence, we demand that the scenario configuration document is able to include a previous deployment configuration.

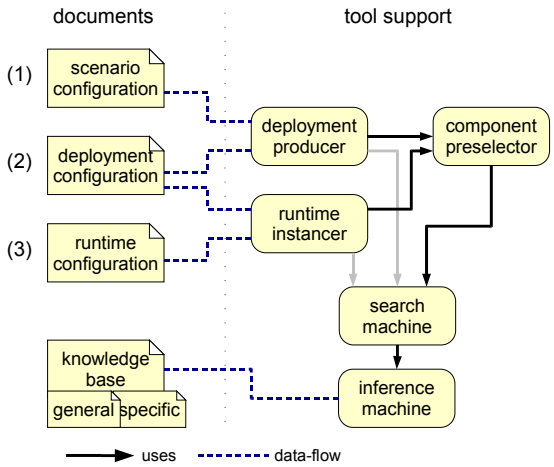

Fig. 3. Configuration Document Levels, Tools, and Knowledge Base

According to the life cycle described above, there exist three levels of a configuration document. Our language describes all three corresponding configuration types starting with very simple and abstract scenario configurations up to complete runtime configurations including the current states of the already installed devices. Apart from software components, the language also describes all devices and platforms used in the system. The term *platform* stands for a combination of a framework and the device it is executed on. Software components, devices and platforms are called *units*. An important part of the model is the knowledge base, which classifies all known unit types. Figure 4 shows such a simplified device classification. The type attributes in the hierarchy are passed down to further type specifications. A description of a unit should always name its type to specify its functional context and include the corresponding attributes.

The language also describes the dependencies between units. In our context, these are called *connections*. Together, units and connections form a configuration graph where units are represented by nodes and connections by edges. There are three main types of connections:

logical connections A *logical connection* indicates where a software component is being executed on and is always directed towards a framework.

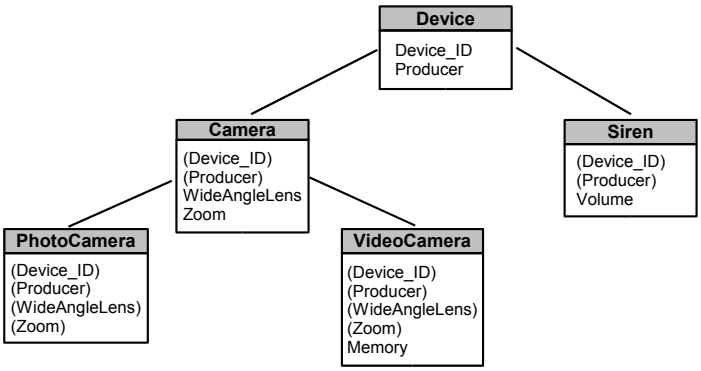

Fig. 4. Classification of Different Device Types

physical connections *Physical connections* give information on the kind of the communication medium and the protocols used by the devices.

usability connections All other dependencies are described by the *usability connections*.

The connection types are also classified in the ontology and there exist many further specializations for each of the three main types. For example special types for USB, Infrared and Bluetooth connections are possible. There are also special types of usability connections, for instance to indicate driver and control components of devices. In addition, the ontology can be extended if any new types with new attributes are required.

Finally, a service itself must be described. On the one hand, all units and connections have to be assigned to the service they are used by, so the service also needs a name and an identifier. On the other hand, more specific attributes of the service are required. Often it is important to know what priorities the service has while accessing shared resources. A good example are loudspeakers, which can be used by surveillance, infotainment and communication services at the same time. Other service attributes can describe functional behavior of a service. However, sometimes it seems more sensible to use special formalisms to describe such kinds of information and place them in an extra file. In this case, a corresponding link to the external data and some meta information should be provided within the service description. In our implemented examples we use rule-based descriptions for the service behaviors, which are managed by a rule engine [20,21] after the installation. But also in the descriptions of units and connections, the use of special formalisms seems sensible, for example to specify the geographic locations of the devices. In this context, it is also important to observe that it is not possible to achieve a complete description of really all aspects of the eHome system.

The distinction between the different types of units, connections, and services shows that it is not possible to define a complete language for eHome configurations. For this reason, we define only a core set of standard attributes required for the description of every configuration graph. Such information include for example unique identification

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<eHome xmlns="x-schema:eHomeSchema.xml">
  <Units> ... </Units>
  <Connections> ... </Connections>
  <Services> ... </Services>
</eHome>
```

Listing 1.1. Structure of a Configuration Description

numbers, names and types of each element in the configuration. For the type specific information the required attributes must be added to the language definition. The names of the new attributes and their range should always be chosen according to their descriptions in the classification. This convention guarantees that by adding new types of products to the knowledge base the description language can be easily extended without causing problems in the tool implementations.

```
<Unit  id="56"  type="motion_detector_control"
    supertype="component" status="ready">
  <Attributes>
      <Producer value="Philips"/>
      <Platform>
        <OS value="DebianLinux3.0"/>
        <Framework value="rio"/>
        <MinPlatformCPU value="2"/>
        <MinPlatformMemory value="16"/>
      </Platform>
  </Attributes>
  <Resources>
      <DriverFile value="sensor_driver"/>
      <Component>
        <InterfaceFile value="sensor65-dl.jar"/>
        <ImplFile value="sensor65.jar"/>
        <DataFile value="action_sensor.xml"/>
      </Component>
  </Resources>
  <Instances> ... </Instances>
  <States>... </States>
</Unit>
```

Listing 1.2. Component Description

For the realization of the language we define an XML document class [22]. It includes all core elements needed for the description of the graph structure and also some standard attributes of the most usual unit, connection and service types. Listing 1.1 shows the skeleton of a typical XML document describing an eHome system. Depend-

ing on the product types used in the house different XML elements must be added to the schema definition (or DTD) of the language. It is the task of the Runtime Instancer to extract the required information and use them during the deployment and installation process. Listing 1.2 shows the specific description of a component for the RIO framework [11]. If describing components for other frameworks, the attributes can vary. Correspondingly, listing 1.3 presents the structure of a connection description and listing 1.4 shows an example of a simple service description.

```
<Connection  id="6" type="physical_USB" supertype="physical"
       source_unit_id="58"  target_unit_id="62"
       scenario="simple_security" status="ready">
  <Attributes>
      <Bidirectional value="false"/>
  </Attributes>
  <Resources>...</Resources> ...
  <States>...</States>
</Connection>
```

Listing 1.3. Connection Description

```
<Scenario  id="100" type="simple_security"
      supertype="security"  status="ready">
  <Attributes>
      <Description value="a very simple security scenario"/>
      <PriorityAudio value="5"/>
  </Attributes>
  <Resources>...</Resources> ...
  <States>...</States>
</Scenario>
```

Listing 1.4. Service Description

5 Tool Support

The configuration process can be simplified by using several tools. The main goal of the Deployment Producer is the automatic creation of deployable deployment configurations containing all relevant information about the eHome services. Especially, the data required for the installation of the software components is important. The user should provide an abstract description of the wanted future eHome system, specifying its devices and services only as far as desired. The tool should then be able to generate descriptions of possible deployment configurations, to estimate their values and to

present the best solutions to the user. The descriptions should be based on the XML language introduced above. In order to provide the semantic information required for the configuration process an appropriate knowledge base has to be provided. We propose an algorithm for the automatic search of runnable configurations. A simplified version of this algorithm is described in this section. We also present its first implementation.

It is easy to show that the creation of deployable configurations is a NP-hard problem and that excessive search is needed to find all existing solutions. For all units in the original configuration, the implemented algorithm examines recursively all combinations of possible products. Of course, not all arrangements of devices and components are possible. The products cannot be regarded independently from the service context and from their dependencies to the other products. On each level of the recursive search one more unit will be specified. The Deployment Producer tests all possible product choices for this unit and for each choice creates a new configuration containing the product's specification. These new configurations are then passed in succession to the next recursion level, where the new added information can be taken into account while processing the other still not processed units. Hence, regarding a single search path of the tree, for the units from the lower recursion levels the number of the possible product choices is narrowing by each new added product specification. By using depth-first-search, the algorithm traverses such a tree of different configurations and tries to find a solution with all units successfully specified (see figure 5).

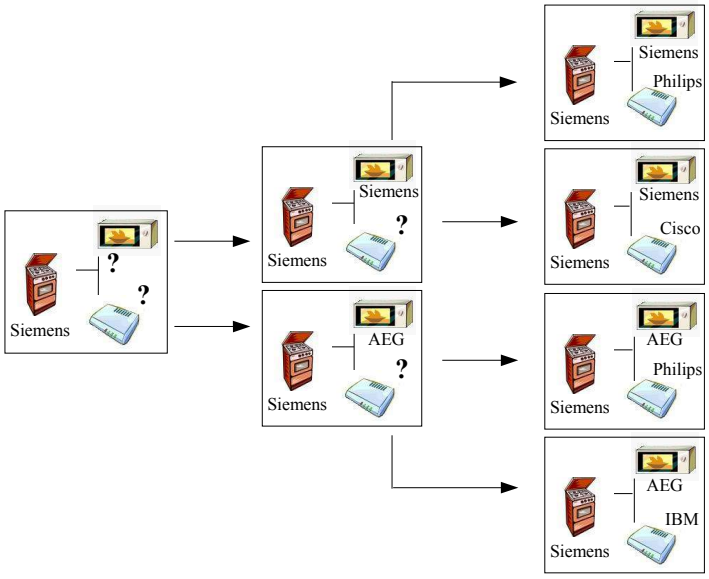

Fig. 5. Building the Search Tree (Recursive Search)

To decide, which devices and components can work together, the tool has to analyze the possible product combinations in the context of the given services. The required data

is stored in the knowledge base. It contains both a hierarchy of all unit, connection and service types with their attributes and states, and the semantic dependencies between them and their requirements. By using this knowledge base, the program can process each pair of connected units and depending on their already fixed attributes determine their other properties. Every already fixed unit attributes narrows the number of product choices for the other connected units. So in order to allow all possible configurations on each recursion level only the necessary attributes should be set.

To access the knowledge base, the Deployment Producer uses the tool *Component Preselector*. The development of the interface between these two programs is one of the main challenges of the project. In order to use the semantic information of the knowledge base we define the communication procedures between them. For this reason we define several *request types* for accessing the knowledge base. As described above in the first request type the Deployment Producer passes the attributes of the two connected units in order to receive more complete specifications of both. In certain situations, depending on the attribute values of the units no connection is possible at all. Such situations are called *conflicts*. If a connection can be found , the configuration on the current recursion level will be filled with the new attribute values and the search can proceed, either with the next connection of the current unit or the first connection of the following unit on the next recursion level. Two further request types of the interface will be mentioned later in this text.

It is important to remark that the original input configuration can be rather abstract and it will usually never consist of all devices and components needed for the realization of the final eHome system. In case of an upgrade there will be at least one deployable part in the configuration in addition to the abstract description of new devices and services. That is why the Component Preselector not only confirms and specifies a possible interaction of two units standing in a certain relation, but also returns a small subconfiguration containing all the other units needed for their connection. For example a house security control component, which should be connected with a motion detector device will always need at least a driver component for the device and a platform where the component can be executed. Hence, the answer of the Component Preselector will also contain these two new units and the connections between them. The Deployment Producer adds this subconfiguration to the current eHome configuration replacing the two old units and their connection. After all units and connections from the original input file were processed, the found configuration finally contains all information required for the deployment of the system. All relevant attributes of the units are fixed and all missing parts of the configuration are added. That is why this first phase of the search is called *completion phase*.

A found configuration already contains the information required for the deployment. All element attributes are specified and all missing units added. However, now the configuration can contain several units of the same type, which are also equivalent in their functions. The second phase of the search is called *merging phase*. Now the program must try to find all equivalent units in the configuration graph and merge their nodes if all dependencies with the other involved units allow the transformation. This is not only necessary to generate sensible and cheap configurations but also to guarantee the actual realization of the system. The reason is that during the completion phase all

missing units are added to the configuration without regarding any physical limits of the existing resources, like the amount of the available connection slots or the disc memory capacities of the platforms. The aim is to reach the smallest possible amount of units, but at the same time guarantee that all limits are respected. To find out if two units of the same type can be merged the program has once again to consult the knowledge base of the Component Preselector. Here the tool uses the second request type. During this procedure the tool must check the available resources and all the other assumed characteristics of the units, like their geographical positions and their functions. Usually several merging transformations are possible and one transformation can make other possible transformations impossible, which could lead to much better configurations later. In order to find the best solution all these possibilities should be examined. It is necessary to recursively test all possible merging transformations for all configurations found in the completion phase. The chance of losing better merging transformations is a further reason why it is not possible to look for double units already during the completion phase and add only units, which really do not exist so far. Configurations with missing merging possibilities, which still contain some units not respecting their resource limits will be rejected. Figure 6 presents an example of a simple search tree showing the two phases of the search. As already described above, the nodes in the tree represent configuration graphs consisting of many units. Each child node of such a configuration graph specifies a configuration graph with further attributes of the units, connections or services set.

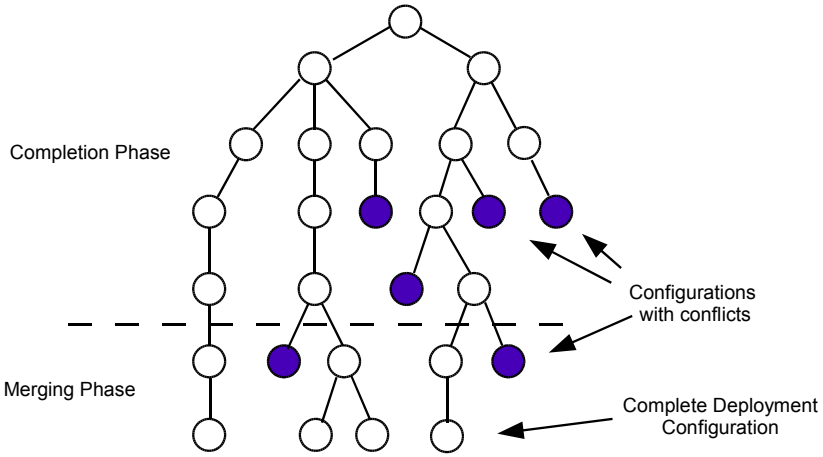

Fig. 6. Search Tree

If no single path without conflicts can be found an appropriate message will be shown to the user. The message will also state the reasons for the failure of the search, usually naming the products causing the problems. To get this information the third re-

quest type of the Component Preselector is used. In this case the user can either add new products to the knowledge base or review the original configuration file. Each change requires a new start of the search afterwards. The program valuates all successfully completed and merged configurations according to the amount and the costs of all new devices and software components. Then it offers the best solutions to the user. If there are still some attributes of the units left unspecified the user will be asked to choose between all products suiting the properties. In the final description of the eHome system all devices and software components are specified by concrete products. The Runtime Instancer can find all information needed for the deployment of the services on the framework.

The tool shown in figure 7 implements the described algorithm. The user interface integrates some features of a simple XML viewer, which allow to manage and to navigate through the configuration files. During the configuration process the tool interacts with a simple Component Preselector whose implementation is based on the Jess [20] rule engine and on a knowledge base represented in an OWL file [23]. Unavoidable

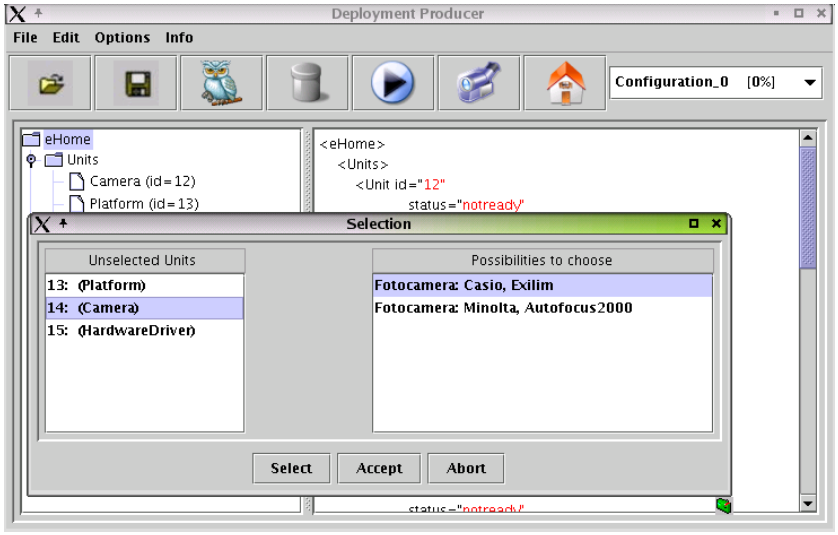

Fig. 7. DeploymentProducer

conflicts during the search are reported and finally the remaining choices for equivalent products are offered to the user. If a conflict is caused by a missing description of units, this description has to be developed manually. This is a major task and handled in another paper [24]. If all software components are already available it is also possible to test the found configuration by deploying it on the given framework.

Figure 8 shows small cutouts of a successful automatic configuration process for a security scenario on the basis of figure 3. In this example, the scenario configuration consists of the request for a security service and the fact that 2 cameras are al-

ready installed on an OSGi gateway. This is translated by the Deployment Producer with the specific information of the destination house (here depicted with a floor plan) and general information (driver, dependency, platform and service descriptions) from an OWL-based knowledge base into a customized deployment configuration document. This document includes all information, which is needed by the Runtime Instancer to load and startup all software components to run the security service. These are on the one hand the OSGi capable security controller and on the other hand the drivers to control the alarm, the sensors, and the cameras. Furthermore, the sequence of starting the components is coded in this document. The Runtime Instancer can generate a document, which stores current states in addition to the former mentioned information. The Component Preselector translates questions like ”What’s the OSGi driver for this camera?” or ”Which components are needed at least for this security service?” in queries for the knowledge base, which can be operated by the Search Machine and the Inference Machine.

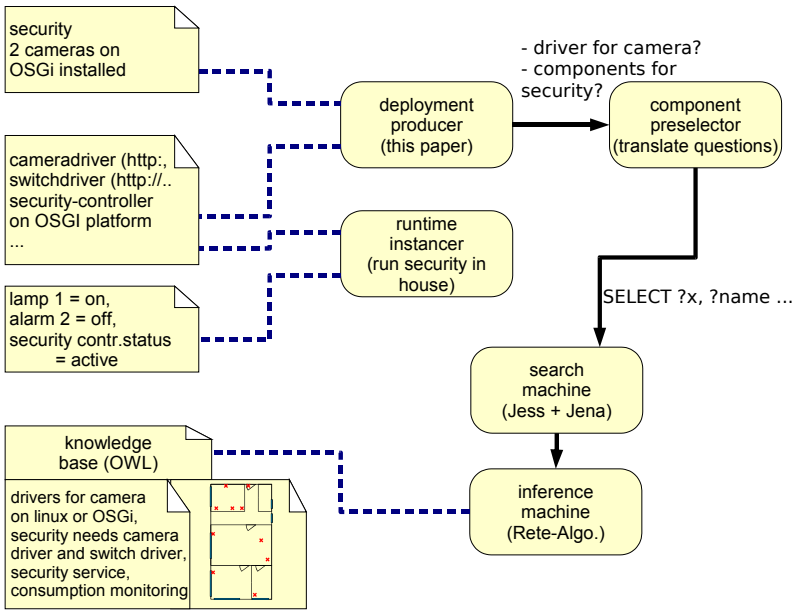

Fig. 8. Example Applied to Configuration Document Levels

The process was evaluated with some simple examples from the security domain with up to 40 units. In this case, the computation time for producing a deployment configuration document was less than a minute on a usual desktop computer. However, we know that this time heavily benefits from the small knowledge base we used.

Additionally, the intended scenario was predefined and thus the specification of the necessary dependencies and attributes could easily be derived. Tests in more realistic environments are current work with our cooperation partner inHaus [25]. Realistic environments require tests with bigger knowledge bases, more devices, and more complex services.

6 Summary & Outlook

One of the major problems hindering broad application of eHome systems is the expected variance of different configurations. To cope with these problems, applying methods from configuration management and software deployment combined with process automation seems to lead to reasonable solutions, because the restriction to the eHome application domain enables the usage of a clear-cut and comprehensive ontology. Automation requires a uniform description language for all configuration and deployment steps. Languages partly capable of fulfilling our requirements are widely used in the area of architecture description languages and component frameworks. The proposed language combines the features required for an automated eHome configuration and deployment process, as discussed in section 3. We developed a tool to automate one step of the configuration and deployment process. The developed Deployment Producer is able to create service configurations based on rather abstract system descriptions. Promising tests with our cooperation partner inHaus [25] show the applicability of our approach in a realistic environment.

However, a great number of units is expected in future eHome systems. The development of more efficient, heuristic search algorithms will be necessary due to computational complexity. Strategies for keeping the knowledge base up-to-date must be found. The more available products are taken into consideration, the better the results are the tool can offer. Further integrative work has to be done: The interoperability of the Component Preselector, Deployment Producer, and the Runtime Instancer has to be extended. Tight integration in the underlying frameworks like the Distributed Service Framework (DSF) [6] and the Layered Approach to OSGi (PowerArchitecture) [5] has to be further investigated, in order to ease the realization of Web-enabled eHome systems.

References

1. Lee, W.M., Foo, S.M., Watson, K.: Beginning WAP, WML, and WMLScript. Wrox Press Ltd (2000)
2. 3GPP: Multimedia Messaging Service, TS 22.140. `http://www.3gpp.org` (2002)
3. Westfechtel, B., Conradi, R.: Version Models for Software Configuration Management. ACM Computing Surveys **30** (1998)
4. van der Hoek, A.: Integrating Configuration Management and Software Deployment. In: Proceedings of the Working Conference on Complex and Dynamic Systems Architecture (CDSA 2001). (2001)
5. Kirchhof, M., Linz, S.: Component-based Development of Web-enabled eHome Services. In: Proceedings of Ubiquitous Mobile Information and Collaboration Systems Workshop 2004 (UMICS 2004). Lecture Notes in Computer Science, Springer (2004) to appear.

6. Kirchhof, M.: Distributed and Heterogeneous eHome Systems in Volatile Environments. In: Proceedings of Forum at 2^{nd} International Conference on Service Oriented Computing (ICSOC 2004), IBM Technical Reports (2004) Refereed Papers, to appear.
7. Dashofy, E.M., van der Hoek, A., Taylor, R.N.: A Highly-Extensible, XML-Based Architecture Description Language. In Kazman, R., Kruchten, P., Verhoef, C., van Vliet, H., eds.: Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01). Volume ix., IEEE Computer Society, Washington, DC, USA (2001)
8. Dashofy, E., Garlan, D., van der Hoek, A., Schmerl, B.: xARCH. http://www.isr.uci.edu/architecture/xarch/ (17.06.2004) (2002)
9. General Dynamics Decision Systems: Openwings. (http://www.openwings.org (10.05.2004))
10. Smith, M., Carpenter, J.: Openwings Policy Service Specification Ver. 1.0 Final. General Dynamics Decision Systems, INC. (2003)
11. Dennis Reedy: Project Rio. (http://rio.jini.org (10.05.2004))
12. Sun Microsystems, Inc.: The Community Resource for Jini Technology. (http://www.jini.org (8.5.2004))
13. Open Services Gateway Initiative: OSGi Service Platform. http://www.osgi.org (13.11.2003) (2003) Release 3.
14. Prosyst Software AG: mBedded Server 5.x. (http://www.prosyst.de/solution_html/mbeddedserver.html (22.6.2004))
15. Kirchhof, M., Norbisrath, U.: Configuration and Deployment in eHome-Systems. In: Proceedings of Information Systems: New Generations (ISNG 2004). (2004) to appear.
16. Sun Microsystems, Inc.: Java. http://java.sun.com (06.05.2004) (2004)
17. Open Services Gateway Initiative: OSGi Service Platform. (http://www.osgi.org (13.11.2003))
18. Waring, D.L., Kerpez, K.J., Ungar, S.G.: A newly emerging customer premises paradigm for delivery of network-based services. In: Computer Networks. (1999) 411–424
19. Saito, T., Tomoda, I., Takabatake, Y., Ami, J., Teramoto, K.: Home Gateway Architecture and Its Implementation. IEEE Transactions on Consumer Electronics **46** (2000) 1161–1166
20. Friedman-Hill, E.: Jess, The Rule Engine for the Java Platform. (2003) Version 6.1.
21. Kirchhof, M., Stinauer, P.: Service Composition for eHome Systems: A Rule-based Approach. (2004)
22. World Wide Web Consortium (W3C): XML - Extensible Markup Language. (http://www.w3.org/XML/ (06.05.2004))
23. Bechhofer, S., van Harmelen, F., Hendler, J., Horrocks, I., McGuinness, D.L., Patel-Schneider, P.F., Stein, L.A.: OWL Web Ontologie Language Reference. http://www.w3c.org/TR/2003/PR-owl-ref-20031215/ (2003)
24. Norbisrath, U., Salumaa, P., Schultchen, E., Kraft, B.: Fujaba based tool development for eHome systems. In: Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004). Electronic Notes in Theoretical Computer Science, Elsevier (2004) to appear.
25. inHaus Duisburg: Innovationszentrum Intelligentes Haus Duisburg. (http://www.inhaus-duisburg.de (22.6.2004))