Journal of Universal Computer Science, vol. 16, no. 12 (2010), 1592-1611
submitted: 29/12/09, accepted: 5/5/10, appeared: 28/6/10 © J.UCS

Configuration Process of a Software Product Line for AmI Middleware

Lidia Fuentes
(University of Malaga, Málaga, Spain
lff@lcc.uma.es)

Nadia Gámez
(University of Malaga, Málaga, Spain
nadia@lcc.uma.es)

Abstract: Developing Ambient Intelligence applications is a very complex task since it implies dealing with low-level software and hardware resources. The use of a middleware platform may alleviate this task by providing a set of high-level and platform-independent services to these kinds of applications. Nevertheless, the tendency is that the middleware deployed in each device has a flat and homogeneous architecture, although these devices and the requirements of intelligence environments are heterogeneous. This implies the middleware software deployed in each device normally contains more functionality than strictly required, leading to waste resources so scarce in lightweight devices. But the configuration and deployment of a minimal middleware customized to a target platform is a complex task, due to the diversity of hardware and software present in devices and the variable requirements of ambient intelligence applications. In order to solve these shortcomings, we propose to customize the piece of software related to the middleware platform by using a Software Product Line engineering approach. This paper presents an innovative configuration process for a software product line for ambient intelligence middleware where a minimal set of high-level parameters needs to be specified. So, the software engineers for this kind of systems can automatically obtain customized middleware by simply specifying this high-level information.

Keywords: AmI, Middleware, Variability, SPL, AAL.
Categories: C.2.1, D.2.1, D.2.2, D.2.11, D.4.7

1 Introduction

Most Ambient Intelligence [Bravo, 06] (AmI) systems nowadays are comprised of a set of intercommunicated and heterogeneous devices, which range from small sensors to PDAs or mobile phones. Developing AmI applications is a very complex task since it requires a thorough knowledge of the software deployed on these devices. It also requires managing the hardware resources provided by them as well as dealing with network issues. This implies that AmI applications need to deal with low-level APIs and operating system functions to manage these hardware resources. An additional problem is that these low-level APIs and functions are sometimes vendor-dependent and vary among the different devices and/or networks.

A well-known solution to this problem in the software community is to create a middleware platform that offers a set of high-level and platform-independent services to AmI applications (e.g. access to battery, memory, sensor location, etc.). This helps to hide the particularities of the different APIs when managing hardware and network resources and contributes to the portability of AmI software to different target devices [Fuentes, 09]. As a result, several middleware platforms for AmI systems or for embedded or pervasive systems have been created in recent years. Nevertheless, an important limitation is that they are developed focusing on mobility or real-time issues, or are specifically for wireless and sensor networks (WSNs) [Wang, 08], but they do not cover the necessities of all types of devices or applications.

An additional shortcoming is that the internal middleware architecture is normally *homogenous*, providing a fixed set of services, although target devices and the requirements of each AmI application are *heterogeneous*. This implies the middleware software deployed in each device contains more services than strictly required. For instance, the middleware can contain a service for encryption task even when the application does not require it. As a result, the size of the piece of software implementing the middleware is larger than necessary. In desktop or server applications, where memory consumption is not a critical issue, this is not a problem. But in AmI applications, where each kilobyte is appreciated, this can be a limitation.

Summing up, a very specific AmI middleware and/or not optimized in size can not be deployed on any device and for any application. In order to address this shortcoming, we propose to customize the piece of software related to the middleware platform that is deployed in each device according to: (1) the device features; (2) the network used to connect the different devices; and (3) the AmI application that these devices run. So, basically, we are looking for a technique that allows easy customization of a middleware according to the requirements previously described.

Software Product Line (SPL) engineering [Pohl, 05] aims to provide a set of tools and techniques for creating infrastructures that allow the rapid and systematic production of similar software systems for a specific market segment. Therefore, SPL engineering seems to be the most appropriate technique to create an infrastructure from which we can construct customized middleware to be deployed in embedded heterogeneous devices with different application requirements.

Nevertheless, the customization process for deriving specific AmI middleware from the SPL infrastructure can be a repetitive, laborious and error-prone task. This is mainly due to the large variability inherent in the AmI software domain, which implies a large number of options and parameters must be specified during the configuration process. Moreover, although the SPL infrastructure provides a set of precise rules for customizing a product according to the parameters previously specified, in typical SPL engineering, these rules need to be applied manually, which is time-consuming and, since software engineers are human, it is not free of error.

To address this problem, this paper presents an innovative configuration process for a SPL for AmI middleware where only a minimal set of high-level options and parameters need to be specified. Using this set, a larger set of low-level options and parameters is automatically calculated. Using model transformations, the design for a specific AmI middleware, customized for the parameters previously calculated, is automatically constructed from a generic middleware design model. Finally, using code generation, 100% of the code for deploying the middleware in a specific device is obtained. So, using this approach, software engineers can automatically obtain customized middleware by specifying a minimal set of high-level features.

The reminder of the paper is organized as follows. In Section 2 we present our motivation describing the problems of developing and customizing middleware for AmI systems and how our process for the customization of the middleware helps to address these problems. In Sections 3 and 4 the process is described in detail and case studies are presented to illustrate it. The evaluation of our approach, some discussion and related work are presented in Sections 5 and 6 respectively. Finally, in Section 7 we outline some conclusions and future work.

2 Motivation

2.1 Problem

As already commented in the introduction, the initial idea proposed by several authors is to deploy a middleware platform in those devices which offer a set of services for developing AmI applications, abstracting from low-level details of device and networks [Sarnovsky, 08]. Nevertheless, state-of-the-art middleware for AmI systems are *homogenous* in spite of the fact that the devices that comprise an AmI application are *hetereogenous*. This has the limitation described below.

Figure 1 illustrates an AmI application for an Accidental Fall Report [Keshavarz, 06], which is responsible for reporting accidental falls of elderly and/or disabled people. The application is comprised of a set of image sensor nodes (cameras), three wall-mounted sensors able to measure distances, one user badge node and one sensor with a modem that communicates with a central call center. These devices must work coordinately for detecting and properly reporting falls of elderly people. The user badge has an accelerometer and a voice circuit for transmitting voice. When the accelerometer detects a potential fall scenario, the three wall-mounted sensors are informed and they calculate the exact position where the elderly person might have fallen. At the same time, the central call center is informed and the nearest image sensor is activated. The sensor sends the images to the modem, which transmits the video signal to the call center. Here, a worker checks the video stream and confirms if it is indeed a fall or, instead, a false alarm. All these devices communicate by radio technology, more specifically by means of ZigBee (802.15.4), and the sensor with the modem is connected to the internet broadband. Moreover, according to the personal data protection laws, images sent through a ZigBee connection must be encrypted. So, the middleware for this application would need basic services for communication, data delivery discovery or location, plus services for encryption or coordination tasks. Therefore, we should construct a middleware platform with at least these services. It should be noticed that not all services are required for all the devices at the same time. For instance, the movement tracking algorithm running in the wall-mounted devices requires the coordination service since it is a collaborative application. Only the image nodes and the sensor with the modem require an extra service of security since the generated image data must be encrypted.

If we deployed a *homogenous* middleware in the application, i.e. a middleware containing all the services that might be required by any device, then each device would deploy services which were never going to be used. For instance, the user badge would have deployed an encryption service, even though the user badge does not have to encrypt any data. Similarly, all the devices would have deployed services for communicating on a broadband network when in fact only the node with the modem is connected to a broadband network. This implies some memory is simply wasted, since it would be devoted to useless services. In the AmI systems domain, where memory consumption is critical, this can be a serious limitation. Also, the AmI middleware must be implemented using the appropriate operating system and API required by these devices.

In order to overcome these shortcomings, we propose to customize the middleware platform according to needs of each device or application. Since this customization process can be laborious, time-consuming and error prone, we propose to apply SPL engineering techniques to automate this configuration process. The next section gives an overview of our solution.

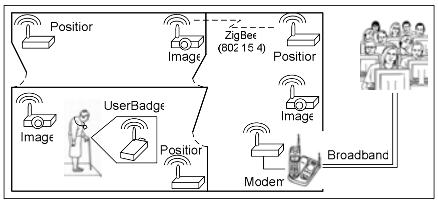


Figure 1: Scenario for Accidental Fall Report System.

2.2 Overview of our approach: A Software Product Line for AmI Middleware

A SPL aims to create the infrastructure for the rapid production of similar software systems belonging to the same domain [Pohl, 05]. Thus, SPL engineering seems to be a suitable solution for our goal, which is to build similar, easily customizable AmI middleware systems. In this article we will present how to construct a SPL specifically for AmI middleware. One of the salient contributions of this article, is that the process for customizing a middleware platform using the SPL infrastructure is largely simplified both by reducing the number of parameters and options the application developer needs to specify and also by automating several parts of the customization process, or in SPL terms, of the *product derivation* process.

Any SPL is comprised of two phases, or related development processes: *Domain Engineering* and *Application Engineering*. *Domain Engineering* deals with the creation of the infrastructure from which specific or customized products will be constructed. *Application Engineering* is concerned with the creation of specific or customized products using the infrastructure that has been previously created at the domain engineering level. Domain engineering needs to be carried out only once per family of products, whereas application engineering must be performed each time we want to construct a new product belonging to the family. If the family of products needs to evolve, we must update the infrastructure created at the domain engineering.

In our case, Domain Engineering is concerned with the creation of a flexible AmI middleware design and implementation, one which supports all the customizations that might be required. Application Engineering would be concerned with the customization of this flexible design to obtain a specific middleware platform that fits appropriately with the specific device, network and application requirements of a particular AmI system. In our case, we use model transformations and code generation techniques to automate part of this process. Figure 2 gives a general overview of our approach. Figure 2 (top) represents *Domain Engineering* and Figure 2 (bottom) *Application Engineering*. The very first step when creating a SPL is to analyze the variability inherent in the domain, in our case, to analyze the variability inherent in AmI middlewares. For this task, a *feature model* (FM) [Loughran, 08] is constructed (See Figure 2, label D1). A FM specifies which elements, or *features*, of the family of products are common, which are variable and the reasons why they are variable, i.e. if they are alternative elements or optional elements. In addition, it is possible to specify constraints or dependencies between these elements. For instance, a dependency might specify that if the device where the middleware is going to be deployed is in an insecure network, a service for encrypting must be automatically added to the middleware. How this FM is constructed is explained in Section 3.1.

Due to the large variability inherent in the AmI middleware domain, we aim to greatly reduce the number of features a user needs to specify in order to customize a product by exploiting these *dependencies* that are established between features belonging to different levels of abstraction. This means that the selection of a group of high-level features implies the selection of one or more low-level functions. The rationale behind this organization is that the user only has to specify a minimum set of high-level features for customizing the middleware. Then, the minimum set of low-level features that must be selected to obtain a specific middleware is automatically inferred using a constraint solver provided by a feature modeling tool we have constructed, called Hydra[1].

Once we have constructed a FM for our domain, the next step is to create a flexible design, or *product line architecture* (PLA), which supports the variations specified by the FM (Figure 2, label D2). This design must allow its customization for creating any middleware included in the family of products. This flexible design is explained in Section 3.2.

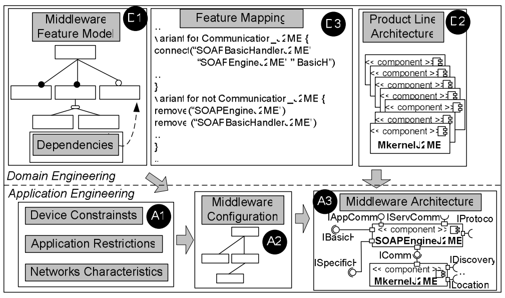
Finally, we need to specify somehow how this flexible design must be customized as different features are selected or *un*selected. For instance, the selection of a specific service might require the addition of several software components to the middleware. Similarly, the selection of a certain feature might imply setting a component parameter to a specific value. This link between a FM and a PLA is specified (Figure 2, label D3) using an innovative language developed in the context of the AMPLE project[2], called VML4Arch [Loughran, 08]. This language allows the automation of the rules for customizing products in a SPL context as well as the abstraction of low-level details from model transformation languages. This allows this mapping to be specified in VML4Arch even by software architects without skills on model transformation languages. This is explained in Section 3.3.

Once we have completed the domain engineering process, we are ready to customize products, i.e. to move to the application engineering phase (See Figure 2, bottom), which is repeated for each product we want to customize. The first step for obtaining a customized middleware is to create a configuration of the FM, i.e. a selection of features to be included in the customized product. As explained before, to reduce the number of features a user must select, s/he only would need to deal with a

[1] http://caosd.lcc.uma.es/spl/hydra
[2] http://www.ample-project.net

reduced set of high-level features, which corresponds to requirements imposed by the device (e.g. vendor of the device), by network (e.g. number of nodes) and by the application (e.g. coordination requirement) (See Figure 2, label A1). With a selection of these high-level features as input, Hydra calculates an optimal complete configuration of the feature model for customizing the middleware (See Figure 2, label A2). This configuration is given as input to the VML4Arch, which, by means of generated model transformations, will automatically produce the design model for the customized middleware as output (See Figure 2, label A3). Finally, we give this customized design model to a code generator, which produces the 100% of the code for deploying this middleware into a specific device. The advantages we are:

- Only the minimal middleware configuration will be installed in each device, thereby making use of device resources in a energy efficient way.
- The configuration process is substantially simplified since the configuration logic is not part of the middleware, but of the SPL derivation process.
- Using MDD technologies the product derivation process is automatic, so each time that we need install the middleware in one particular device, new middleware configurations can be derived simply by pressing some buttons.



```
..
variant for Communication_J2ME {
connect("SOAPBasicHandlerJ2ME"
            "SOAPEngineJ2ME" " BasicH")
..
}
variant for not Communication_J2ME {
remove("SOAPEngineJ2ME")
remove ("SOAPBasicHandlerJ2ME")
..
}
..
```

Figure 2: Overview of our Approach.

3 Domain Engineering for an AmI Middleware

Domain Engineering in our context deals with the creation of the infrastructure which will enable the rapid, or in our case automatic, construction of a family of AmI middleware applications. Section 3.1 will focus on the first step to construct such a family of AmI middleware applications, which is to analyse and specify, using a feature model, the different kinds of variations that exist between different middleware platforms. Once variability of the family of products has been identified, the next step is to design a flexible system that supports this variability (see Section 3.2). Finally, as we commented in a previous section, we need to specify which actions must be performed when a certain feature is selected or deselected. This is described in Section 3.3. This phase only has to be realized once for the whole family of AmI middleware and need only be modified for the evolution of the middleware.

3.1 Feature Model

A FM [Lee, 02] allows us to specify which parts of a system are variable, independently of the software design of our middleware. This allows high level reasoning about all the different possible configurations. In the AmI systems domain we can identify three different viewpoints reflecting the different stakeholders of such systems: the builder of the physical devices, the network expert, and the application domain expert. Then, we have divided our FM into three main kinds of features: **Device Driven**, **Network Driven** and **Applications Driven Features**. We have modeled our FM using a tool we have developed, called Hydra. Hydra is a feature modeling tool, provided as an Eclipse plugin based on Ecore+GMF. Hydra, as compared to state-of-the-art feature modeling tools, offers full graphical capabilities, both for editing FM and configurations. Figure 3 shows a screenshot of Hydra with the complete FM of our middleware.

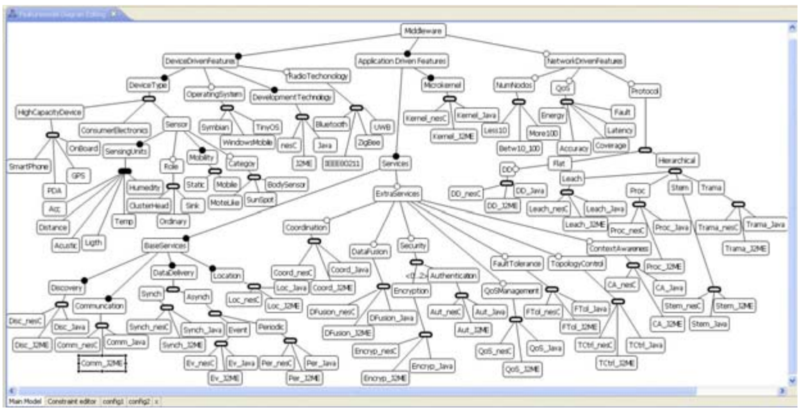


Figure 3: AmI Middleware Feature Model with Hydra.

Device Driven Features basically concern the hardware properties. A device is characterized by its **Type,** which refers to the device hardware architecture and capacity (the available computational resources) and determines the supported **Operating System**. Devices can be classified into three categories: **High Capacity Devices** (as smart phones or GPS devices), **Sensors** and **Consumer Electronics**. The **Device Type** and **OS** also influence which **Development Technologies** (APIs) are available to build the embedded software. The **Development Technology** captures the variability of the available APIs and programming languages for every version of each distribution of an OS. Finally, the last device feature is the **Radio Technologies** that can be used in the device, such as the **IEEE 802.11**, the **ZigBee** or **Bluetooth**.

Network Driven Features include the logical topologies used to organize the nodes and specific protocols used to exchange data and control information inside the network. In AmI systems most of the networks are Wireless Sensor Networks (WSNs), consisting of a set of sensor nodes linked by a wireless medium. They are able to perform distributed sensing and to convey useful information to control stations. In large WSNs the data delivery in **one hop**, i.e., directly from sources nodes to sink nodes, consumes too much. A more efficient approach is to send data in **multiple hops** using intermediary nodes. With such approach, the paths from sources to sinks are managed by data dissemination protocols, which can be classified according to the network logical topology into hierarchical and flat protocols.

Finally, **Application Driven Features** encompass the internal structure (components) of the middleware. To follow an SPL approach, the middleware must be structured in fine-grained components (**Services**) decoupled from a base infrastructure (**Microkernel**). The middleware services for our proposed family are classified as basic (mandatory for the whole middleware family) and extra (optional, dependent on the specific product to be generated). Basic services are: **discovery**, **communication, data delivery**, and **location**. The **discovery** service provides a mechanism to enable nodes to advertise and know the capabilities of their neighboring nodes and to allow applications to submit a high level description of their requirements. The **communication** service includes a proxy responsible for interacting with applications, and drivers for communicating with the underlying network protocols and devices. The **data delivery** service delivers the network generated data to the application and it is dictated by the application requirements. This service may be **synchronous** or **asynchronous.** Finally, the **location** service allows a node to know its own geographical position and to store this value for posterior use. In addition, we have identified the need for the following extra services in this domain: **coordination**, **data fusion**, **security, QoS management, fault tolerance, topology control** and **context-awareness** [Bravo, 05]. For a detailed description of these services we refer the reader to [Delicato, 09].

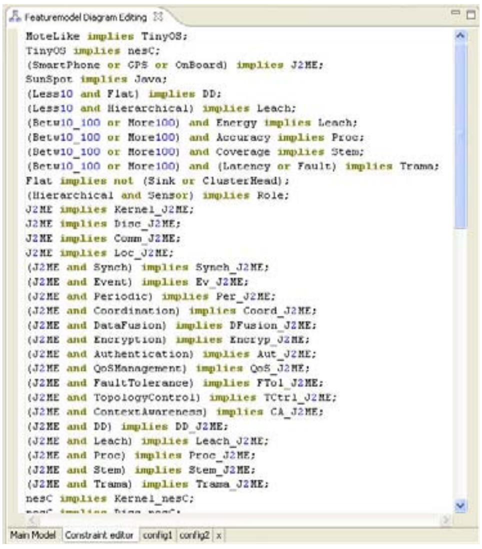
Nevertheless, the features mentioned are not independent of each other. For instance, a given network protocol (e.g. Trama [Rajendran, 03]) may be more suitable to one specific application (e.g. Road Safety) than another protocol. This means that, in addition to the FM, it is necessary to define *Dependencies* between features. These dependencies are articulated in terms of logical expressions that can be reduced to *usage* and *mutual exclusion* dependencies. A usage dependency could be formulated as, “if a feature A is selected, then a feature B has also to be selected”. Mutual exclusion would be, “if a feature A is selected, feature B must not be selected”.

As already commented in a previous section, in order to reduce the number of features specified by a developer to customize the middleware platform, we have divided the features into different levels of abstraction and we have also defined dependencies between low-level features and high-level features. For instance, the developer must specify the number of nodes of the networks and the QoS required by the application (for example, low latency) and with this information our process infers that the most appropriate network protocol for this application is Trama, but the developer does not need to select manually the protocol.

The Hydra tool is able to infer, using a constraint solver called *Choco*[3], the minimum number features to be added to a configuration in order to create a valid configuration that satisfies the constraints defined by the dependencies. Therefore, we use this capability and the dependencies between features, for calculating how many low-level features must be included to create a valid configuration according to a certain set of high-level features given as input. This frees the developer to deal with

[3] http://www.emn.fr/x-info/choco-solver/doku.php

low level decisions, while ensuring that a valid configuration of the FM is deployed. Furthermore, using Hydra we avoid the undesired side effects that may be produced in the manual selection of features. The configuration process is simplified by means of reducing the number of features that need to be specified.



```
MoteLike implies TinyOS;
TinyOS implies nesC;
(SmartPhone or GPS or OnBoard) implies J2ME;
SunSpot implies Java;
(Less10 and Flat) implies DD;
(Less10 and Hierarchical) implies Leach;
(Betw10_100 or More100) and Energy implies Leach;
(Betw10_100 or More100) and Accuracy implies Proc;
(Betw10_100 or More100) and Coverage implies Stem;
(Betw10_100 or More100) and (Latency or Fault) implies Trama;
Flat implies not (Sink or ClusterHead);
(Hierarchical and Sensor) implies Role;
J2ME implies Kernel_J2ME;
J2ME implies Disc_J2ME;
J2ME implies Comm_J2ME;
J2ME implies Loc_J2ME;
(J2ME and Synch) implies Synch_J2ME;
(J2ME and Event) implies Ev_J2ME;
(J2ME and Periodic) implies Per_J2ME;
(J2ME and Coordination) implies Coord_J2ME;
(J2ME and DataFusion) implies DFusion_J2ME;
(J2ME and Encryption) implies Encryp_J2ME;
(J2ME and Authentication) implies Aut_J2ME;
(J2ME and QoSManagement) implies QoS_J2ME;
(J2ME and FaultTolerance) implies FTol_J2ME;
(J2ME and TopologyControl) implies TCtrl_J2ME;
(J2ME and ContextAwareness) implies CA_J2ME;
(J2ME and DD) implies DD_J2ME;
(J2ME and Leach) implies Leach_J2ME;
(J2ME and Proc) implies Proc_J2ME;
(J2ME and Stem) implies Stem_J2ME;
(J2ME and Trama) implies Trama_J2ME;
nesC implies Kernel_nesC;
```

Figure 4: Dependencies between Features in Hydra.

Figure 4 shows the dependencies between features for the model depicted in Figure 3. We have used the Constraint Editor of Hydra to specify them. There are several *usage dependencies* between the **Device Type**, the **OS** and the **Development Technologies**. For example (see Figure 4), if the **MoteLike** (the most common sensor device) feature is selected then the **TinyOS** (the OS commonly used for sensors) feature has to be selected and **nesC** (the programming language for TinyOS) development technology must be selected. It should be noticed for instance, if the nesC feature is selected, Hydra would automatically infer that the Kernel_nesc, the nesC version of the base services and the nesC version of the selected extra services and protocol, must be selected. The same thing happens with the rest of development technologies and other dependencies.

3.2 Product Line Architecture

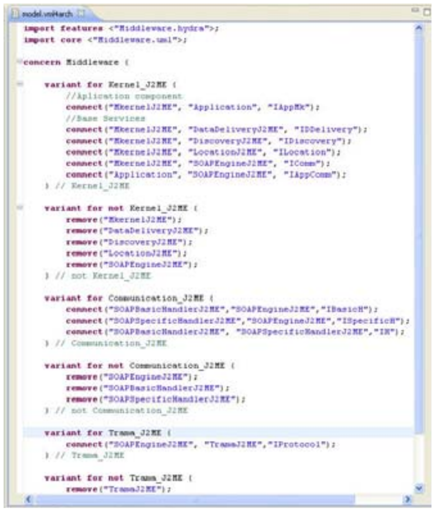
Once the FM has been designed and all the variability in the AmI middleware has been identified, we need to design a flexible architecture that supports these variations. Such architecture was designed using the component model of UML 2.0.

As the mechanism for supporting the variations, we use a schema where plug-in components, implementing different services, are plugged into a microkernel. The microkernel is in charge of loading and composing the middleware services with application components. The microkernel term describes a form of operating system design in which the amount of code that must be executed in privileged mode is kept to an absolute minimum. As a consequence, the rest of the services are built as independent modules that are plugged in and executed by the kernel on demand. In this way, a more modular and configurable system is obtained.

The correspondence between features and components is largely far from being trivial. Each feature is often designed by using more than one component, depending either on the implementation strategies of the service, on the network protocols or on the development technology. So, the rules for selecting or deselecting components according to a feature selection must be, at least, made explicit using some kind of language; or, in the ideal case, automated. We explain in the next section how these configuration rules can be automated.

3.3 Feature Mapping

To automate the rules that describe how a software model must be configured according to a certain feature selection, we use the VML4Arch language. VML4Arch[4] is an innovative language for specifying SPL configuration processes for architectural models. The main contribution of VML4Arch is that it provides automation at the same time that it hides low-level details of general-purpose model transformation languages. Using high-order model transformations, VML4Arch *compiles* a description of the configuration process into a set of low-level general purpose model transformations, which, when executed, are able to customize the product line architectural model according to a feature selection. This allows the VML4Arch to be used even by software architects without skills in model transformations.



```
import features <"Middleware.hydra">;
import core <"Middleware.uml">;

concern Middleware {

    variant for Kernel_J2ME {
        //Aplication component
        connect("MkernelJ2ME", "Application", "IAppMk");
        //Base Services
        connect("MkernelJ2ME", "DataDeliveryJ2ME", "IDDelivery");
        connect("MkernelJ2ME", "DiscoveryJ2ME", "IDiscovery");
        connect("MkernelJ2ME", "LocationJ2ME", "ILocation");
        connect("MkernelJ2ME", "SOAPEngineJ2ME", "IComm");
        connect("Application", "SOAPEngineJ2ME", "IAppComm");
    } // Kernel_J2ME

    variant for not Kernel_J2ME {
        remove("MkernelJ2ME");
        remove("DataDeliveryJ2ME");
        remove("DiscoveryJ2ME");
        remove("LocationJ2ME");
        remove("SOAPEngineJ2ME");
    } // not Kernel_J2ME

    variant for Communication_J2ME {
        connect("SOAPBasicHandlerJ2ME","SOAPEngineJ2ME","IBasicH");
        connect("SOAPSpecificHandlerJ2ME","SOAPEngineJ2ME","ISpecificH");
        connect("SOAPBasicHandlerJ2ME", "SOAPSpecificHandlerJ2ME","IH");
    } // Communication_J2ME

    variant for not Communication_J2ME {
        remove("SOAPEngineJ2ME");
        remove("SOAPBasicHandlerJ2ME");
        remove("SOAPSpecificHandlerJ2ME");
    } // not Communication_J2ME

    variant for Trama_J2ME {
        connect("SOAPEngineJ2ME", "TramaJ2ME","IProtocol");
    } // Trama_J2ME

    variant for not Trama_J2ME {
        remove("TramaJ2ME");
```

Figure 5: Feature Mapping in VML4Arch.

[4] http://caosd.lcc.uma.es/spl/vml/vml4arch

Using VML4Arch, we specify which actions, or *customizations*, must be performed on the architectural model of the AmI middleware when a certain feature is selected. Figure 5 shows a screenshot of the mapping between the feature model of Figure 3 and the UML components of the architectural model. The code shown in Figure 5 presents the mapping of the microkernel and the communication service when the development technology selected is J2ME and the network protocol used is Trama [Rajendran, 03]. This specification establishes that whenever the Kernel_J2ME is selected, the J2ME versions of the basic services must be connected to the MkernelJ2ME component through the corresponding interfaces. In the case where the Kernel_J2ME variant is not selected all the J2ME base services must be removed from the architecture. If the Communication_J2ME feature is selected the SOAP engine and the handlers must be connected, and if it is not selected, all these components must be removed. Finally, if the Trama_J2ME feature is chosen the TramaJ2ME component has to be connected throng the IProtocol interface with the SOAPEngineJ2ME component. The same is done for the other protocols and each development technology. Moreover, it must also be done for the communication service in the other development technologies and for the other services.

4 Application Engineering for an AmI Middleware

Application Engineering is concerned with the engineering of specific products or single software systems using the infrastructure previously created at the Domain Engineering level. In our case, we will customize a specific middleware instantiation depending on the requirements of a particular device, network and application. This phase must be repeated every time the device, network or application change. We illustrate this process by customizing our middleware for two different AmI domains: the Vehicular Ad-hoc Networks (VANETs) [Dikaiakos, 2007] and the Ambient Assisted Living (AAL) [5] domain. Note that we use both case studies to detail how our configuration process works and which services our middleware provides to these applications. Nevertheless, it is out of our scope to describe both systems completely.

A VANET is composed of a set of autonomous vehicles that can operate with minimum help from the driver. Each vehicle is equipped with sensors (acoustics, accelerometers, temperature, etc.) an on board computer with a GPS. In VANET, vehicle sensors are used to ensure safe driving, by measuring the distance with respect to other objects/obstacles in the vicinity, e.g. other vehicles, pedestrians, etc. Vehicle sensors use information received from the surrounding vehicles, such as their speed or distance and are able to communicate with each other in order to cooperate for safety and other purposes. Finally, traffic signals can send information to the vehicles informing them of speed limits and general traffic regulations. The VANET domain encompasses a large range of specific application sub-categories, such as road safety applications (accident warnings, red-light warnings, etc.), information dissemination (parking spots, fuel prices, etc.), entertainment applications or assisted driving. The vehicles can be considered as distributed sensors that collect data and report them to base stations. In this section we use a road safety application.

[5] http://www.aal-europe.eu/

The goals of the AAL applications are to prolong the time people can live safely and comfortably in their own home by increasing their autonomy and self-confidence [Delicato, 09]. WSNs are a crucial component of AAL applications, supporting the provision services for elderly and disabled people, such as position location and movement tracking. We will use the Accidental Fall Report described in Section 2.1.

The first task at this level is to specify a set of selected high-level features, according to the requirements of the device, the network and the application (see Section 4.1). Then, this set of high-level features are given as input to Hydra, which, using also the FM model and the dependencies, calculates the minimum set of low-level features that must be added to the configuration in order to create a valid configuration. This is described in Section 4.2. Finally, using this valid configuration as input, VML4Arch automatically customizes the architectural model created at the domain engineering level according to this particular configuration. The customized architectural model is given as input to a code generator, which automatically produces 100% of the code for deploying the AmI middleware (Section 4.3).

4.1 Eliciting Device, Network and Application Requirements

The goal of this step is to obtain a selection of high-level features according to the requirements imposed by the device, the network and the AmI application running on the device. In order to avoid the user needing to deal directly with the FM, we have developed a simple tool for gathering this information in a user-friendly way. This tool has three windows, each one having a form, where the experts have to select in which kind of device, application or network the AmI middleware must be installed. Each form is represented with one XML document with respect to one XML Schema, or, in terms, as a model conforming to a metamodel.

The **Device Constraints** document specifies the device profile, including its type (smartphones or sensors) and vendors, among others. The **Application Restrictions** express the subset of the available middleware services that will be required by the application and the application QoS requirements. Finally, the **Network Characteristics** specify details of network-level protocols, used to manage the sending of messages and the multihop routing including the QoS parameters they fulfil. For instance, some protocols are more energy efficient, others are fault-tolerant and others address time requirements. With this information, the tool creates an initial set of selected high-level features, which satisfies these requirements.

Let's suppose that we want to install the middleware on the on board computer of a **road safety** VANET application with heavy traffic conditions. The only information that has to be selected is the vendor and the model of this particular on board device. With respect to the application restrictions, five extra services (Topology Control, QoS Management, Encryption, Fault Tolerance and Coordination) are selected, this kind of application demands a low delay (latency) as QoS requirement and the data delivery will be event based. At least, considering the heavy traffic conditions number of nodes will be very high, more than 100. If we enter this data our tool gets the initial features: OnBoard, TopologyControl, QoSManagement, Encryption, FaultTolerance, Coordination, Event, Latency, More100.

For the AAL application selected, we want to install the middleware in one of the wall-mounted nodes in charge of estimating the user position. We must inform that the device is a mote-like sensor with a distance sensing unit. The extra service required is the Coordination, no QoS restrictions are needed, and data delivery must be periodic data monitoring. This network has less than 10 nodes. The initial features will be: MoteLike, Static, Distance, Coordination, Periodic, Flat, Less10.

This initial set of selected high-level features is given as input to Hydra, which automatically calculates the minimum number of low-level features that must be added to the configuration in order to create a valid configuration.

4.2 Middleware Configuration

A complete and valid configuration is automatically calculated by Hydra, the same tool we have used for modeling the FM and its dependencies. Hydra accepts as input a set of selected high-level features. Then, using the FM, and the dependencies defined between high-level and low-level features at the domain engineering level, Hydra calculates, using a constraint solver (Choco), the minimum number of low-level features that must be added in order to create a valid configuration. This simplifies the configuration process, since the application developer: (1) does not need to deal with the FM directly, as the initial selection of features is automatically created by the tool described in the previous point; and (2) does not need to deal with low-level features, which are automatically inferred by Hydra, reducing the amount of information, and therefore the time, required for customizing the AmI middleware.

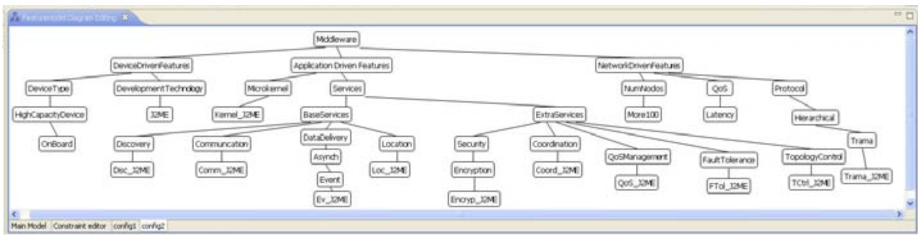


Figure 6: Configuration for Road Safety in the On Board computer in Hydra.

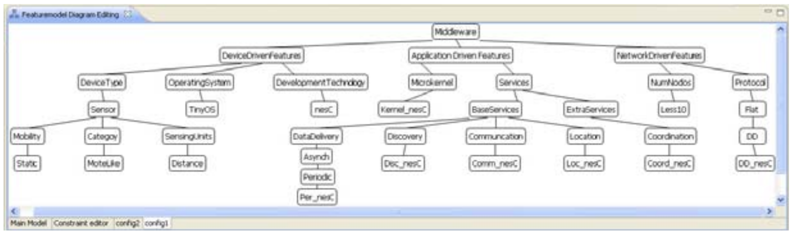


Figure 7: Configuration for Accidental Fall Report in the wall sensors in Hydra.

Figure 6 represents the particular configuration for the road safety VANET application in the on board computer. In order to obtain this configuration, Hydra has used some dependencies, for example between the on board device and the J2ME development technology. This development technology also influences the versions of the microkernel, services (base and selected extra services) and protocol. In this case the protocol chosen is Trama, because if More100 feature is selected, then the topology is hierarchical and Trama is the protocol for low delay requirements.

In Figure 7, the configuration for the accidental fall report in a wall mounted sensor node is shown. The dependencies used by Hydra will be the usage dependencies between the Mote Like and the TinyOS and between TinyOS and nesC development technology. So the version of the microkernel, services (the base services and coordination extra service) and the protocol will have to be nesC versions. Also the device will be static and will have a distance sensing unit. And finally as the Less10 feature is selected the topology will be flat and the network protocol will be Direct Diffusion (DD).

4.3 Middleware Product Architecture

The next step is to customize the architectural model created at the Domain Engineering level using the configuration calculated in the previous subsection. This task is automatically executed by VML4Arch. As already commented, a VML4Arch specification is *compiled*, by means of high-order transformations into a set of low-level model transformations in xTend, the model-to-model transformation language of the openArchitectureWare suite[6]. At the application engineering level, these generated model transformations are executed by the VML4Arch tool. These generated model transformations customize the domain engineering architectural model according to the selection of features, or configuration model, given as input. For the two examples previously described, the corresponding customized architectures are generated. In these architectures, all the plug-in components have been properly connected and the unnecessary components have been removed.

The next and final step is to generate implementation code from this customized architectural model. The microkernel must have a way to know which services and protocol will be used in order to manage them and compose them with the application component. We have implemented code generators that generate the part of the microkernel code related to the addition and initialization of the selected services and protocols. Figure 8 illustrates a piece of the generated code for a J2ME microkernel is shown. In order to run the application, the microkernel must create the components for the network protocol (Trama in this case, line 05) and the components for the base (lines 07-10) and the selected extra services (we only show Topology Control, line 11). All these components must have a reference of the microkernel and the component for the communication service must be connected with the selected protocol. Finally, the application component must be created, connected with the microkernel and the communication service and also the Display may be passed to the application (line 14). In this way, a complete package of the microkernel will be ready to be installed in the device. Components that compose the specific middleware, plus their instantiation, initialisation and configuration files are automatically obtained.

[6] http://www.openarchitectureware.org/

```
public class MkernelJ2ME extends MIDlet implements CommandListener{
 …
 public void runApplication() {
  //Create the components that realize the communication protocol selected
  protocols.TramaJ2ME prot = new protocols.TramaJ2ME();
  //Create the components that realize the base and the extra services
  services.communication.SOAPEngineJ2ME comm = new services.communication.SOAPEngineJ2ME(mkernel, prot);
  services.datadelivery.DataDeliveryJ2ME datadel = new services.datadelivery.DataDeliveryJ2ME(mkernel);
  services.discovery.DiscoveryJ2ME disc = new services.discovery.DiscoveryJ2ME(mkernel);
  services.location.LocationJ2ME loc = new  services.location.LocationJ2ME(mkernel);
  services.topologyControl.TopologyCtrlJ2ME top = new services.topologyControl.TopologyCtrlJ2ME(mkernel);
  …
  //Run the application
  application roadsafety = new application(display, "roadsafety", mkernel, comm);
```

Figure 8: Generated code for a J2ME microkernel.

5 Evaluation and Discussion

This article has presented a process for the automatic generation of customized middleware for AmI systems. This generated middleware contains exclusively the services required for a certain device, using a specific network and executing a concrete application. Therefore, memory usage is optimized, since no memory is used for useless services. Moreover, the configuration process is simplified because to customize the middleware only a reduced set of high-level features must be specified by the developers, and a larger set of configuration parameters is automatically inferred using a constraint solver provided by the Hydra tool. As we explained, Hydra tool is able to infer the minimum number of features to be added to a configuration to create a configuration that satisfies the constraints defined by the dependencies.

At a first glance, it might seem that to set up a SPL infrastructure may require great effort, and it is reasonable to think perhaps it might be cheaper to manually create these products. Although following an automatic SPL approach the initial cost for setting up the SPL infrastructure is actually higher, as soon as individual AmI middlewares start to be automatically generated, this initial effort becomes cost-effective. Moreover, we would like to point out that the use of innovative tools, such as VML, contributes to hide part of the complexity associated to the automation of the configuration process, decreasing the initial effort.

First of all, it should be taken into account that the definition of a family of AmI middleware(s) is a prerequisite for the generation of a customized middleware for a wide range of devices, networks and applications. Secondly, it should also be considered that a manual configuration process would imply a manual manipulation of the components implementing the selected middleware services, which is a complex and error prone task. We present quantitative evidence about the complexity of a manual configuration process by calculating the number of components that could be involved in the configuration of an AmI middleware.

In the FM of Figure 3 there are some features that can be selected independently of each other: **network protocols** (currently there are 5 *protocols (p)* available, p=5), **base services** (there are 4 *base services (bs*) available in Figure 3, *bs=4*), **extra services** (there are 7 extras services available in Figure 3, *es=7*) and the **microkernel** (1 *mk* avalaible). These features are affected by the OS and the development technologies (e.g. nesC, J2ME, Java), because a different version of each feature must be implemented for each OS/development technology included in the feature model. Let us consider three development technologies (*development technologies*, *dt*=3). It should be noticed this number could increase if we add to the feature model other existing development technologies, such as .Net or C or Python for Symbian. Now, let us calculate the number of components that correspond to these features. Furthermore, a feature is not implemented exclusively by just one component. For example, the communication service may be implemented by three components: an engine and two handlers. Other features are implemented by a higher number of components (e.g. encryption). We have calculated that the average number of components implementing a service is 4 (*components per service, cps=4*). Therefore, using the following formula, we calculate the total number of components that are included in the PLA. This number is estimated at 150 components.

$$[5p + (4bs + 7es) * 4cps + 1mk] * 3dt = 150c$$

This number of components is high and it may significantly grow if we would add one extra service or a new network protocol. For instance, considering a constant number of base services and protocols, if we increase the number of extra services (*es*), the number of components included in the PLA would increase depending on the number of development technologies (*dt*) as shown in Figure 9.a ([(22+4**es*)**dt*]=*c*; for *es*=10 and *dt*=8, then *c*=496), and as a function of *p* (protocols) and *dt* ([(*p*+45)**dt*]=*c*; for *p*=6 and *dt*=8, then *c*=408) (Figure 10.b).

So, in conclusion, it can be stated that the number of components the developer needs to deal with during a manual configuration process can be quite high. This makes a manual configuration process an error prone, tedious and time consuming task. In order to solve this issue, we automated this configuration process. Our automatic configuration process simplifies the configuration since the developer needs to specify a lower number of features than in the manual case, and s/he does not need to deal with component selection.

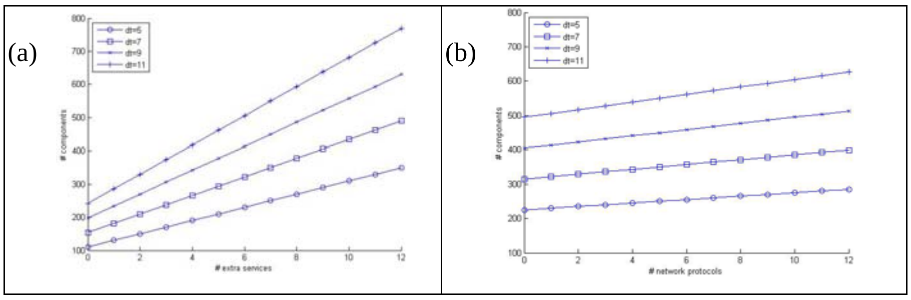


Figure 9: Number of components depending on the number of extra services (a) and networks protocols (b) with respect to the number of development technologies.

In our approach, all the possible configurations are taken into account. Nevertheless, we only have developed a significant subset of components, only those necessary to configure the middleware for different applications with good results. Table 1 shows the amount of information that the developer had to provide as input features to the process presented in this paper, for each application. The other columns show the devices, services, protocols, features, dependencies and components which are automatically managed by the automatic configuration process, transparently to the developer. For the accidental fall report shown throughout this paper, the developer only had to provide the number and type of devices (mote like sensors), the extra services required (coordination for the wall-mounted and security for the cameras) and the amount of network nodes (less than 10). The process automatically infers that the Direct Diffusion protocol must be selected. To calculate the low-level features required for customising the middleware for the static wall-mounted nodes, 10 dependencies were considered: the usage dependencies between the device type and OS, the concrete OS and development technology, the correct microkernel version, base services and extra services implementation, the number of nodes and the protocol version. For the other devices the dependencies taken into account were similar. Finally, for every implementation the number of components also varies depending on the development technology and the extra services selected. Table 1 is completed with data from other two applications of the VANET domain, one for ensuring road safety and other one for information dissemination. The customised middleware for these applications were obtained using a process very similar to the described for the Accidental Fall Report application. In all these cases, it was possible to automatically generate the customised middleware by providing a reduced amount of information about the selected features and executing our tool chain, i.e. the automatic constraint solver provided by Hydra, the model transformations generated by VML4Arch and the code generators. So, the configuration process is reduced to the specification of a reduced amount of information by the developers and pressing some buttons, so the customization task is drastically simplify and free of manual errors.

Configuration	**Input Feat.**	**Devices**	**Services**	**Protocol**	**Feat.**	**Dep.**	**Comp.**
AAL Mw (Accidental Fall Report)	7	3 wall sens.	Base, Coord	DD	12	10	12
	7	3 cameras	Base, Security	DD	12	10	13
	6	1 badge	Only Base	DD	11	9	10
VANET Mw (Road Safety)	9	On board	Base, TopCont, Security, QoS, FTolerance, Coord	TRAMA	16	14	40
	10	Signal Motes	Base, TopCont, Security, QoS, FTolerance, Coord	TRAMA	16	14	21
VANET Mw (Information Dissemination)	9	Motes	Base, DFusion, Coord, TopCont, Security	LEACH	15	13	18
	9	Sun Spots	Base, DFusion, Coord, TopCont, Security	LEACH	15	13	20
	7	Sink	Base, DFusion, Coord, Security	LEACH	14	12	17
	6	On board	Base, TopCont, Security	LEACH	13	11	27

Table 1: Results of the implementation.

Regarding the size of the middleware generated, we have a large middleware (with 40 medium size J2ME components) for the installation of the road safety application in the on board computer, and a really small middleware (with 12 tiny size nesC components) for the installation of the accidental fall report application in the wall-mounted sensors. One interesting result worth attention is that the number of components included in the customised middlewares is relatively small as compared to the total number of components included in the PLA, which were 150. This means that only a small set of components will be part of the customised middleware installations and the other components are simply removed in order to save resources.

The constraint solver provided by Hydra ensures that the configuration with the minimum number of features is the selected configuration. This demonstrates the power of our approach to manage variability of AmI middleware making feasible the generation of such totally different middleware configurations. With this, we have the minimum FM configuration, and VML4Arch automatically selects the minimum architecture using the feature mapping. In order to support this, as readers remember, the PLA was designed encapsulating the functionality that corresponds with optional features in separate components. The problem of selecting the correct component that must be part of a middleware instantiation, our process solves it automatically and generating configurations that satisfies the restrictions.

6 Related Works

There are several existing works that propose the use of middleware for AmI, embedded or mobile systems. One recently developed middleware is that presented by the Hydra Project, which is an ongoing project with similar goals to our proposal. Its objective is to create a widely deployed middleware for intelligent networked embedded systems that will allow producers to develop cost-effective and innovative embedded applications for new and already existing devices. However, until now, this project does not pay much attention to the configuration task, which in our opinion is very important in embedded devices due to the considerable resource restrictions. This restriction is particularly relevant in sensors, but in this work [Sarnovsky, 08], sensors are barely considered, whereas we believe them to be one of the most important devices in this kind of applications. Hydra middleware structure is fixed, it seems to configure some layers in the application level but always in a manual way, which is one of the disadvantages we have highlighted in the evaluation section. Furthermore, of the services provided, only the security layer is mentioned. While in their approach, it is fixed and orthogonal, we consider that this is not always necessary and that this service can be implemented just like any provided service.

On the other hand, recently has been published a systematic review [Morais, 09] of SPL applied to mobile middleware and some of our previous works [Fuentes, 08][Fuentes, 05][Fuentes, 06] has been selected for such review. This paper demonstrate that the proposal of applying SPL for middleware for this kind of system is a novel and interesting idea, since the goal of the authors is construct a family of middleware platforms for mobile devices using SPL. However, only few approaches have been found for the systematic review. Apart from our previous work, they have selected four different approaches. In [Lee, 07] a SPL process for configuration using FMs is presented. The goal and the function of the process is similar to ours, but it is a systematic process not automatic as ours is. Furthermore, the middleware and the process are only applied to the different roles that can play node sensors, so is a very small and particular case of our middleware that we also consider. Apel and Bohm [Apel, 05] have shared our object to design a lightweight, device-independent and customizable middleware, which is able to run on heterogeneous hardware and software. They use the mixin as SPL technology and we use the FM which allows us to better manage all the variability found in the AmI middlewares. Furthermore, in their proposal the customization of the middleware is the responsibility of the application programmer while in our proposal this process is completely automatic. Finally, in their work they focus only on the communication services, proposing to add services such as security or fault tolerance as future work, features which are already taken into account in our work. PLIMM [Zhang, 07] is a Product Line enabled Intelligent Mobile Middleware, in which Frame based on SPL techniques is applied to help manage the contexts. In PLIMM the context modelling is performed using ontologies and the authors propose an SPL process for the evolution of the ontology. However, they do not propose a process to configure and customize the middleware as we do. The variability of the services and devices that an AmI middleware has to deal with is not considered in this work. Finally, in [Krishna, 06] the authors present how context-specific techniques can be applied to the design of AmI middleware whose architecture is based on product lines, but they do not design a family of middleware using SPLs. They show a toolkit for automating context specific specializations, allowing already developed middleware to be customized; they do not propose a new middleware as we do. This process is not automatic as the code annotations that indicate the variations points have to be done by hand. They only automate the delivery of specialization but not the identification of specializations suitable for the PLA of a system. With our process however, given only the application restriction, device constraint and the network characteristics can automatically generate the minimum valid configuration that works properly.

7 Conclusions

Our initial hypothesis was that an SPL approach helps to deal with the complexity and heterogeneity of AmI systems. In this sense, we have presented a configuration process for an SPL AmI middleware to obtain customized middleware instantiations. This process is automatic and only a minimal set of high-level parameters needs to be specified by the developer of the applications. Using some tools (Hydra, VML4Arch and code generators) together with these high-level parameters the particular, ready to install configurations will be automatically generated by our process.

AmI applications will benefit from such a highly-optimized and custom middleware, which will offer appropriate services consistent with device configuration and resource constraints. In this way, final applications will only have to model the application specific functionality, thereby removing the code dependence of the hardware and software of devices and network, what will be incorporated by invoking the correct middleware services through high level interfaces.

In the discussion and evaluation section we have shown the complexity of configuring an AmI middleware due to the high number of components involved. We have also shown how the automatic configuration process presented in this paper was successfully applied to different case studies, alleviating the engineer's task of managing such diverse features, components and the mappings between them.

References

[Akyildiz, 04] Akyildiz, I., Kasimoglu, I.: Wireless Sensor and Actor Networks: Research Challenges, Ad Hoc Networks Journal, Elsevier, 2(4), pp. 351-367, October 2004.

[Apel, 05] Apel, S., Bohm, K.: Towards the Development of Ubiquitous Middleware Product Lines, LNCS Software Engineering and Middleware, Vol. 3437, March 2005.

[Bravo, 05]Bravo, J. et al.: Ubiquitous Computing in the Classroom: An Approach through Identification Process. Journal of Universal Computer, 11(9), pp. 1494-1504. 2005

[Bravo, 06] Bravo, J. et al.: Visualization Services in a Conference Context: An Approach by RFID Technology. Journal of Universal Computer, 12(3), pp. 270 – 283, 2006.

[Delicato, 09] Delicato, F., Fuentes, L., Gámez, N., Pires, P.: Variabilities of Wireless and Actuators Sensor Network Middleware for AAL, LNCS 5518, pp. 851-858, 2009.

[Dikaiakos, 07] Dikaiakos, M., Florides, A., Nadeem, T., Iftode L.: Location-aware Services over Vehicular Ad-Hoc Networks using Car-to-Car Communication, IEEE Journal on Selected Areas In Communications, 25(8), 2007.

[Fuentes, 05] Fuentes, L., Jimenez, D.: An Aspect-Oriented Ambient Intelligence Middleware Platform, In Proc. 3rd Int. Work. on Middleware for Pervasive and Ad-Hoc Computing, 2005.

[Fuentes, 06] L. Fuentes, and D. Jiménez, "Combining Components, Aspects, Domain Specific Languages and Product lines for Ambient Intelligent Application Development", Proc. of International Conference on Pervasive Computing, Ireland, May 2006.

[Fuentes, 08] Fuentes, L., Gámez, N.: A Feature Model of an Aspect-Oriented Middleware Family for Pervasive Systems, In Proc. AOSD Workshop on Next Generation Aspect Oriented Middleware, pp. 11-16, Belgium, April 2008.

[Fuentes, 09] Fuentes, L., Gámez, N., Sánchez, P.: Managing Variability of Ambient Intelligence Middleware, Int. Journal of Ambient Computing and Intelligence, 1(1), pp. 64-74, March 2009.

[Keshavarz, 06] Keshavarz, A., Tabar, A., Aghajan, A.: Distributed Vision-Based Reasoning for Smart Home Care, In Proc. SenSys Workshop on Distributed Smart Cameras, USA, 2006.

[Krishna, 06] Krishna, A., Gokhale, A., Schmidt, D.: Context-Specific Middleware Specialization Techniques for Optimizing Software Product-Line Architectures, ACM SIGOPS Operating Systems Review, Vol. 40, No. 4, 2006, pp. 205-218.

[Lee, 02] Lee, K., Kang, K., Lee, J.: Concepts and guidelines of feature modeling for product line software engineering, LNCS 2319, pp.62–77, 2002.

[Lee, 07] Lee, W., Kang S., Lee, D.: Product Line Approach to Role-Based Middleware Development for Ubiquitous Sensor Network, In Proc. 7th IEEE Int. Conf. on Computer and Information Technology, pp. 1032-1037, Japan, October, 2007

[Loughran, 08] Loughran, N. et al.: Language Support for Managing Variability in Architectural Models, LNCS Software Composition, Vol. 49, pp 36-51, 2008.

[Morais, 09] Morais, Y., Burity T., Elias, G.: A Systematic Review of Software Product Lines Applied to Mobile Middleware, In Proc. 6th Int. Conf. on Information Technology: New Generations, USA, April, 2009.

[Pohl, 05] Pohl, K., Böckle, G., Linden, F.: Software Product Line Engineering – Foundations, Principles, and Technique, Springer, Berlin, Heidelberg, New York, Aug. 2005.

[Sarnovsky, 08] Sarnovsky, M. et al.: First Demonstrator of HYDRA Middleware Architecture for Building Automation, In Proc. Znalosti 2008, pp. 204-214, Feb. 2008.

[Rajendran, 03] Rajendran, V. et al.: Energy-efficient collision-free medium access control for wireless sensor networks, In Proc. Int. Conf. on AmI Networked Sensor Systems, USA, 2003.

[Wang, 08] Wang, M. et al.: Middleware for wireless sensor networks: A survey, Journal of Computer Science and Technology. Vol. 23, No. 3, pp. 305-326, May 2008.

[Zhang, 07] Zhang, W. et al.: Product Line Enabled Intelligent Mobile Middleware, In Proc. 12th IEEE Int. Conf. on Engineering Complex Computer Systems, pp. 148-160, 2007.