A GENERIC FRAMEWORK FOR CONTEXT AWARE AND ADAPTATION BEHAVIOUR OF RECONFIGURABLE SYSTEMS

Eiman Mohyeldin [1], Michael Fahrmair [2], Wassiou Sitou [2], Bernd Spanfelner[2]

[1] Siemens AG, Sank Martin str. 76, D- 81541, Munich, Germany, eiman.mohyeldin@icn.siemens.de

[2] Munich University of Technology, Munich, Germany, {fahrmair | sitou | spanfeln}@in.tum.de

Abstract - It is clear that the reconfiguration of mobile SDR terminals requires complex interactions between the mobile terminal and the network or application server entities. To enable mobile devices to utilize different radio access technologies and communication protocols depending on the requested application QoS and to enable development of context aware application that can adapt their level of functionality to dynamic radio resource restrictions like available bandwidth, delay and link interruptions, a generic software framework for adaptation and reconfiguration is necessary. In this paper we describe a generic technical and methodical framework for designing context awareness and adaptation behaviour that is based on formal methods, thus allowing for a sophisticated engineering approach in designing and implementing complex context aware adaptive systems.

This framework supports the development of customized middleware, reconfigurable protocol stacks and adaptive application services for the three main phases of reconfiguration: profile and context management, Adaptation decisions, technical deployment.

Keywords - Context Awareness, Software Defined Radio, Reconfigurable Terminals

I. INTRODUCTION

The development of reconfigurable radio is accompanied by a corresponding development of the radio network infrastructure. Whatever future mobile network infrastructures will look like in detail, one thing can be taken for granted: Future mobile networks will be much more heterogeneous, offering different radio access technologies, service quality levels and multi-link connection options at the same time. Therefore, it is important to make clever choices regarding usage of these options in order to maximize the benefits and optimize the cost-performance ratio for all parties involved. In order to be able to make intelligent decisions at least two ingredients are required:

1. An entity that has sufficient cognitive abilities and background knowledge at its disposal to be able to make such clever decisions has to be available. Ideally, this entity is able to learn from experience, in particular with respect to user preferences and user behaviour, but also regarding recurring situations in the mobile network infrastructure and the surrounding environment.
2. A broad range of information characterizing the current situation of the mobile terminal is required as a basis for adaptation of the communication modalities and service usage patterns to the state of the environment. Part of this information may be provided by the mobile network infrastructure, but information from other sources including sensors, information systems, databases or other mobile devices can also be involved.

There are already several different architectures and frameworks supporting developed context aware HW/SW-systems such as [1] and [2]. However, the important aspect of designing contexts and adaptation logic itself is typically overlooked. Moreover the context model and adaptation decision logic are usually static and hard-coded into the adaptable entities that are therefore suitable only for implementing relatively small scenarios within predictable environments. In view of 4G and ubiquitous systems, though, this approach seems inadequate, since the environment in which a system's functionality can be executed and the context parameters that may influence it will not be predictable a priori at the time that the function is being developed [3].

In [4] a generic software framework for adaptation and reconfiguration is described, where a mathematical founded approach for designing flexible adaptation is introduced.

In this paper we now present a technical framework design that supports context awareness and reconfiguration concepts, This generic, re-usable mechanism offers runtime customizable context criteria and adaptation algorithms and therefore helps to circumvent the AI frame problem [5] that could otherwise generate spontaneous unexpected behaviour in long-living ad-hoc reconfigurable protocol stacks or adaptive application services [6], [7]. The paper is organized as follows: in section II, an overview of the framework is given. The implementations of the context elements are described in section III followed by the implementation of the abstract classes in section IV. The description of the syntactical and semantical types is given in section V. Finally the application subsystem is described in section VI.

II. FRAMEWORK OVERVIEW

In this section a short overview about the overall MIDAS (MIddleware DemonstrAtor Server) framework is given. The MIDAS framework consists of the following elements:

- A set of components (technical implementation)

- A set of interfaces (API)
- A reference architecture
- A framework infrastructure

Components, interfaces and architecture together form a basic framework for context awareness and adaptive applications. Hence framework is a certain reference architecture proposed and implemented for most parts of functionality that can be reused between different context aware applications. Application specific details however are only implemented as empty placeholders (configuration files, interfaces, abstract classes). To develop a certain application these parts need to be filled with application specific logic. The collaboration between framework- and application specific components as well as application specific components among themselves is however controlled and defined by the framework. This in general differentiates frameworks from class libraries.

Nevertheless a context aware application developed under the MIDAS framework still can provide its own implementation of framework components. Therefore application can have its own private context server built into its core system which can be suitable for non distributed HW/SW setups, for example if the application has its own private device with enough CPU power to run on and moreover has no components involved that are shared among other applications and are adaptable itself. This approach can be very ineffective in situations where several context aware applications share the same hardware infrastructure.

However using a shared framework infrastructure provides for much more flexibility by sharing for example the context server (CS) among several applications. The private framework scenario has the advantage of an easy installation and management. All necessary components are installed as part of the core system of the application with one installer. This method is suitable if there are additionally devices required for the application (like a terminal which needs a separate installer). But as soon as such external resources are to be shared among several adaptive applications, there need to be a higher instance to synchronize adaptations of applications that are connected with each other by using the same external resource. In these cases it would be advantageous to have the framework functionality installed separately from one specific application.

The MIDAS framework provides support for both scenarios. The framework infrastructure is a special context aware application and can be installed and run separately from any specific applications (comparable to a middleware or application server) and contains context adaptive logic to manage and share resources between several context aware applications.

The framework functionality can be a shared infrastructure or an application specific one, the minimal required subsystems are:

- At least one CS that can hold many virtual context spaces i.e. acts as a virtual private server for each application. Context spaces of different applications are isolated from each other. They can be explicitly coupled. More than one CS can be used in wide area networks to make communication more efficient. For example there can be private CSs on terminals to cache/proxy context signalling for low or interrupted bandwidth situations.
- At least one terminal (T) that can be used as a default fallback user interface, e.g. while installing or bootstrapping a new context adaptive application.
- Several reconfigurable devices (Nodes) that allow for core system reconfiguration. These devices have the ability to instantiate and de-instantiate sets of components on the device. For the framework in infrastructure mode it is usually possible to download additional (mostly application specific) component sets. Currently, the framework knows three types of such device specific reconfigurable containers: The IIS node (optimized to run (web-) server services), the Windows XP computational node (for high performance terminal devices like notebooks) and finally the Windows CE.net Compact Framework Node (for medium performance terminal devices like Smart phones or Pocket PCs).

III. IMPLEMENTATION OF CONTEXT ELEMENTS

A context element based on its formal definition is strictly speaking nothing more than an intermediate storage for arbitrary information, e.g. a user, terminal, network or service profile [8], [9] and [10]. A context element therefore can be implemented as a container for a self-describing data-format like for example a XML document.

To allow for calibration (adaptation of adaptation) the technical framework specifies a context element as a total reconfigurable service. This means a component that is later bound to fulfil the context element service described in the abstract adaptation model “k-model [4]” is always called using a special proxy component. This proxy is[1] connected to all possible component bindings (implementations) of the service it represents and therefore can adapt (reconfigure) between all of them at runtime.

The extension of the framework compared to the formal mathematical specification is the possibility to multiplex several similar channels into one channel with an id value that allows for separation of the multiplexed content. The reason behind this is that implementing one context element with one component could be inefficient. The optimization of binding several context element services to the same component ,e.g. a context server is perfectly valid approach,

[1] Or can establish a connection at runtime, e.g. by downloading a software module and starting the component contained in it.

since by definition a service can be fulfilled by one or more components as well as one component can fulfil more than one service at the same time as long as long as no channels are shared. Multiplexing several (virtual) channels into one (physical) channel fulfils the condition. The technical realization of a context element is presented in Figure 1.

IV. IMPLEMENTATION OF ABSRTACT CLASSES (SENSORS, INTERPRETERS, ACTUATORS)

Sensors, interpreters as first introduced in [1] and actuators introduced in [11], are realized as abstract classes in the framework., the abstract classes are implemented depending on the concrete interpretation e.g. a rule, a neural network etc. or actuation that is a certain device, telecommunication service/layer, application etc. functionality. To allow for reconfigurable adaptation, components derived from these classes are used in form of a reconfigurable service represented by a transparent service proxy component. This proxy component is part of the context adaptation subsystem but not part of the adaptable/reconfigurable subsystem. The functionality itself however, i.e. the component bound to the service proxy at runtime, can be either part of the system core or the technical system environment.

The main goal of reconfigurable services is the decoupling of communication between service provider and consumer components using proxy components. These proxy components represent a logical service independently of its technical implementation, since they can reroute messages between consumer and changing provider implementations transparently at runtime [12], [13] and [14]). From an adaptation point of view service proxies therefore are just entities of adaptation logic that can show or hide certain behavioural aspects of the core system or the environment respectively.

To provide a simple model of the adaptation subsystem the number of component types that can be adapted and modelled using a k-model is restricted to four roles, context elements, sensors, interpreters and actuators.

The classification in context elements, sensors, interpreters and actuators is merely the result of a trade-off between easy and intuitive modelling and the technical possibilities to automatically implement any changes to the model at runtime by a special actuator service (*activator*) that works on a context containing a k-model.

Consequently the activator manages four sets of service proxies. Depending on the current k-model the activator has read from the context and instantiate or de-instantiate new proxies or reconfigure their implementing components as well as their interdependencies.

The proxies, all four, inherit from a generic service-proxy interface `IServiceProxy` that contains their reconfiguration possibilities. The reconfiguration possibility is expressed with the `SetBinding` Method. With this method the actual provider for service implementation is chosen. Since individual service-proxies should be used transparently by any service consumer, they hold both a generalization of their respective `IContext`, `ISensor`, `IInterpreter` and `IActuator` interfaces as well as a reference to the actual provider component of the same interface.

The service providers need to implement a reconfiguration interface that is necessary in order to fulfil the condition of technical component offering several services. While this is not a very difficult requirement, if it would only span over services with different types of interfaces, it becomes complicated if one wants to allow for service fulfilment of several services with the same interface. The latter one however is a technical necessity. Figure 2 depicted the abstract model of the framework.

V. SYNTACTICAL AND SEMANTICAL TYPES DESCRIPTIONS

All k-model elements have a syntactical (`SyType`) and semantical (`smType`) description of the service they provide. Both types are used to match a suitable technical component that could implement the service. In other words `syType` and `smType` specify which components the service representing proxy could possibly route messages to.

`SyType` describes any higher level protocol the service implementing component should understand using the standardized low level interfaces of sensors, interpreters, actuators and contexts including at least the data format accepted. Moreover it could contain any other technical information needed to reduce the number of matching components e.g. QoS parameters, billing information etc. `SyType` should contain information needed by the activator to contact and bind possible candidates or even the reference to a single component instance insuring that only one specific component will match the description.

`SmType` in contrast describes the meaning or usage intention of a certain component instance besides its technical characteristics. Usually this can be used to distinguish between several instances of technical identical components. For example there could be several identical temperature sensors or terminals connected to a single system. However they can have different meanings regarding to the context like outside temperature, inside temperature, kitchen, terminal or entrance terminal. In order for the activator to distinguish which component instance should be bound to e.g. a sensor that feeds a context element with a meaning of “outside temperature” and can be expressed by the context element’s `smType`. A syntactical description is insufficient in this case since it could match more than one component instance. Instead one of the available sensors needs to have been marked with a meaning of “outside temperature” as well. Semantical marking is part of the context and one of the main tasks of calibration. The

`smType` information is obviously not part of the component instance except with two potential exceptions:

- The identity of the component instance could indirectly describe its `SmType`
- A component instance could be configured by means external to the context aware system (e.g. if it is connected to a physical device by pressing certain buttons to choose a predefined meaning role).

Both potential exceptions can be resolved to the idea that the context holds the meaning of a service instance. An identity `SmType` is merely a description by the meaning of the instance's own existence. Any external configuration possibilities can be modelled by a further sensor that generates semantical information about the component instance in question. This information is then available in the context or could even be used to change the k-model.

There could be possibly further discussion about semantic meanings being external i.e. a context, especially in case of differentiation between syntactical and semantical description reside on the views that a syntactical type should be the interface and the semantical type the behaviour of a component. However the real "meaning" of a component is only generated by observation in a larger correlation with other entities and can not be grounded in a symbolic description of the component instance alone. An indication of this fact would be a component instance that, though it has a constant behaviour can have different meanings in two different observation contexts. For example the same camera instance that shows the entrance of a building for one observer could show in the same picture a certain street segment for another observer or the weather conditions for a third observer, the water level of the nearby river and so on. Another example would be a temperature sensor on the outside of a package. It can mean the outside temperature (compared to the packages inside temperature) but also at the same time could have a meaning of inside temperature for the owner of a storage house the package is currently stored in.

VI. APPLICATION SUBSYSTEM

An application subsystem built with the MIDAS framework, illustrated in Figure 3, consists of a single system bootstrapper (System Seed SYSE). Each application usually has its own SYSE that can be installed and uninstalled separately. A SYSE Package typically includes:

1.) A boot sensor,

2.) An optional boot actuator and

3.) The applications core system.

Usually the application core system initialized by the SYSE contains only a boot sensor specification and an administration component (AC) implementing that boot sensor.

The administration component, usually a GUI, connects to the application origin server and from there downloads or updates a k-model XML file for the application and any necessary core system components that run in the domain of the application.

In the following the initializing steps of the necessary framework components are described:

1.) The first step is initializing the framework. This usually means doing an auto-discovery procedure to detect a public context server in the same subnet as the SYSE. Optionally the AC can also initialize its own context server within its own core system.

2.) After finding a proper CS, the next step is to initialize application specific context space e.g. virtual context server. This is because of the regulation that context elements are only valid within one application domain. Cross domain sharing of context information is only allowed using a sensor/actuator coupling to avoid implicit shifts of semantic types. Of course there can be CS that do not allow for creation of context spaces e.g. if the CS is an application private one. In these cases it is only possible to create one single virtual CS with a specific name.

3.) Creating virtual context spaces usually is part of the CS subsystem k-model adaptation. This means a CS usually runs its own adaptation regarding itself as the core system and has proper sensors to detect incoming requests to create a virtual context space as well as actuators to do the necessary management. Usually creating a virtual CS also contains (within the request) a boot k-model to set up the requested context space. Usually this is a boot sensor from our SYSE that transfers the application k-model and uses the built in k-model-actuator (activator) to boot up the application adaptation.

There is however a second alternative. Creating a virtual CS could involve creating a standard boot k-model in this context space. This standard k-model needs to allow for automatic sensor discovery, in order to deliver an application k-model. This way the applications boot actuator can connect to the CS, is automatically connected into the boot-k-model of the given context space and can deliver its new k-model (usually an application specific boot-k-model first). This auto-discovery method however has some security disadvantages compared to the standard boot procedure where the boot sensor is part of the CS core system and more trustworthy than an external bound boot sensor. This is even truer for binding an external activator.

4.) So either the application specific boot-k-model was sent when creating the virtual CS or it was delivered afterwards by sensor auto-discovery. Either way the k-model sent is usually a boot-k-model that manages static resources allocation within the framework. In highly dynamic/long running environments resource allocation can be part of the application k-model itself. For simplicity a short demonstration application with a hardware environment is assumed, which regarded as static during the application usage but dynamic in between two application runs.

During the boot process the application SYSE uses the CS to detect any necessary hardware resources that are needed for application execution. These resources can be either bound as external components directly to the adaptation subsystem of the application or be bound into the application core system by a core system reconfiguration. However external resources usually require a registration process to couple their adaptation into the application context adaptation. Terminals for example can download application specific software components and initialize them by reconfiguring their core system. The registration makes sure the terminal creates an application specific component space. The SYSE can then start to check and update software components that are required to run on the terminal for the application.

5.) After this boot adaptations have finished, the k-model in the applications virtual context space can again be updated to the final model.

6.) The final model is used during application execution. Many applications however have a k-model that, besides the application adaptations, also contains an administration actuator and or sensor. Both specifications are usually bound to the administration component in the applications SYSE package. These admin sensors and actuators can allow for monitoring or outside control of the application. This control can also contain calibration capabilities that allow for modification of the current running k-model during runtime.

7.) Ending the application, the SYSE terminates the application but usually leaves its context information intact for persistency.

Uninstalling the SYSE however needs to remove all locked resources. This usually means deleting all virtual context spaces and their contents as well as component download spaces in terminals or similar external resources. Since however removal cannot be guaranteed each external resource of course has own clean up mechanisms.

VII. CONCLUSIONS AND OUTLLOK

Based on a clear scope of reconfiguration as a form of technical solution to achieve ubiquity, the framework presented in this paper is a generic approach to support all kinds of adaptation in reconfigurable systems. With its support for calibration even the adaptation logic itself can reconfigured to avoid typical framing problems like spontaneous unexpected behaviour that can emerge especially in long running systems or consumer systems with a large number of users with different and changing expectations toward a semi-intelligent system [6].

The process of changing context or adaptive behaviour depending on the current context is not necessarily self-contained since the information can be produced by sensors and interpreters outside of the system. This way the system's context and adaptation behaviour can also be modified by means outside the scope and knowledge of the system architect during design time of the system, thus providing enough flexibility to integrate further modular techniques (including human customization) to address the frame problem.

Since the framework supports generic adaptation such a personalization mechanism is not limited to modifying a simple set of rules. Instead it is possible to rearrange abstract function roles (sensors, interpreters, context and actuators) that can hide any kind of technical realization. Therefore it is even possible to mix rule based decisions with fuzzy logic components or to use neuronal networks to customize adaptation behaviour.

Further research efforts are directed towards applying and integrating various concrete approaches from artificial intelligence research for changing adaptive behaviour and relevance. Still open is a semantically well defined process to design adaptive systems, while concentrating on the adaptive behaviour rather than discussing implementation details. Our ongoing research suggests an approach that involves facing possible abstract situations with objectives and afterwards designing an intermediary connection between the two. This is done in form of formal conditions and action specifications to reach the planned situation. Both conditions and actions are then expressed as context interpreters inside the context model itself.

References

[1] Anind K. Dey, "Providing Architectural Support for Building Context-Aware Applications PhD thesis", College of Computing, Georgia Institute of Technology, December 2000.

[2] A. Chan, S. Chuang. MobiPADS "A Reflective Middleware for Context-Aware Mobile Computing", IEEE Transactions on Software Engineering, Vol. 29, No. 12, December 2003

[3] Christopher Lueg, "Operationalizing Context in Context-Aware Artifacts: Benefits and Pitfalls", Informing Science Vol. 5 No. 2/2002.

[4] E. Mohyeldin, M. Dillinger, M. Fahrmair, W. Sitou, P. Dornbusch, "A Generic Framework for Negotiations and Trading in Context Aware Radio"

SDR Technical Conference, Phoenix Arizona USA, November 2004.

[5] Daniel C. Dennet, ”Cognitive wheels: The frame problem of ai”, C. Hookway, editor, Minds, machines, and evolution, pages 129--151. Cambridge University Press, Cambridge, 1984.

[6] M. Dillinger, E. Mohyeldin, J. Luo, M. Fahrmair, P. Dornbusch, E. Schulz, “Cross Layer and End to End Reconfiguration Management”, WWRF11- Services and Applications Roadmaps- Invigorating the Visions, Oslo/Norway , 2004.

[7] E. Mohyeldin, E. Schulz, M. Dillinger, M. Fahrmair, P. Dornbusch, “Dynamic Reconfiguration of Wireless Middleware”, IST Mobile & Wireless Communications Summit 2004, Lyon/France, June 2004.

[8] M. Dillinger, K. Madani, N. Alonistioti (Editors). “Software Defined Radio. Architectures, Systems and Functions”, Wiley Series in Software Radio, John Wiley and Sons, Ltd, 2003

[9] P. Dornbusch, M. Fahrmair, E. Mohyeldin, M. Dillinger , “Communication Profiles and Active Attributes”, IADIS International Conference WWW/Internet, Algarve/Portugal, P. Isaias, N. Karmakar (ed.), IADIS Press, 2003.

[10] M. Dillinger , E. Mohyeldin , J. Luo, P. Dornbusch , M. Fahrmair, C. Salzmann, “Structure and Management of SDR System Profiles” SDR Technical Conference, San Diego USA, 2002

[11] N. Houssos, A. Alonistioti, L. Merkakos, M. Dillinger , E. Mohyeldin, M. Fahrmair, M. Schoenmakers ”Advanced Adaptability and Profile Management Framework for the Support of Flexible Mobile Service Provisioning“, IEEE Wireless Communication Magazine, August 2003.

[12] M. Fahrmair, C. Salzmann, and M. Schoenmakers, ”A reflection based tool for observing jini services”, Reflection and Software Engineering, number 1826 in LNCS. Springer Verlag, 2000

[13] M. Fahrmair, C. Salzmann, and M. Schoenmakers ”Carp@ - managing dynamic jini systems”, Proceedings of Middleware 2000

[14] M. Breitling, M. Fahrmair, C. Salzmann, M. Schoenmakers,”Carp@ - Managing Dynamic Distributed Jini Systems”, OOPSLA'99 Workshop on Reflection and Software Engineering, pages 173--184, 1999

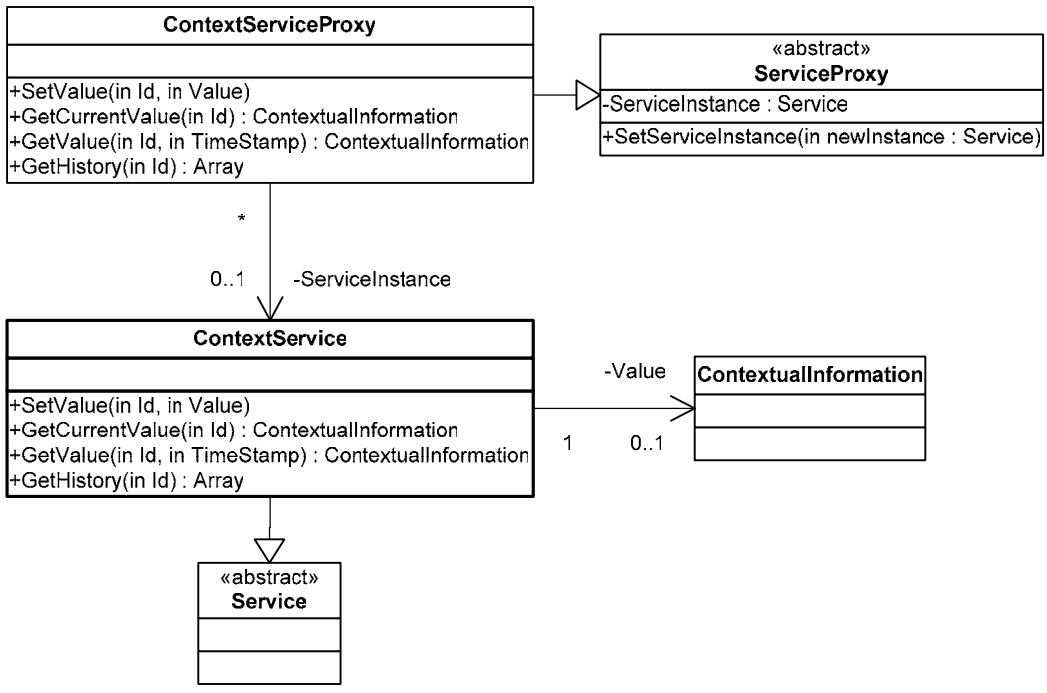

Figure 1: Technical Realization of a Context Element

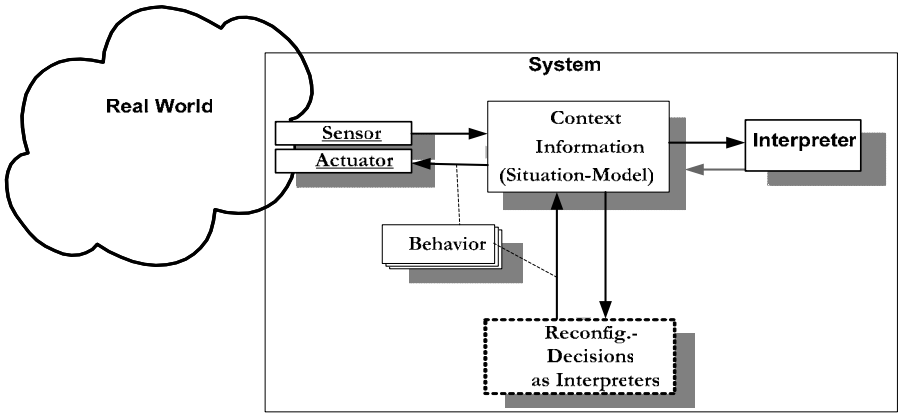

Figure 2: abstract model of the framework

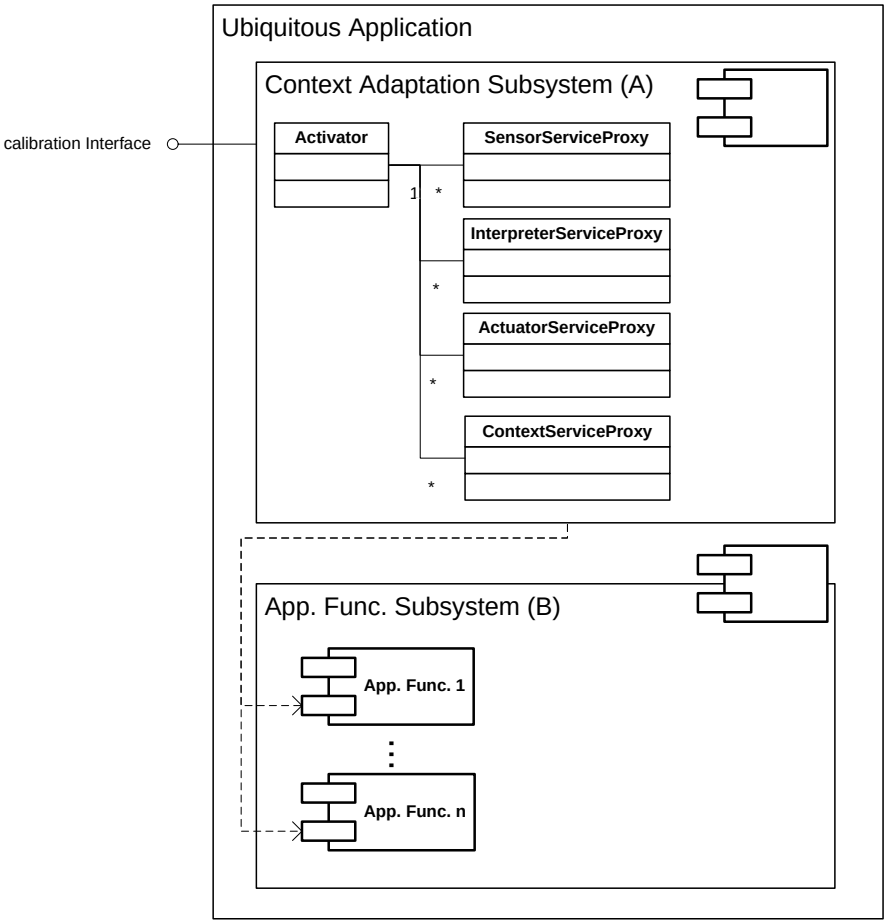

Figure 3: Separation of Adaptation Control and Application Core