# Survey of Modeling and Engineering Aspects of Self-Adapting & Self-Optimizing Systems

V. Bauer, M. Broy, M. Irlbeck, C. Leuxner, M. Spichkova, M. Dahlweid, T. Santen

TUM-I1324

# Survey of Modeling and Engineering Aspects of Self-Adapting & Self-Optimizing Systems

V. Bauer, M. Broy, M. Irlbeck, C. Leuxner, M. Spichkova

*Technische Universität München, Department of Informatics*
*Boltzmannstr. 3, 85748 Garching, Germany*
{*bauerv, broy, irlbeck, leuxner, spichkov*}*@in.tum.de*

M. Dahlweid, T. Santen

*Microsoft Research Advanced Technology Labs Europe*
*Ritterstrasse 23, 52072 Aachen, Germany*
{*thomas.santen,markus.dahlweid*}*@microsoft.com*

**Abstract**

In this paper, we give a concise survey of concepts, architectural frameworks, and design methodologies that are relevant in the context of self-adapting and self-optimizing systems. These topics are motivated by open research questions and some current requirements of adaptive systems from an industry point of view.

*Keywords:* adaptation, adaptive behavior, architecture, methodology, autonomic computing, context awareness

## 1 Introduction

We consider adaptive systems as reactive, software-intensive systems, which expose more flexible system behaviors than conventional software systems. Adaptive systems typically use information about the system environment to adapt themselves to certain usage situations. This adaptation can have many forms.

In our daily life, many systems adapt their behavior in reaction to a given usage situation. A prominent example of such systems are smart phones and their software applications. Using the possibilities of built-in sensors, such as location or orientation sensors, many smart phone applications are able to provide a context-aware behavior. Weather applications show the current weather depending on the user's current location, navigation applications guide users to a given destination, and graphical user interfaces automatically adjust their screen orientation according to the orientation of the phone, allowing them to render the content in an upright or landscape format.

But not only conventional sensors allow applications to adapt themselves to a changing environment. Communication interfaces such as Wi-Fi or Bluetooth with their ever-changing context in terms of connections to other devices force applications to adapt their behavior or internal state: the establishment, loss and reestablishment of connections to other devices has an impact on the behavior of applications that use these communication interfaces.

This interactive system behavior can foster many positive effects. Systems with the ability to adapt themselves provide a richer set of features and promise to release users from certain interactions with the system by reacting to a given context. Therefore, adaptability aspects play a major role while designing systems. However, adaptivity capabilities have to be analyzed carefully, to avoid unwanted incidences as presented by the following example.

Most of modern cell phones provide the possibility to connect to a headset through a Bluetooth interface. The user pairs the headset to the phone, ensuring that the cell phone will always connect to the right headset. After that, the phone is able to connect to the headset automatically: Whenever the headset is within reach of the phone, the user has just to turn it on and the phone will forward all calls to the headset.

Connecting to the phone via Bluetooth, however, is a feature not only offered by headsets. Notebooks can also be connected through the same interface, enabling useful functionality such as synchronization of contacts, appointments or music files. However, the user usually pairs the notebook to the phone with other intentions than with the headset, and sometimes is unaware in which way the phone will adapt and which features are activated. So, when receiving a call near the notebook with activated Bluetooth can lead to unwanted behavior, e.g. calls being forwarded to the notebook.

This scenario shows the benefits as well as the limits and drawbacks of adaptive behavior and context awareness. Exploring in which way systems are able to adapt themselves, opens an interesting field of research. However, in our understanding it has to be clearly defined what is meant by adaptive behavior, in which way it is emerging and what kind of properties are connected to it.

The second motivating example originates from the context of software development. Modern Integrated Development Environments (IDEs) such as Visual Studio or Eclipse provide a plethora of functions to the developer. However, only a small subset of functions is needed for specific development tasks, e.g. coding, testing, debugging. Finding the appropriate functions for a task can be time consuming. Also, the developer might be unaware of useful functions and, thus, can not benefit from them. Some IDEs already group a certain set of functions into specific profiles, such as debugging. However, the developer still needs to manually switch between those. Also, IDEs currently do not recognize whether a developer is currently stuck with a problem, or employing solution strategies, for which more appropriate functions would be available. It would therefore be beneficial to provide an adaptive IDE, presenting the adequate functions for the task at hand. At the same time, care needs to be taken to not confuse the user by continuously changing the interface they are working with.

In this context, research has been conducted by Roehm and Maalej [25] on automatically detecting the activities of developers based on their actions. The activities are grouped into an iterative model containing the steps in development work, such as searching for the cause of a problem or for a solution, applying the solution, and testing it. From these activities, they derive what the developer is trying to do and whether they are encountering problems. This research could help to enable IDEs to adapt to the current development task and thus support the developer more effectively.

## Research Dimensions

The above examples illustrate a number of important aspects that research on self-adaptive systems must address. In the context of a Dagstuhl seminar, the software engineering research community has systematically gathered further aspects and clustered them into four *views*: modeling dimensions, engineering, assurances, and requirements [6]. Figure 1 captures these views and details the aspects comprised in them:

- *Modeling dimensions* addresses aspects of how goals of the systems can be captured and pursued under changing, possibly suboptimal, environmental conditions. It furthermore raises questions on how change is perceived by the system and which mechanisms are required to adapt the system to change. Finally, modeling is concerned with the effects that a self-adaptation of the system has on the environment.
- *Engineering* is concerned with the technical realization of self-adaptive systems. It therefore comprises the aspects that need to be addressed to design, implement, and deploy a concrete self-adaptive system. This affects all artifacts of the standard software development cycle, which need to be tailored to meet the needs introduced by adaptivity.
- *Requirements* is considered a seperate view, as many questions are raised by self-adaptivity: foremost, new languages are required to capture requirements. Then, it is not yet established how uncertainty and incompleteness of requirement specifications can be mitigated in the face of unpredictable environment conditions. From this follows the need for reflection of re-

quirements during the system's runtime, which raises issues of refinement, traceability, and architectural mapping.

- *Assurances* addresses the behavioural guarantees required for self-adaptive systems. On a technical level, these might be addressed by model-driven development and agile runtime assurance. Preliminary for these techiques, dynamic identification of changing requirements must be enabled. Furthermore, self-adaptation raises fundamental questions of liability and social responsibilities with respect to safety and security.

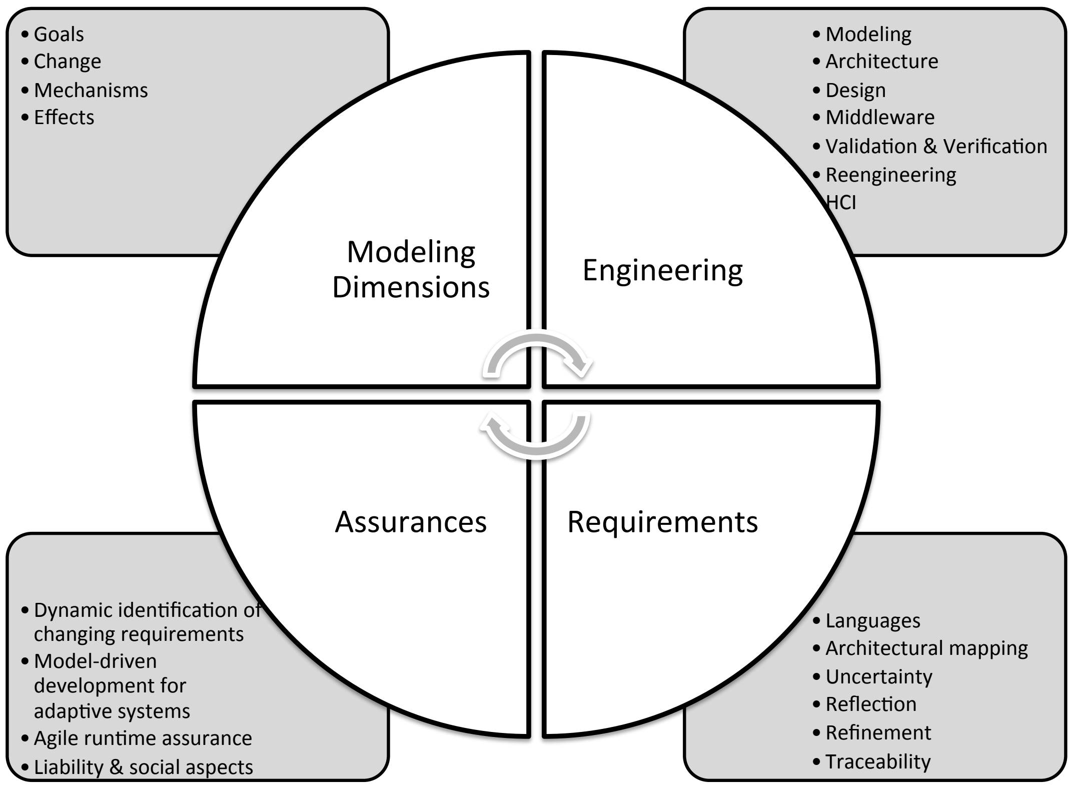

Fig. 1. Research fields for adaptive systems, according to [6].

## Problem Statement

Adaptivity of software systems is expected to provide great benefits. However, many different aspects need to be investigated carefully to avoid unwanted behavior and to mitigate limitations of these systems. Adaptive systems therefore give rise to a plethora of challenging research questions, ranging from the definition of adaptive behavior and its goals, over modeling in which way it is emerging, and what kind of properties are connected to it. Also, strategies for self-adaptation need to be developed. Furthermore, an array of engineering and assurance questions are opened by a new kind of requirements.

### Contribution

In this survey, we address aspects from the following views: modeling dimensions, engineering, and requirements. To lay the foundations for modeling, we provide a general definition of system modeling followed by an extension to express adaptivity. In doing so, we differentiate between technically motivated and informal categorizations. Furthermore, we identify the actors involved and concerned with adaptivity. This first part of our contribution closes with further relevant research aspects and open questions on context interpretation and inference, comprehensive context models, and specification of and reasoning about adaptive behaviour.

The second part of this survey focuses on aspect of the engineering view. We provide an overview and comparison of current architectural frameworks for adaptive systems. For deeper understanding, we detail on the characteristics of one approach. We raise questions related to validation and verifiction of adaptive systems. The requirements view is highly related to these aspects, as it is concerned with the mapping of requirements onto architecture. Therefore, we briefly address two different ways of collecting requirements for self-adaptive systems.

### Outline

This paper is organized as follows. In Section 2, we discuss the requirements for adaptive systems from an industry point of view. In Section 3, we provide the mathematical foundations for the formal definition of adaptive system behavior. Then, Section 4 makes the notion of adaptive system behavior more precise and discusses corresponding methodological implications for building adaptive systems. In Section 5, we evaluate state-of-the-art architectural frameworks for building self-adaptive systems in the context of Autonomic Computing. Then, Section 6 describes a concrete architectural framework for building adaptive systems which was developed at the TUM; a basic methodology tailored for this framework is outlined. Finally, Section 7 concludes this paper by discussing relevant research questions, open problems, and blind spots in current research.

## 2 Industrial Requirements on Adaptive Systems

This section gives an overview on requirements for adaptive systems from an industry point of view. The requirements are derived from two specific scenarios that apply to many more general scenarios as well.

### Scenario 1: Consumerization of IT

The first scenario is based on a trend that more and more enterprises are currently observing: the consumerization of IT. Instead of centrally managing company owned resources, like PCs, Notebooks, network infrastructure, or mobile phones, IT departments have to shift towards providing services to their users, which can be accessed from arbitrary devices, including devices that are not even owned or managed by the company. Mobile devices enabling employees to access corporate services are part of any modern scenario in commercial and industrial environments. Conceivably, mobile devices will also plug into home entertainment devices to provide the same personal services on a big screen that they provide on the device itself. Designing adaptive systems that provide context-dependent functionality is one of the key challenges in this scenario that needs to be addressed for the devices, and the services that those devices can access.

Nowadays management of enterprise IT resources often implies that a set of policies are enforced on devices, for example, imposing restrictions on passwords or PINs, hard drive encryption, or even disabling hardware drivers (e.g. USB, camera) on devices. As those devices used to be company owned, people accept these restrictions that are enforced on their work devices. A recent trend that enterprises face, and also often actively support, is people bringing their own private devices, like mobile phones and tablets, into the enterprise context and expect them to operate seamlessly in the work environment. Still, people are usually not willing to allow IT operations to manage those private devices as they are also used in the private context, but the enterprise also needs to protect sensitive business data. This is leading to conflicting interests what data should be accessible from a device and how it can be accessed.

To address this problem, a different way of IT operations management is necessary: instead of limiting access and rights on a device, restrictions can now only be imposed on the services that devices access. These services, like email or storage-server access, should behave differently depending on the properties of the device accessing the service. While a fully managed device with enforced access control might be allowed to keep a local cache of company emails also for offline use, privately owned devices might only be allowed access to emails with client software that keep no local cache of the messages. The adaptivity of the service and devices always tries to give the best possible service quality to the user, and is only restricted by the constraints that the IT operations enforce. As these constraints might be conflicting, finding a good or even optimal solution it not a trivial task.

As the perception of IT becomes one of services that are merely mediated by devices, the fixed association of different but similar services to different devices will be perceived as anachronistic. People find it annoying to install different applications on different devices to obtain the "same" service. Instead of having many special purpose devices, which have to be managed individually, today's mobile devices like phones and tablets have the potential to serve as the gateway to all personal services an individual is using. The devices are personalized and have access to all relevant personal context data, like calendar or address book, but also to the sensors of the devices, like GPS or wireless technologies. These devices could utilize special-purpose devices in its vicinity to provide a context-adequate user experience by connecting to TV or media players at home, sensors and antennas in the car, and to company resources at work.

To enable such aspects of this scenario, several challenges have to be addressed: first, a service model will be necessary that supports discovery, aggregation, and adaption. As new devices or sensors might appear and disappear at any time, the applications need to quickly adapt to the changing environment to always provide the best possible service with the available resources. Second, interfaces, protocols, and matching platform features are required to allow a mobile device to smoothly and efficiently connect to and use sensors, actuators, and UI in its vicinity to provide a genuine user experience. Finally, the mentioned functionality should be exposed as a programming model to developers, to allow them concentrating on the application logic instead of orchestrating the different devices and services. The programming model has to provide appropriate abstractions to control distribution of functionality and adaption to changing availability of devices and services.

**Scenario 2: Adaptivity in Data Centers**

A second scenario addresses an issue that operators of data centers often encounter. Datacenter architectures today consist of many virtual machines running on the same PC hardware, several PCs are stored inside a rack and multiple racks are assigned to clusters. Managing such datacenters is a very complex task. The overall goal of the operators is to keep up the functionality of the services, although failures might happen at any level of the architecture. Once a failure is detected, the system automatically adapts its configuration, by isolating the failing component, and redeploying the services to other parts of the data center.

Resiliency against faults by automatically adapting its internal configuration is one of the key requirements for datacenters, as they constantly have to deal with the fact that certain components will fail—from a statistical point of view. To achieve resilience against hardware faults, the nodes need to provide diagnostics functionality, which constantly measures its own state and aggregates the state of the hosted components. A certain level of adaption, like restarting a crashed virtual machine, can be triggered locally, while other changes, like moving virtual machines between nodes, might also require reconfiguring other components of the data center as well.

Many of the necessary function to achieve the required level of adaptivity requires platform support. For example, the platform needs to ensure that a malicious or crashing virtual machine cannot interfere with the monitoring and sensing platform; also moving a running virtual machine from one node to another needs platform support as the virtualized client should not realize that it has been moved. Adaptivity is a key feature of cloud operating systems, and it needs to be investigated how this feature can be exposed to users in an easy to understand abstraction, as today's operating systems already do on a per PC level, for example, by providing the process abstraction.

One of the key challenges in building adaptive systems is supporting the engineer in designing such a system. The engineer requires tool support to verify that an architecture is able to support at least degraded functionality under the assumption that one component fails and that the functionality can be moved to another node. A platform should provide a capability to dynamically deploy, start, and stop functionality on a node and seamlessly move the functionality to another node. Not only functionality, but also data from the environment and other computation nodes must be moved between nodes. If a platform supports dynamic adaption to faults by moving functionality between nodes, the question remains which of several deployment options is the best. A designer needs a way to express quality criteria, which could be used to optimize different deployment options.

Another aspect in this scenario is the optimization of resource utilization in data centers. Data center operators regularly observe peaks in usage of the services at specific times of a day, while at other times the usage is significantly lower. Being able to dynamically scale up and down services, or moving multiple virtual machines to a single host, allows for shutting down hosts and saving energy. Due to the size of the data centers the capability to dynamically adapt to load can help to significantly reduce the cost of operations.

**Conclusion**

We believe that the design of adaptive systems will be a key technology to address many of the challenges we see in next generation systems. On the one hand, the systems designer will require design patterns and frameworks that help in building and deploying adaptive systems; on the other hand, the increasing complexity demands for good design tools which assist the designer and provide capabilities to verify the validity and consistency of the design.

## 3 Foundations

In this section, we introduce the formalism to express our intuition of adaptive system behavior in a concise manner. The basis for our formalism are the Focus theory [5] for specifying interactive systems and the Janus approach [2] as its extension to services. Since we define adaptation in terms of the interactions between a subject, a system and its environment, we concentrate on the system *structure* and the *interface behavior* of the communicating entities. Focus and Janus provide appropriate modeling techniques for treating both aspects on a mathematical basis.

### *3.1 Streams and components*

In Focus, the behavior of a system is described by relating system inputs and outputs. Both inputs and outputs are sent along typed input and output *channels*, respectively. A typed channel is a directed communication line over which *messages* of its specific type are communicated. A type is simply a name for a data set.

Let $M$ be a set of messages, for instance the carrier set $CAR(M)$ of a given type $M$. By $M^*$ we denote the *finite* sequences of elements from $M$. By $M^\infty$ we denote the set of *infinite* streams of elements of set $M$, which can be represented by functions $\mathbb{N}_+ \to M$, where $\mathbb{N}_+ = \mathbb{N} \setminus \{0\}$. By $M^\omega = M^* \cup M^\infty$ we denote the set of streams of elements from the set $M$, which are finite or infinite sequences of elements from $M$. A *stream* represents the sequence of messages sent over a channel during the lifetime of a system.

The *syntactic interface* $(I \rhd O)$ of a system is characterized by its set $I$ of input channels and its set $O$ of output channels. A channel is basically a name for a stream.

Let $C$ be a set of typed channels, a *channel history* is a function

$$x : C \to M^\omega$$

mapping each channel to the messages communicated over that channel, where the stream $x(c)$ carries only messages of the type of $c$. We denote the set of all channel histories for the channel set $C$ by both $\mathbb{H}(C)$ and $\overrightarrow{C}$.

The *interface behavior* of a system specifies an input/output (I/O) function that defines a relation between the input streams and the output streams of a system. It describes the behavior of a system in the most abstract way.

An I/O function is represented by a set-valued function $F : \overrightarrow{I} \to \wp(\overrightarrow{O})$. The function yields a set of histories for the output channels $O$ for each history of the input channels $I$.

An interactive composed system consists of a family of subsystems called *components*. These components interact by exchanging messages via their channels, which connect them. A network of communicating components forms a *system architecture*, which can be graphically represented by directed graphs (cf. Fig. 2). Components are depicted as rectangles, directed channels are denoted as arrows, pointing from their source to their destination. A channel can be labeled to indicate its name and data type ($name : Type$). We refer to [5] for further details regarding the properties of composition and the semantics of Focus specifications.

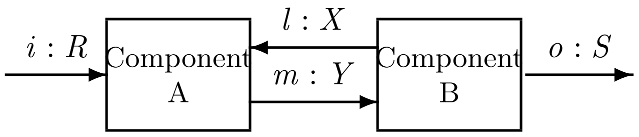

Fig. 2. Syntactic interface of two components

## *3.2 Services*

A service has a syntactic interface ($I \rhd O$) and a corresponding behavior function $F : \overrightarrow{I} \rightarrow \wp(\overrightarrow{O})$ just like a component. By $\mathbb{F}[I \rhd O]$ we denote the set of all service interfaces with input channels $I$ and output channels $O$. However, as opposed to a component, a service may have a *partial* interface behavior, i.e. $F$ is only defined for a subset of input histories $x \in \mathbb{H}(I)$. This subset is called the *service domain*, which is defined by

$$Dom(F) = \{x : F(x) \neq \emptyset\}.$$

The corresponding *service range* is defined by

$$Ran(F) = \{y \in F(x) : x \in Dom(F)\}.$$

Hence, a component is a special case of a service, with either an empty or a total domain.

In order to relate a system with the services it provides, we use the concept of *splicing* or *projection*, respectively. For a set of typed channels $C' \subseteq C$ and a channel history $x \in \mathbb{H}(C)$, we denote $x|_{C'} \in \mathbb{H}(C')$ as the *restriction* of $x$ to the channels and messages in $C'$.

Given a service $F \in \mathbb{F}[I \rhd O]$ and a subset of channels $I' \subseteq I$, $O' \subseteq O$, we define its projection $F \dagger (I' \rhd O') \in \mathbb{F}[I' \rhd O']$ to the interface $(I' \rhd O')$ as follows: for all input histories $x' \in \mathbb{H}(I')$,

$$F \dagger (I' \rhd O')(x') = \{y|_{O'} : \exists x \in \overrightarrow{I} : x' = x|_{I'} \wedge y \in F(x)\}.$$

We call a projection $F \dagger (I' \rhd O')$ *faithful* if for all input histories $x \in Dom(F)$, we have

$$F(x)|_{O'} = F \dagger (I' \rhd O')(x|_{I'}).$$

In other words, a faithful projection does not introduce additional non-determinism, since the input deleted by the projection does not influence the output (cf. [3] for more details).

We also say, that a service $F' \in \mathbb{F}[I' \rhd O']$ is a *sub-service* of a service $F \in \mathbb{F}[I \rhd O]$, where $I' \subseteq I$, $O' \subseteq O$, if $F' = F \dagger (I' \rhd O')$ (see Fig. 3). In this light, a faithful projection is a projection of a behavior to a sub-service that forms an independent sub-behavior, where all input messages in $I$ are included that are relevant for the considered output messages.

Let $F_1 \in \mathbb{F}[I_1 \rhd O_1]$ and $F_2 \in \mathbb{F}[I_2 \rhd O_2]$ be two services with $I_1 \subseteq I_2$ and $O_1 \subseteq O_2$, respectively. $F_2$ is a *service refinement* of $F_1$, denoted by $F_1 \rightsquigarrow F_2$, if for all input histories $x' \in Dom(F_1)$,

$$\{y|_{O_1} : \exists x \in \overrightarrow{I_2} : x' = x|_{I_1} \wedge y \in F_2(x)\} \subseteq F_1(x') \quad \text{and}$$

$$Dom(F_1) \subseteq \{x|_{I_1} : x \in Dom(F_2)\}.$$

In other words, the refining service $F_2$ behaves *more* deterministic than the refined service $F_1$, since the set of possible output histories of $F_2$ is reduced. On the other hand, the service domain of $F_2$ is enlarged. This enables us to properly replace each occurrence of $F_1$ by its refinement $F_2$, without violating any design contracts.

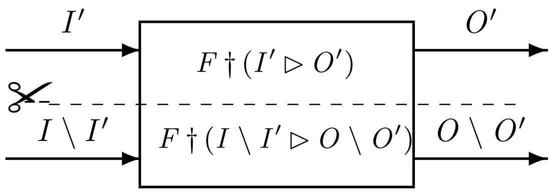

Fig. 3. Splicing a sub-interface of service $F$

## 4 Adaptive System Behavior

### 4.1 Informal Definitions

Basically, adaptive systems are interactive systems, i.e. systems that continuously exchange information with their environment. At a certain level of abstraction it is formally insignificant, from where input to the system originates, e.g. whether the input comes directly from a user or a technical sensor. Therefore, a key question is which aspects differentiate non-adaptive behavior from adaptive behavior? A tendency is that observing the environment makes the difference. In order to investigate the crucial decision criterion, we classify the system inputs into direct / explicit inputs as well as indirect / implicit inputs and use an explicit user model to distinguish both. The user model comprises all interaction possibilities for a user.

The methodological implications of this decision are discussed in Section 4.3. We assume that user inputs are always entered explicitly. Intuitively, a user experiences an adaptive system behavior, if the system reaction resulting from his inputs is additionally determined by some information about the environment, i.e. implicit inputs. In the following, we distinguish between four perspectives of observable system behavior w.r.t. the user.

*1. Non-adaptive behavior:* The system behavior is exclusively determined by user inputs. In particular, the behavior is independent of implicit inputs of the system environment.

*2. Non-transparent adaptive behavior:* Both user inputs and implicit inputs of the proximate system environment determine the behavior of the considered system. In addition, we assume that the user can not observe the implicit inputs that influence the system behavior. Infrared measurements are examples for non-observable implicit system inputs.

*3. Transparent adaptive behavior:* Again, the system behavior is determined by user inputs as well as inputs of the environment. As opposed to the previous perspective, the user (partially) observes those implicit inputs. Hence, the user infers a relationship between events occurring in the observable environment and the resulting system reactions. Nevertheless, the user is not able to influence or control the former. Weather conditions are an example thereof.

*4. Diverted adaptive behavior:* Consider that the user is also able to control some of the implicit inputs of the environment. Obviously, users might unintentionally influence their environment – and sometimes adaptation exactly relies on detecting these influences. However, what we are concerned with is the intentional influencing of the environment in order to achieve certain system reactions. Users influencing a system in that way certainly miss some directly accessible usage functionality or are unsatisfied with the resulting system reactions. We say that a system exposes this "diverted adaptive" behavior, whenever a user exploits this additional possibility for influencing the system.

The above classification implies that a system can not be attributed to behave adaptive without explicitly relating to

- a subject (person or other technical system) and
- the environment of use (modeled as context).

Regarding the subject we differentiate between

- ordinary users that have no mental model of the adaptive system behavior, i.e. from their perspective the system exposes a non-transparent adaptive behavior;
- cautious users having a mental model of the adaptive behavior, but do not

try to control the implicit system inputs of the environment;

- manipulative users, which intentionally influence the environment in order to achieve certain system reactions.

This classification leads to a user taxonomy, which we recommend to take into account when constructing adaptive systems (cf. Section 4.3). Formal definitions of the four perspectives are presented subsequently. Details concerning the proposed user taxonomy can be found in [27].

## *4.2 Formal Definitions*

According to the classification provided in Section 4.1, we distinguish between three different types of adaptive system behaviors; two of them are formally described in the following. Basically, we consider the interactions between three entities:

- the *subject* $U$ (person or other technical system) communicates with,
- the *system* $S$ through a set $I_U$ of input channels and a set $O_U$ of output channels, respectively,
- the system $S$ observes certain aspects of its proximate *environment* $E$ by means of sensors through a set $I_E$ of input channels.

The system might also influence its environment by actuators sending signals along a set $O_E$ of output channels. The basic construction is formulated in terms of the abstract system architecture illustrated in Fig. 4–6. All channels just described are depicted as solid arrows.

### *4.2.1 Non-transparent adaptive behavior*

The non-transparent adaptive behavior is schematically presented in Figure 4: the subject $U$ is not able to observe or influence the system environment $E$, since no communication channels between $U$ and $E$ exist. The interaction takes place *only* between the system and its environment and between the subject and the considered system, respectively. Thus, the syntactic interfaces of $U$ and $E$ are denoted by $(O_U \rhd I_U)$ and $(O_E \rhd I_E)$, respectively.

The system $S$ exposes the syntactic interface $(I \rhd O)$, whereby $I = I_E \cup I_U$ and $O = O_E \cup O_U$. By $S^U$ we denote the sub-service of system $S$ obtained by projecting the overall interface $(I \rhd O)$ to the communication channels with subject $U$, that is, $S^U = S \dagger (I_U \rhd O_U)$. Sub-service $S^E$ is defined analogously, that is, $S^E = S \dagger (I_E \rhd O_E)$.

In order to formally characterize the notion of adaptive system behavior, we introduce a predicate *nonfaithful* : $\mathbb{F} \times \mathbb{F} \to \mathbb{B}$, which is defined for a service

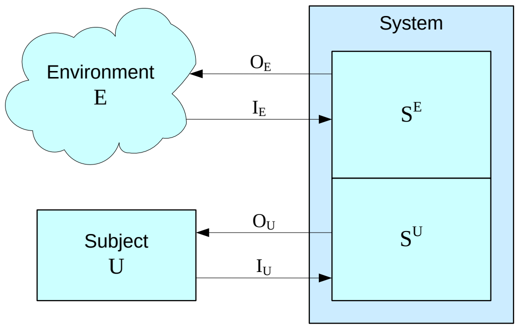

Fig. 4. Architecture of "non-transparent" adaptive behavior of the system w.r.t. the subject.

$F \in \mathbb{F}[I \rhd O]$ and its sub-service $F' \in \mathbb{F}[I' \rhd O']$, where $I' \subseteq I, O' \subseteq O$:

$$\begin{aligned} &\mathit{nonfaithful}(F, F') = \exists x, y \in \mathit{Dom}(F) : \\ &\quad F' = F \dagger (I' \rhd O') \wedge x|_{I'} = y|_{I'} \wedge F(x)|_{O'} \neq F(y)|_{O'}. \end{aligned}$$

In other words, $\mathit{nonfaithful}(F, F')$ is true if $F' = F \dagger (I' \rhd O')$ is not a faithful projection of $F$, that is, $\neg \forall x \in \mathit{Dom}(F) : F(x)|_{O'} = F'(x|_{I'})$. More precisely, we obtain the following two equations

$$\begin{aligned} &\exists x \in \mathit{Dom}(F) : F(x)|_{O'} \subset F'(x|_{I'}), \\ &\forall x \in \mathit{Dom}(F) : F(x)|_{O'} \subseteq F'(x|_{I'}). \end{aligned}$$

Thus, $F'$ introduces additional non-determinism since it abstracts away in projection $x\,|_{I'}$ some input messages of $x$ that are needed to determine the output in $F(x)|_{O'}$.

**Definition 4.1** System $S$ exposes a *non-transparent adaptive behavior* w.r.t. subject $U$ if $\mathit{nonfaithful}(S, S^U)$ is true, i.e., $U$ observes some non-deterministic system reactions of $S$.

Clearly, each implementation is deterministic. Hence, the indeterminate system reactions observed by $U$ are attributed to (unobservable) system inputs, which implicitly influence the behavior of $S$. We call such input *context*.

**Example 4.2** A *context-aware car navigation* which adjusts the route in case of a traffic jam exposes a non-transparent adaptive behavior w.r.t. an *ordinary driver*, who has not already been informed about the traffic jam.

### *4.2.2 Transparent adaptive behavior*

In order to express that subject $U$ may at least partially *observe* the system environment $E$, we first of all introduce an additional set of communication channels $C_1$, pointing from $E$ to $U$. $C_1$ refines the syntactic interfaces of both entities to $E'$ and $U'$, respectively. Furthermore, we split up the contextual system inputs $I_E$ into two disjoint channel sets $I_{E'} \cap C_1$ (contextual inputs observable by $U'$) and $I_{E'} \setminus C_1$ (contextual inputs not observable by $U'$). Fig. 5 illustrates this situation.

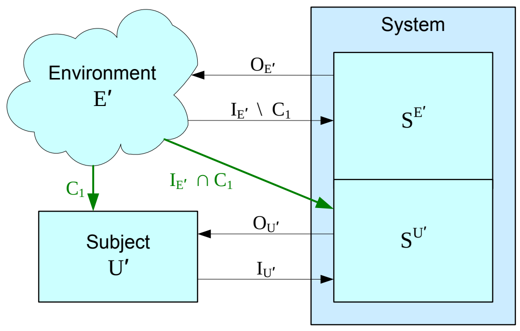

Fig. 5. Architecture of "transparent" adaptive behavior of the system w.r.t. the subject.

We exploit *service refinement* as described in Section 3.2 for our further argumentation concerning the notion of "transparent adaptive" behavior. Consider a "cautious user" $U'$ with the syntactic interface $(O_{U'} \cup C_1 \triangleright I_{U'})$, who at least partially observes the system environment through the channel set $C_1$.

We assume that $U'$ interprets the messages received along $C_1$ as contextual system inputs. From the perspective of $U'$, the service $S^{U'}$ constitutes an appropriate characterization of the system's behavior, since its interface comprises all system inputs and outputs observable by $U'$. In comparison to the service $S^U$ that represents the "non-transparent adaptive" behavior, $S^{U'}$ behaves more deterministic. Obviously, some of the non-deterministic system reactions previously exposed by $S^U$ can easily be explained by means of the additional system inputs $x \in \mathbb{H}(I_{E'} \cap C_1)$. Formally, $\mathit{nonfaithful}(S, S^{U'}) \Rightarrow \mathit{nonfaithful}(S, S^U)$. Moreover, $S^{U'}$ fulfills the refinement relation defined in Section 3.

**Definition 4.3** A system $S$ exposes a *transparent adaptive behavior* w.r.t. subject $U'$ if the proposition

$$\exists S^U, S^{U'} \in \mathbb{F} : \ \mathit{nonfaithful}(S, S^U) \ \wedge \ S^U \rightsquigarrow S^{U'}$$

is true, that is, due to its refined interface, the subject $U'$ observes a more deterministic service $S^{U'}$ in comparison to $S^U$.

Clearly, the sub-services resulting from those projections to observable communication channels behave more and more deterministically from the perspective of the subject, as we enlarge the set of observable input channels originating from the environment. If the projection comprises all input channels originating from the subject and the environment, respectively, the resulting service $S^{U^*}$ becomes totally deterministic, since it accumulates to the overall system $S$, that is, $S^{U^*} \rightsquigarrow S$.

**Example 4.4** The car navigation from Example 4.2 exposes a transparent adaptive behavior w.r.t. a *cautious driver* who actually recognizes that the route is adjusted according to an oncoming traffic jam, since the jam was announced via radio broadcasting (implicit input for the system).

### *4.2.3 Diverted adaptive behavior*

We denote the last perspective as *diverted adaptive behavior* w.r.t. a "manipulative user". It reflects the circumstance, that a user may not only observe but also *influence* the system environment over a set $C_2$ of communication channels (cf. Fig. 6). A formalization of this circumstance results in refining the system environment $E'$ in order to behave more deterministic. For the sake of brevity, we skip the formalization, since it does not contribute to a better understanding. The corresponding methodological implications are discussed subsequently.

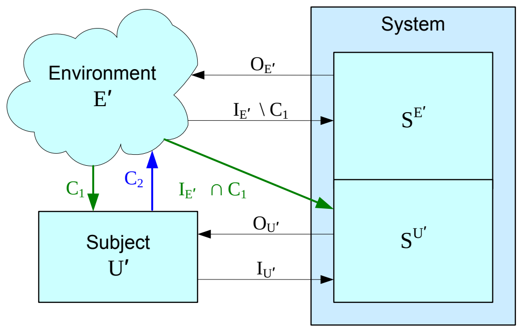

Fig. 6. Architecture of "diverted" adaptive behavior of the system w.r.t. the subject.

## *4.3 Methodological Implications*

*Requirements elicitation:* As discussed in the previous sections, the behavior of a (context) adaptive system does not only depend on user inputs and internal system states. "External" factors concerning the situation of use need also to be taken into account. The user needs associated with a certain usage situation constitute such an external factor. The context elicitation denotes the identification of such external factors in terms of context. The formal definitions provided in [4] contribute to this task by helping the requirements engineer to specify the context information, which is relevant for the adaptation of the system. Formally, this task comprises the specification of the environment $E$, the subject $U$ and all contextual channels $I_E$ between the environment and the system.

*Architectural pattern:* The decomposition of the system's functionality into two sub-services (cf. Fig. 4) indeed constitutes a design proposal. It already suggests a certain decomposition of the overall system into two main constituents as proposed in [29]:

- a service $S^U$ comprising all core functionalities of the system, that is, the set of all observable behaviors, and
- an adaptation logic service $S^E$, which is responsible for realizing the "most appropriate" behavioral pattern for a given usage situation.

Both constituents interfere with each other in order to provide the required system behavior, while their separate design fundamentally reduces the system complexity.

*Tailored methodology*: The template-like structure of our definition enables the designer to determine whether a considered system exposes an adaptive behavior. If the system's interface description contains some output channels to the user, whose histories not only dependent on the user's input, but also on additional information about the proximate environment, this fact can be exploited. In order to reduce the system complexity and to facilitate the model analyses, a designer might

- reconsider some design decisions along the lines of the architectural pattern described in the following section and [29], respectively, and
- choose a tailored design methodology, such as [13,11,10,28], which facilitates the specification of adaptive behavior and exploits appropriate models, such as mode-automata for structuring reconfigurable behavior.

We review such a tailored methodology in Section 6.3.

*Unwanted behavior*: A "perfect" adaptive system would—depending on the situation of use—always expose the most appropriate behavior to the user. This system adapts its behavior "perfectly transparent", i.e. the user always understands and affirms the taken adaptation decisions of the system, respectively. Given this idealized system, all misunderstandings concerning the system usage and related effects like automation surprises [26] and unwanted behavior [12] can be neglected. However, as argued in Section 3, a perspective called "diverted adaptive behavior" often can be experienced, which is associated with profound methodical implications. The detection of use cases exposing such a behavior reveals deficiencies of the system's interaction interface or its adaptation logic. In the first case the system's interaction interface needs to be enhanced by additional services the user is currently missing. The second case requires a mechanism for influencing the logic of the adaption. Fahrmair et al denote such a mechanism as *calibration* [9].

## 5 Architectural Frameworks

In this section, we summarize and discuss relevant requirements for architectures and frameworks enabling the construction of adaptive systems along the lines of [16]. We focus on one particular application domain which is closely related to adaptive systems: autonomic computing (AC). In a nutshell, AC denotes an emerging paradigm aiming to simplify the administration of complex computer systems. In this light, adaptation can be regarded as an enabling technology for future applications in the AC field. Consequently, we present a survey of past and future secrets of this enabling technology within the AC context. Section 5.1 presents the criteria for evaluating state-of-the-art architectural frameworks, while Section 5.2 evaluates these frameworks concerning the criteria.

### *5.1 Evaluation Criteria for Frameworks*

*Adaptability:* The core concept behind adaptability is the general ability to change a system's observable behavior, structure or realization [12,8]. This requirement is amplified by *self-adaptation*. Self-adaptability enables a system to decide about an adaptation by itself—in contrast to an "ordinary" adaptation, which in turn is decided and triggered by the system's environment, e.g. a user or administrator. Adaptation may affect the change of some functionality, algorithm or parameters as well as the total system structure or any other aspect of the system. If an adaptation comprises the change of the complete system model, including the model that actually decides on the adaptation and deploys the decisions, this system is called a totally reconfigurable system. In case a change of behavior can be expressed by exchanging some functional entities, the system is simply called reconfigurable. Self-adaptation requires a model of the system's environment. This model is often referred to as context. Thus, self-adaptation is also called context adaptation. We use both terms synonymously throughout the paper.

*Awareness:* Awareness is closely related to adaptation and context, as it is a prerequisite for self-adaptation. It has two aspects: self-awareness enabling a system to observe its own system model, state, etc. and awareness of the environment. As stated above, the model of the system's environment is often called context [8,17]. More precisely, we denote context as the sufficiently exact characterization of the situations of a system by means of perceivable information that is relevant for the adaptation of the system [12]. In principle any (measurable) characteristic of the system or its environment could be considered for adaptation decisions. A systematics to model a system's context is proposed in [29].

*Introspection / Monitoring:* Since monitoring is often regarded as a prerequisite for error discovery and handling, it is also required in the context of awareness [23,20]. The peculiarities of monitoring are not discussed within this document. However the notion of monitoring is relevant for this discussion, since it is closely related to the notion of context. Context embraces the system state, its environment, and any information relevant for the adaptation. Consequently, it is also a matter of context, which information for instance indicates an erroneous system state and hence characterizes a situation in which a certain adaptation is necessary. In this case, adaptation can be compared to error handling, as it transfers the system from an erroneous (unwanted) system state to a well-defined (wanted) system state.

*Dynamicity / Reconfigurability:* Dynamicity embraces a system's ability to change during runtime. In contrast to adaptability, this only constitutes the technical facility of change. While adaptability refers to the conceptual change of certain system aspects, which does not necessarily imply the change of components or services, dynamicity is about the technical ability to remove, add, or exchange services and components. Once more, there is a close but not dependable relation between both dynamicity and adaptation. Dynamicity may also include a system's ability to exchange certain (defective or obsolete) components without changing the observable behavior. Dynamicity deals with concerns like preserving states during functionality exchange, starting and stopping functionality etc.

*Autonomy:* As the term Autonomic Computing already suggests, autonomy is one of the essential characteristics of such systems. AC aims at unburdening human administrators from complex tasks, which typically require a lot of decision making without human intervention (and thus without direct human interaction, i.e. the set of communication channels $I_E$, cf. Fig. 4 is restricted or even contains only channels for system initialization/start). Autonomy, however, is not only intelligent behavior but also an organizational manner. Context adaptation is not possible without a certain degree of autonomy. A rule engine obeying a predefined set of conditional statements (e.g.

*if_then_else*) is the simplest form of autonomy. In many cases, such a simple rule-based mechanism, however, may not suffice.

*Robustness:* Robustness is a requirement that is claimed for almost every system. AC applications will specially benefit from robustness since this may facilitate the design of system parts that deal with self-healing and self-defense. In addition, the system architecture could ease the appliance of measures in cases of errors and attacks. Robustness states the first and most obvious step on the road to dependable systems. Beside a special focus on error avoidance, several requirements aiming at correcting errors are forced. Robustness is often achieved by decoupling and asynchronous communication. Both are approved techniques in software and systems engineering which help to prevent the propagation of errors.

*Mobility:* Mobility enfolds all parts of the system: from mobility of code on the lowest granularity level via mobility of services or components up to mobility of devices or even mobility of the overall system [20,9]. Mobility enables dynamical discovery and usage of new resources, recovery of crucial functionalities etc. Often, mobile devices are used for detection and analysis of problems.

*Runtime-Traceability:* Traceability enables the unambiguous mapping of the logical onto the physical system architecture, which inter alia facilitates an easy deployment of necessary measures [20]. The notion of traceability is once more closely related to that of adaptation: adaptation decisions namely also base on an abstract system model in order to reduce the necessary computational power. These decisions are afterwards deployed in the physical system, too. The deployment is usually automatic, and thus requires traceability. Traceability is additionally helpful when analyzing the reasons for wrong decisions made by the system.

*Criteria relations:* Fig. 7 shows the relations between the above evaluation criteria, which is based on the categorization suggested by [6]. First, we assume that a set of *goals* is given for a system. Goals have to maintained by the system by all means and are the aim of each adaption the system decides to take.

Each adaptive system needs to gather information about the *change* of its environment or the internal system state in order to adapt its behavior. In this sense, *awareness* of the present context of use can be understood as a key prerequisite of adaptive behavior. Thereby, the concept of *introspection* and

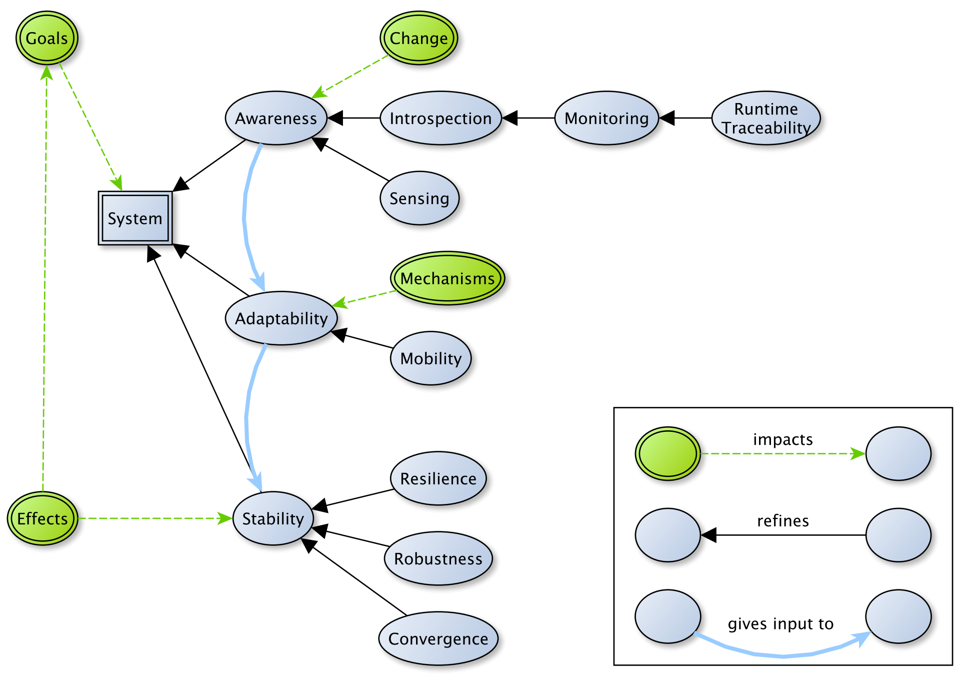

Fig. 7. Relations between evalutation criteria

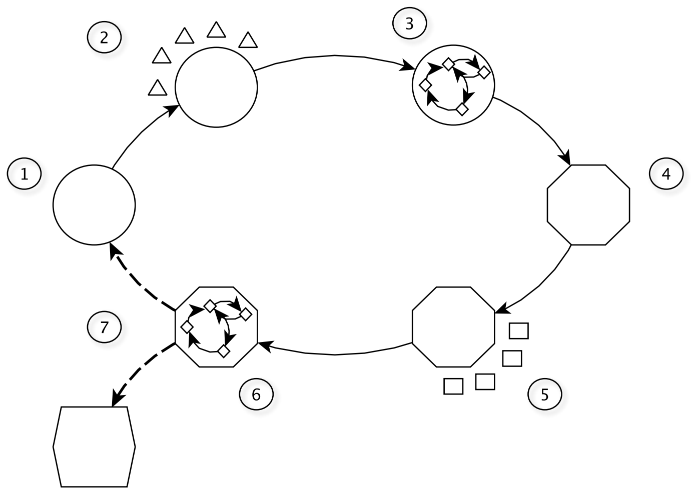

Fig. 8. Exemplary adaptation cycle

more specifically *monitoring* or *(runtime-) traceability* can be seen as a special case of awareness, which focuses on the internal states of the system—instead of the system's environment. *Sensing* is used to gather external information, e.g. through the usage of sensors.

*Adaptability* enables the system to change to a new situation. Therefore the system is using different mechanisms which can be applied to reach the goals of a system. Among the set of systems with an adaptive behavior, more sophisticated systems expose at least some degree of *autonomy* with respect to their reactions onto system inputs. The degree of autonomy thereby determines how intense a user must interact with the system to accomplish a given objective. Another, not necessarily disjoint set of systems with an adaptive behavior is able to change its internal structure on the service, component, or hardware level during runtime, which we denote by *reconfigurable* or *dynamic* systems. A third set of systems with an adaptive behavior can be attributed by the term *mobile*, which for instance implies that such a system runs on a mobile device that can dynamically connect to other devices in the (proximate) environment, e.g. through wireless communication channels. Note that the above sets of systems are not necessarily disjoint, so that an adaptive system can satisfy any combination of the attributes awareness, mobility, and dynamicity.
Each adaptation may have *effects* on the system and its environment which may influence the *stability* of the system. *Resilience*, *robustness* and *convergence* are refined measures to overcome an unstable state and to stabilize the system again.

*Adaptation cycle:* Fig. 8 illustrates the cycle in which systems adapt to a changing context. The observation starts at a given, stable state and configuration of the system (1). The information obtained by aware parts of the system, like sensors, indicates a change in the context of the system (2), symbolized by triangles surrounding the system in the figure. In order to maintain its goals, the system may use this stimulus to activate mechanisms that e.g. change its structure or configuration (3). In step 4 the system is adapted to the new usage context, the effects of the adaptation become visible. Whenever a new stimulus is sensed (5), the system may use its adaptation mechanisms (6) to return to a already taken configuration (1) or to change again its structure.

## *5.2 Evaluation of Architectural Frameworks*

There is a number of architectural styles and frameworks that already support at least some of the above mentioned requirements for Autonomic Computing applications. In the following, a brief characterization of currently available styles and frameworks is given in order to enable a later mapping onto the AC requirements. We evaluate the frameworks especially regarding their suitability to develop AC applications.

*Accord Framework:* The accord framework [1] was designed to cope with three challenges: heterogeneity, dynamism and uncertainty. As the inventors had a focus on grid environments, these three problems are especially immanent. Therefore the framework implements three requirements that aim at solving these problems:

(i) separate interface definition

(ii) separate computational behavior from interaction and coordination

(iii) computation, interaction and coordination should be context aware to adapt them to "dynamic requirements"

The first two requirements are attributes of component and service-based architectures. The third requirement is introduced to cope with dynamism. Accord does not explicitly include the application model into the context. The accord framework can be classified as a managed, agent-based architecture. Components act as agents but are observed and managed by a controller. Replacing components is quite complex, since components maintain a state, which has to be migrated during the replacement.

*Weaves Framework:* In Weaves [15], messages are objects that are forwarded and manipulated throughout the system. Weaves facilitates blind communication: connectors and components are separated, components do not know sources or destinations of objects and neither their semantics. This ensures that components can be replaced without rearranging connections and vice versa. Weaves architectures can be edited on the fly. As weaves uses asynchronous communication, typical problems arising from connection loss that are problematic for synchronous communication, are avoided.

*C2 Framework:* C2 [30] is another architectural style that focuses on a hierarchical organization of components to enable decoupling. Components are connected via connectors and are only aware of components that are above them. Therefore direct invocation can only be made from the bottom to the top. Communication is asynchronous and based on a request/reply pattern. However state changes can be propagated via the connectors top-down. Therefore components on a lower level can be exchanged without causing problems on an upper level.

*PitM Framework:* PitM [19] is an extension of the C2 style aiming at "Programming in the Many". Inter-components connection is extended in a way, that—in addition to top and bottom connectors—side connectors enable synchronous component interaction. As a constraint, two components may not be in a side-to-side and top-down connection at the same time. This restriction prevents from ports misuse. Behavior is described by means of provided and required services and interaction via event-based communication. Furthermore, special connectors called border connectors abstract from distribution over devices, since components may not see the device borders. PitM has a second architecture level called meta-level. Components on this level act as effectors that are aware of the application-level components and may interact with them. This meta-level controls the application behavior. Application data messages are used simply for application related communication, whereas component content, architectural model and system monitoring messages are used to coordinate adaptive features.

*CAWAR Framework:* The Cawar framework [21,22,9,12] differs from the former described approaches in that it is a purely service-based approach. Cawar distinguishes four basic service types: *sensors* acquire data, *interpreters* process data and *actuators* deploy instructions. *Context elements* are abstract and possibly distributed information buffers, which decouple the other three service types dealing with the data processing. Certain constraints govern the possibilities for composition and ensure unambiguous data processing. The usage of services abstracts from implementation details and focuses on describing the behavior of the specified system. All Cawar architectures are self-describing, i.e. the system model itself is part of the system and stored in a dedicated context element. This enables inferencing from the model. The Cawar architecture contains a special management service called *model activator*. This service implements the adaptation decisions within the system model, which may lead to a reconfiguration of the system. Since the activator itself is part of this model, even the activator may be affected by such an adaptation decisions (total reconfigurability). The communication between context elements and the other service types is synchronous, while it is asynchronous in-between the three other types due to the mentioned decoupling via context. Beside the service communications, direct communications on the component layer are also possible and can be negotiated via the context. The context thus buffers information concerning the system states [1], the system model itself and the system environment.

*Summary of the Evaluation:* Each of the architectural styles described above supports a subset of the former described requirements. All of them are suitable for designing self-healing application, or more generally AC application with specified self-X attributes. However, some architectural styles better support the design of certain self-X requirements than others. Table 1 summarizes the evaluation results, whereas "o" means the architecture style does not explicitly support the requirement; however it does not avoid its fulfillment in principle. "+" means the architecture style somehow supports the requirement, while "++" expresses, that the corresponding requirement is highly supported by the architecture. As mentioned in the introduction, we focus our considerations on the self-healing attribute, although support of the further attributes may be straightforward.

We consider both PitM and Cawar as promising approaches for equipping system with self-X characteristics, since they support the design of context-

[1] the involved services are usually stateless.

| | Accord | Weaves | C2 | PitM | CAWAR |
|---|---|---|---|---|---|
| **Adaptability** | + | o | o | ++ | ++ |
| **Awareness** | + | + | + | + | ++ |
| **Monitoring** | + | o | o | ++ | ++ |
| **Dynamicity** | + | + | + | + | ++ |
| **Autonomy** | + | + | + | + | + |
| **Robustness** | + | + | + | + | + |
| **Mobility** | o | + | + | ++ | ++ |
| **Traceability** | o | o | o | + | ++ |

Table 1
Evaluation of Different Architecture Styles

adaptive systems. C2 and Weaves state first attempts in supporting more flexible system designs. Accord has a slightly different focus, since it is designed for grid environments, where context is usually limited to indicate the availability of certain resources within the computer grid.

Both PitM and Cawar support all requirements for Autonomic Computing. However, autonomy for any style and framework is highly dependent on the applied techniques for decision making. Cawar provides a more sophisticated awareness concept than PitM, since Cawar architectures dictate an explicit model of the system environment in form of context. Moreover, Cawar inherently supports self-awareness of the system model for inference purposes, whereas PitM merely considers awareness of application related aspects; adaptation related aspects of the system are not taken into account. Cawar explicitly integrates adaptation and application aspects: the context-adaptive system behavior is specified by only four service types, thus enabling a comfortable design of system architectures. Thanks to management services like the model activator, this architecture can be discovered, deployed and reconfigured at runtime. We refer to Section 6.2.3 for a detailed evaluation of the Cawar framework with respect to the above criteria.

## 6 CAWAR Design Methodology

We now review a basic design methodology tailored for the Cawar framework along the lines of [10]. This framework is based on the notion of a calibrateable adaptation model (K-Model) which is introduced below. Then, the Cawar architectural framework is described in a nutshell. A basic development process for constructing K-Models for the Cawar framework concludes this section.

### *6.1 Calibrateable context adaptation model*

Several architectures and fewer modeling techniques have been evolved for designing (context-) adaptive systems, whereby a couple of selected approaches are shortly summarized in [8] together with their associated pros and cons. The general idea of the Cawar modeling approach is to provide system designers with a clear and structured notation for explicitly representing the (context-) adaptive system behavior, which can be communicated to end users. The principle of this notation is to make both the modular structure and the workflow of the adaptation subsystem explicit in such a way that reasoning about the modeled system behavior is facilitated. This approach explicates the decision logic for the adaptation and makes it comprehensible and modifiable for (expert) users. In conventional software systems, this logics is usually inaccessible for the outside. In contrast we recommend to express the adaptive system behavior by means of adaptation models or shortly K-Models, which are formally founded on the Focus theory introduced in [5].

The development of the proposed modeling technique and it's corresponding implementation on the basis of the Cawar framework (Section 6.2) is motivated by an observation that was made over and over deploying adaptive systems within real world environments: despite of running perfectly under laboratory conditions, these prototypes usually exposed some kind of unexpected behavior when deployed in the wild, even though the underlying specification was accurately implemented. Hence this phenomenon can not be reduced to classical implementation defects. Since no established notion concerning this observation exists, we simply call it *Unwanted Behavior* or shortly *UB* [12]. The reasons for Unwanted Behavior can on the one side originate from insufficient Requirements Engineering (RE), in which certain user needs and usage situations are overlooked. On the other side does even the most sophisticated RE process ultimately result in a requirements specification, which is an abstraction based on static assumption made at some stage in the development process. However this inherently *static* abstraction is consequently subject to the *frame problem* [24] known from AI. In either case does the system model generated at design time not comply with the mental model, which the user is currently associating with the considered system. In consequence a system behavior is exposed, which differs at least for certain usage situations from what the user would expect. Since the frame problem is still an open – and eventually unsolvable – issue, we propose the K-Model and it's runtime calibration as an efficient mechanism for circumventing this issue.

The K-Model uses only four basic elements for describing an arbitrary complex and adaptive system behavior. These elements structure all possible services of the adaptation subsystem into the four service types *sensors*, *interpreters*, *actuators* and *context elements*, which exhibit a type-specific behavior on their own. Currently two representation forms for documenting K-Models exist:

- For the purpose of designing and communicating adaptation models, a graphical notation augmented by different annotations and abstraction techniques was invented, which enhance the readability of adaptation models.
- A machine-readable representation in form of a platform independent XML file was chosen for implementing adaptation models via the Cawar framework (Section 6.2).

The four basic service types of an adaptation model are described in the following sections.

**Sensors** Sensors are responsible for retrieving relevant information from inside and outside the system. They accomplish this task by writing sensed information to dedicated context elements, which in turn act as information buffers. Sensors within an adaptation model qualify to model physical sensors like thermometers, movement and light sensors as well as internal or external software entities that enter information into the system's context (e.g. a remote web service) or even human beings, since terminal inputs may also be treated as perceivable context information.

**Context elements** Context elements act as buffers for storing arbitrary information. In addition they decouple the three other service types (sensor, interpreter and actuator), since a direct communication without context is not allowed for any of these service types. Depending on the adjacent service types and their associated semantics, a context element may represent measured context data (written by a *sensor*), combined and interpreted information (arranged by an *interpreter*) as well as resulting adaptation decisions which are gathered and implemented by an *actuator* (see Fig. 9).

**Interpreters** Interpreters are the information processing entities within an adaptation model. They both gather input in the form of context elements and store the processed information to context elements. The way how interpreters transform their gathered input relies on the underlying logic the certain interpreter exposes and may embrace a simple data forwarding as well as an arbitrary complex interpretation logic (e.g. rule engine, neuronal network). Interpreters acquire the decision making within the adaptation subsystem and hence may expose certain learning capabilities.

**Actuators** Finally actuators are responsible for implementing a calculated context adaptation by triggering the control components within the core system or the environment, which in turn change the system behavior according to the resulting adaptation decision. The context representing this adaptation decision is usually derived from perceivable sensor data, which is appropriately composed by interpreters as described above. For better differentiation, the context elements coupled with actuators are also called *adaptation context*.

The overall process of context adaptation by means of a K-Model containing the just described service types is illustrated in Fig. 9. The typical activities involved in the adaptation process thereby are the *acquisition of context* (step 1), the *situation identification* (step 2) and the *application of the adaptation decision* (step 3).

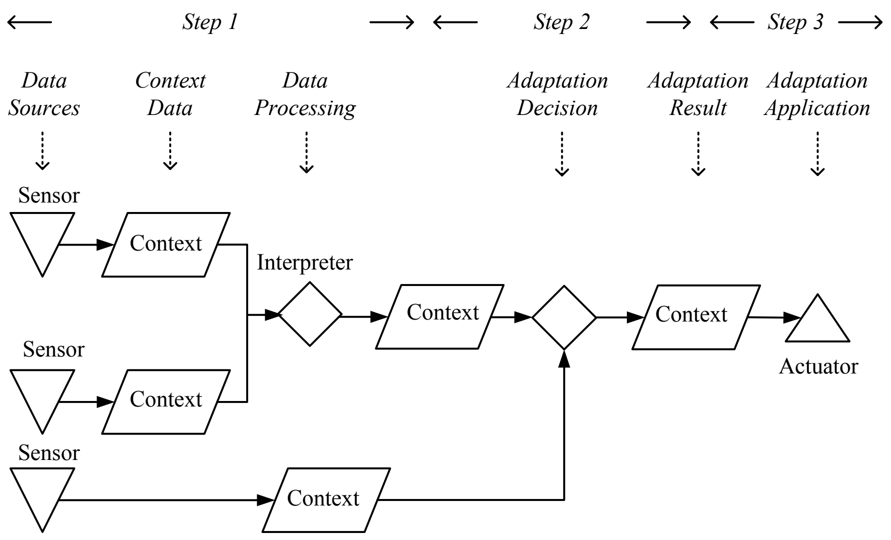

Fig. 9. Context adaptation by means of an adaptation model (K-Model)

As previously stated, these four elements suffice to represent an arbitrary adaptive system behavior. In order to achieve a better structuring of the model, which is easier to read and maintain, it is often beneficial to annotate certain elements like context and interpreters. The context space is thereby separated into a) *sensor context* denoting any information that directly originates from a sensor, b) *situation context* containing all information necessary in order to identify the current usage situation on basis of sensor context and c) *adaptation context* which ultimately comprise the decision about the required adaptation.

Designated interpreters are analogously annotated as *situation adaptors*, which typically combine context information (sensor and situation context) in order to identify a sufficiently exact abstraction of the current usage situation. *Adaptation actions* are another annotation possibility, which helps to model the *situational* requirements occurring in a certain usage situation – represented by a situation adaptor. As opposed to a situation adaptor, an adaptation action is a rather abstract modeling concept (i.e. placeholder) which needs to be refined by one or more basic elements (e.g. a single context element or an interpretation chain), as soon as the exact model representation for this situational requirement is at hand.

It should already be mentioned, that all elements contained in a K-Model are solely *services*; in order to use a contained service, it previously needs to be bound to an actual component fulfilling this service. For maximum software flexibility, this service-component coupling may be delayed until runtime, after appropriate components fulfilling the specified services are discovered. This proceeding is of particular importance when employing external resources only available in certain usage situations: consider an external monitor (environment component), which is detected by the adaptive system hosted on a Pocket PC when entering the room. This monitor is bound to a specified display service for the duration the Pocket PC resides within this room. By the time the user – who is carrying the Pocket PC – is leaving the room, the monitor gets out of range; hence the display service falls back upon it's default display of the Pocket PC. By exchanging the currently available components (*reconfiguration*), the system adapts itself to provide the most appropriate resources currently available, thus enabling a usage in as many situations as possible.

The following definition outlines the main characteristics of a K-Model. An *activator* represents a special actuator, which inter alia identifies and binds components to specified services.

**Definition 6.1** A *K-Model* is a model of a calibrateable context adaption containing an activator that acts on a set of sensors, interpreters, actuators and context elements [9].

### *6.2 CAWAR framework in a nutshell*

The Cawar framework is a generic approach to support all kinds of adaptation in reconfigurable systems. Selected aspects of this framework, which are necessary for the understanding of how context adaptation models are technically realized, are outlined in the following sections. For a detailed conceptual introduction of the framework we refer to [21], whereas a description of it's technical realization can be found in [22].

In this section, a short overview of the overall Cawar (Context aWare ARchitectures) framework is given, while the two subsequent sections discuss certain framework concepts in more detail. The framework principally consists of the following elements:

(i) A set of *components* comprising the technical implementation of typical infrastructure functionality, e.g. context storage, discovery, etc.

(ii) A set of low level *interfaces* (API), that provide the most generic abstraction of context management, e.g. sensors produce context, actuators consume context, etc.

(iii) A *reference architecture* that suggests a basic generic pattern of how a context-adaptive system can be designed in a completely reconfigurable way; formally, context adaptation can be understood as a self-reconfiguring filter [9]. Following that pattern, any implementation of a context-adaptive application can serve as a framework for bootstrapping any other context-adaptive application.

Components, interfaces and architecture together form a basic framework for context awareness and adaptive applications. To develop a certain application, the framework merely must be fed with the desired system behavior in form of an adaptation model, whereby the principles for designing adaptation models are described in Section 6.3. Furthermore the components fulfilling the respective application services must be made available to the framework. However, provided a proper discovery mechanism, such components can be detected and bound at runtime.

The framework initialization is conducted by a designated actuator component named *model activator*, which expects a list of all required services (logical service descriptions) and a list of (currently) available components (technical realizations or references) from the context itself. Such a description is, e.g. given by an XML file representing the adaptation behavior of the considered system, i.e. the K-Model. This model has to be previously read by a special sensor and written into an appropriate context element. The context comprising the K-Model can be further processed—allowing for self-introspection and self-adaptation—before it is ultimately deployed by the model activator. The latter finally reorganizes the services (sensors, interpreters, etc.) as well as their corresponding component bindings (if available), thus reconfiguring the system in order to technically implement the K-Model.

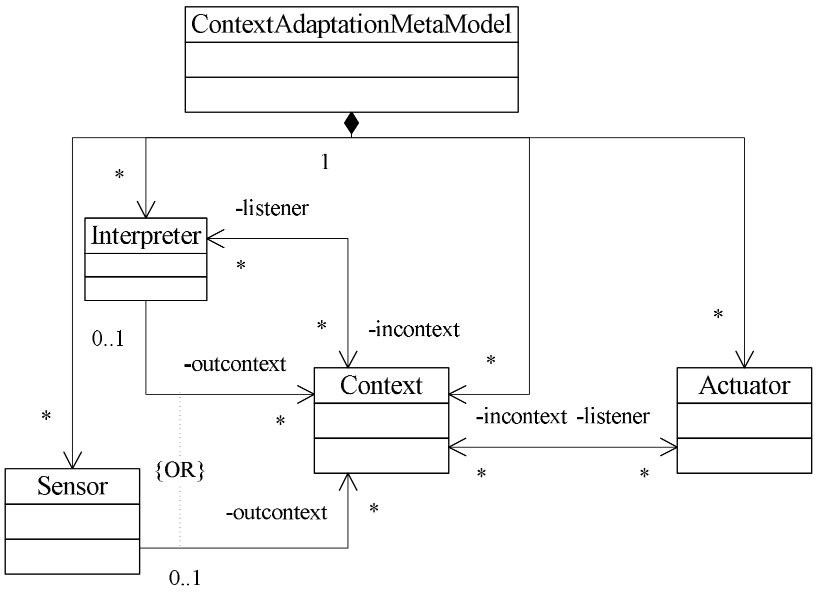

Fig. 10. Meta model describing the principle structure of the K-Model

The underlying meta model of any K-Model, which describes the principle composition of it's logical service architecture—so to speak it's grammar—is illustrated in Fig. 10. A concrete K-Model, containing the particular services that constitute the adaptive behavior of a considered application, is indeed an *instance* of the depicted meta model, which can be read by the model activator for initializing the context-adaptive system as mentioned above.

### *6.2.1 Syntactical and semantical types*

As previously mentioned, all services are contained in a K-Model specified by a logical service description. Each description thereby contains a syntactical (syType) and semantical (smType) description of the service it provides. Both types are used to match a suitable technical component that could implement the specified service. In other words, syType and smType specify, which components the service 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 context elements, including at least the data format accepted. Moreover it could contain any other technical information needed to reduce the number of matching components, such as QoS parameters, billing information etc. syType should contain information needed by the activator to contact and bind possible candidates or it contains even the reference to a single component instance ensuring 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 with respect to the context, such as outside temperature, inside temperature, kitchen termina,l or entrance terminal. A syntactical description is insufficient in this case since it could match more than one component instance. In order for the activator to distinguish which component instance should be bound to, e.g. a sensor that delivers an “outside temperature” an smType can be used. One of the available sensors needs to be marked with a meaning of “outside temperature” as well. Note that semantical marking is specific to the application scenario and hence part of the context and one of the main tasks of calibration.

It should be mentioned that the real “meaning” (smType) 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 behavior, can have different meanings in two different observation contexts. For example the same camera instance that shows the entrance of a building additionally could show for one observer a certain street segment while for a third observer it shows the weather conditions, 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.

### *6.2.2 Application subsystem*

A context-adaptive system built with the Cawar framework typically consists of three subsystems: a) the adaptation subsystem embracing all parts that are responsible for adaptive behavior and which are subject to the frame problem b) the system environment including all service-fulfilling components which are not permanently available due to resource restrictions and c) the application subsystem comprising a single system bootstrapper (System Seed) with all components vital to the running system. Each application usually has its own System Seed that can be installed and uninstalled separately. A System Seed package typically includes:

(i) A boot sensor,

(ii) An optional boot actuator and

(iii) The applications core system.

Usually the application core system initialized by the System Seed contains only a boot sensor specification and an administration component 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. These core system components are necessary for providing a required minimal functionality of the system. This may include at least the necessary framework components as well as a default context server, which handles the initial service communication by storing the messages to (persistent) context elements. Following the principles described in the previous sections, this minimal functionality can of course be extended on the fly, in case the corresponding resources for fulfilling additional services become available.

### *6.2.3 Evaluation of CAWAR w.r.t. the Framework Criteria*

We illustrate how the Cawar framework contributes to the individual evaluation criteria from Section 5.1 in the following.

**Adaptability** is realized in Cawar through the concept of an adaption model (K-Model), which contains the logic that is responsible for the adaptation of the system behavior. Every aspect of the system behavior that is prone to the frame problem must be represented in an adaptation model.

**Awareness** and **monitoring** are realized in Cawar through the sensor services contained in the adaptation model and their implementations in terms of system components.

**Dynamicity** is realized in Cawar through the dynamic binding and the reconfiguration possibilities of the services contained in the adaptation model. In this light, changing an adaptation model and deploying it via the model activator may change each of the system's structural and behavioral aspects.

**Autonomy** is realized in Cawar through the service-fulfilling components, which are bound to the services contained in an adaption model, such as interpreters and actuators. These may be autonomous, third-party components or tailored in-house developments.

**Robustness** is realized in Cawar as a combination of several concepts, such as the dynamic service-component binding, which decouples an abstract service of the adaptation model from its (unreliable) component implementation, the context elements, which decouple the potentially unreliable interservice communication, and the model activator implementing theses concepts.

**Mobility** is realized in Cawar through the portability of the application subsystem and the .NET technologies underlying the framework.

**Traceability** is realized in Cawar through the service-component binding via syntactical (syType) and semantical (smType) types.

## *6.3 Methodical Steps*

The development of adaptive systems for realistic scenarios is today, fifteen years after the announcement of the vision of ubiquity by Mark Weiser [32], extremely difficult and, if at all, only prototypically possible. Development tools and methods are still in an early stage [7]. This is above all a consequence of the fact that these applications are very complex regarding aspects, such as multi-functionality, distribution and situational context. The development as well as the deployment of context-adaptive systems is furthermore often associated with very specific challenges as changing environmental conditions and Unwanted Behavior [12].

The methodology proposed in this section considers these challenges and thereby states an iterative design methodology, starting with the design of adequate scenarios and closing with the final design of the adaptive system behavior of the considered application.

The overall methodology for designing adaptation models is structured into eight individual design steps, which in conjunction guide the engineering

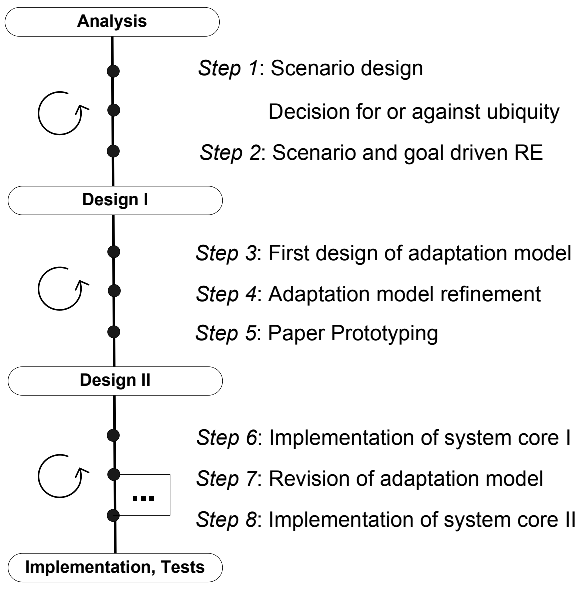
Analysis
*Step 1*: Scenario design
Decision for or against ubiquity
*Step 2*: Scenario and goal driven RE
Design I
*Step 3*: First design of adaptation model
*Step 4*: Adaptation model refinement
*Step 5*: Paper Prototyping
Design II
*Step 6*: Implementation of system core I
*Step 7*: Revision of adaptation model
...
*Step 8*: Implementation of system core II
Implementation, Tests

Fig. 11. The overall Cawar design methodology

of the adaptation behavior of a context-adaptive system. The particular design activities thereby chronologically build upon each other, so that certain results of previous activities are required by succeeding activities. However it is possible, and of course also recommended, to iterate through each individual phase if required. Fig. 11 illustrates the particular steps of the proposed methodology.

It should be mentioned that the activities proposed in the methodology are not mandatory. But, since the proposition is based on several experiences gained from previous application developments, we strongly recommend them. A brief description of the steps illustrated in Fig. 11 is given in the following. The details can be found in [10].

### *6.3.1 Step 1: Scenario Design*

As usual, the methodology starts with the elicitation of functional system requirements in terms of scenarios. As soon as an initial set of scenarios which reflects the usage of the considered application has been designed and was accepted by all involved stakeholders, a decision concerning a ubiquitous or adaptive implementation of the system should be discussed on basis of this scenario set.

Fig. 12 illustrates a simple checklist that is applied in order to determine whether an adaptive or even ubiquitous realization of the application is recommendable or not. Since an adaptive or ubiquitous realization is associated with additional efforts in the development process, this decision should be carefully considered. In case this decision cannot be determined on basis of the previous domain knowledge, this step also can be delayed until the requirements engineering has been completed (step 2 in the overall design process), which ideally results in a complete and non-divergent set of situational requirements sufficient for unambiguously resolving the ubiquity decision.

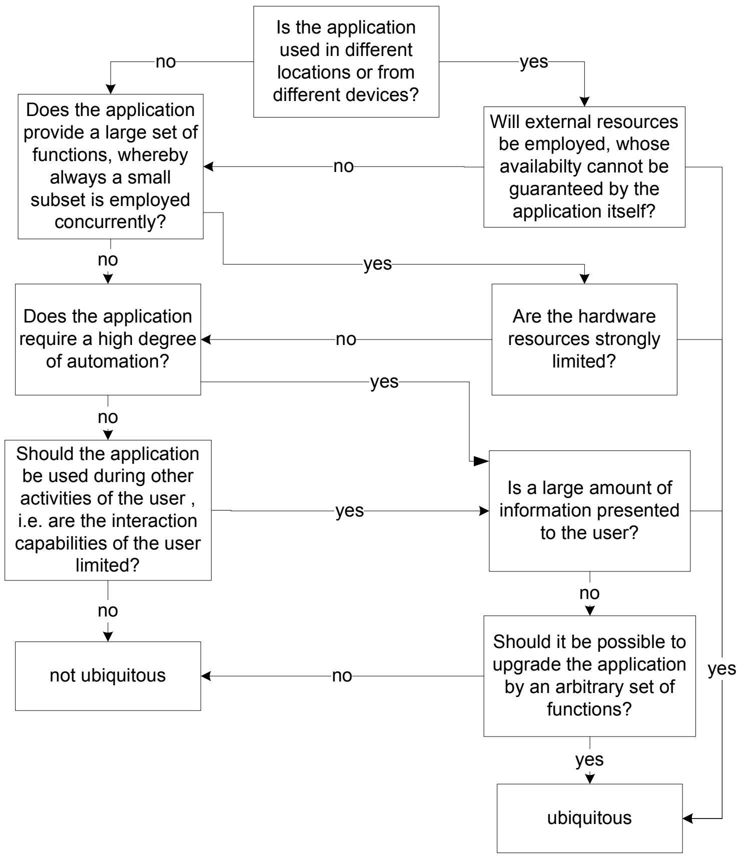
Is the application used in different locations or from different devices?
no
yes
Does the application provide a large set of functions, whereby always a small subset is employed concurrently?
Will external resources be employed, whose availabilty cannot be guaranteed by the application itself?
no
no
yes
Does the application require a high degree of automation?
Are the hardware resources strongly limited?
no
yes
no
Should the application be used during other activities of the user , i.e. are the interaction capabilities of the user limited?
yes
Is a large amount of information presented to the user?
no
no
not ubiquitous
no
Should it be possible to upgrade the application by an arbitrary set of functions?
yes
yes
ubiquitous

Fig. 12. Decision for or against ubiquity and hence explicit adaptivity

If this decision reveals no need for ubiquity, the further system design can be continued with any conventional software engineering process. Otherwise, step 2 will be applied afterwards, namely a scenario and goal driven requirements engineering. The accomplishment of this first step should result in "adequate scenarios" describing the usage of the considered system.

The outcome of the activities described in step 1 is a sufficiently complex scenario description of the application under consideration. In most cases this step also reveals the decision, whether a ubiquitous realization of the application is recommended and hence an explicit handling of context adaptation is necessary.

### *6.3.2 Step 2: Scenario-Based Requirements Engineering*

On basis of scenarios derived from the previous step, a textual-based requirements elicitation is conducted subsequently. The main purpose of this elicitation is to derive an adequate set of requirements, which accurately reflects the intended system behavior and is as consistent and complete as possible. In case of ubiquitous and context-adaptive systems, that are likely to be employed within highly dynamic environments characterized by changing conditions and corresponding user needs, it is indispensable to collect as many information as possible regarding the individual situations, in which the identified requirements occur. Such perceivable information are referred to as *context*.

The requirement of a (context-) adaptive system usually relate to a certain context which characterizes the conditions or situations, under which the requirement is valid. In order to reflect this circumstance, the concept of requirement chunks is consequently augmented by an additional *context* information. Hence every individual chunk consists of a goal or requirement, the associated context in which the former is valid as well as the scenario description, both goal/requirement and context are deduced from. Tab. 2 illustrates such an exemplary requirement chunk taken from a case study of a context-aware task scheduler named Cats.

| | | |
|---|---|---|
| G1: The user wants to be notified *silently* about incoming messages. | C1: The user is currently attending a meeting. | S1: While Jeff is currently attending his 8 o'clock meeting, the scheduler reminds him silently of a consecutively appointment in order to not disturb the participants of the ongoing meeting. |

Table 2
Augmented requirement chunk containing a situation depending goal

As a result of this design step a sufficient number of requirements should be identified together with their corresponding validity conditions represented as context. Due to the notion of requirement chunks and goal refinement graphs, these requirements are already documented in a concise and structured way, which enables the tracing back to underlying analysis documents as the scenario descriptions resulting from step 1. On basis of these requirements and contexts an initial context adaptation model is subsequently designed, which explicitly describes the adaptive behavior of the system under construction. Since the Cawar methodology represents an iterative approach towards the design of context-adaptive systems, the requirements elicited at this stage do not have to be complete. A further elaboration of the requirements namely is conducted by means of an adaptation model in step 4 of the design process.

### *6.3.3 Step 3: First Adaptation Model*

Starting from a set of requirements and associated context data, an initial adaptation model of the considered system can be constructed. Using the requirement chunks introduced in the previous section facilitates the process, although it is not mandatory to document requirements in this way. The purpose of this activity is to express the identified requirements by means of a K-Model. In [10], a couple of design rules are recommended which help to express the textually documented requirements by means of the four service types *sensors*, *context elements*, *interpreters* and *actuators*.

Due to the comfortable graphical notation of the K-Model consisting of only four service types, such models are easy to create and revise. The outcome of this step is a first preliminary K-Model. This model needs by no means to be complete and may also contain adaptation actions and situation adaptors. Both elements enable a more comfortable modeling of the system, since also vague information of the analysis, which are not specified in detail by now, can be integrated into the model. Both elements however have to be refined later on.

### *6.3.4 Step 4: Context Requirements Engineering*

After step 3 an initial adaptation model is available, which usually does not yet cover all aspects of the considered system behavior. Hence the purpose of this step is to fill in the missing elements, thus completing the adaptation model by preferably expressing all requirements and context data identified in the requirements specification. The graphical notation of K-Models facilitates the exposure of gaps previously hidden within the textual requirements specification.

The refinement and elaboration activities described in the following are in principle applied to all elements contained in an adaptation model. However, isolated or unconnected elements which represent the "open ends" of a depicted adaptation thread are of particular interest within this step.

The following table (Tab.3) provides guiding questions for identifying further requirements for situation adaptors (interpreters of sensor data in a K-Model), context elements, interpreters, adaptation actions (context directly fed into actuators) and indications for their possible transformations in a K-Model (see Fig. 9).

Table 3
Guiding questions

| **Requirements for situation adaptors** |
|---|
| – Which information is necessary for identifying a pictured situation in an unambiguous manner? |
| – Are there any further requirements/actions/needs that apply to this situation? |
| – Is it possible to further decompose the associated requirements? |
| – Do any of the associated requirements occur in further depicted situations? |
| – Do the existing situations cover all relevant usage situations, or are there distinguishable situations not considered so far? |
| – Is the correlation between the contexts used for a situation detection and the requirements applying to this situation comprehensible? If not, additional interpretation chains should be appended, which make the considered adaptation thread more readable. |
| **Requirements for context elements** |
| – Can a depicted sensor context be directly used in order to reason about an actual situation, or is additional information necessary? |
| – Is a sensor context appropriate for making an adaptation decision, or is further information necessary? |
| – From which information is a depicted situation or adaptation context composed of? How can this information be deduced from existing contexts? |
| **Requirements for interpreters** |
| – Can a necessary information be supplied by an existing context element? If not, does a combination of existing contexts provide the necessary information? |
| – How does a depicted sensor context contribute to an adaptation decision? |
| – Is it possible to enhance the readability of an adaptation model by appending further interpretation threads? If so, then enhance the model. |
| – Are several contexts relevant for making an adaptation decision? If so, they should be bundled by an interpretation chain and delivered to the responsible actuator afterwards. |
| **Requirements for adaptation actions** |
| – Shall a depicted action also be triggered in further situations? |
| – Which steps are necessary for implementing the action? Should the action be decomposed in a set of (smaller) sub-actions? |
| – Can an action be directly transformed into an adaptation context, or are further interpretations necessary for this task? |

*Application of metrics:* There are in principle two metrics which enable an evaluation of an adaptation model (cf. [9]). An evaluation by means of these metrics may expose some fundamental deficiencies within the considered model. However, in order to produce an expressive evaluation, the considered K-Model must satisfy certain characteristics, which in some cases cannot be assumed a priori. The adaptation metric for instance counts the number of situation adaptors within the model for evaluating the system's ability to adapt to different situations of usage. If for any reason the designer deliberately renounces to use situation adaptors within a certain model (design decision), the metric will certify a non-adaptive model, even if the system may differentiate between several situations very well. Similar constraints apply for the balancing metric, since it builds upon the adaptation metric. We therefore recommend the usage of such metrics with great care and preferably only after subsequent iterations of the design process, when informations concerning the system models have consolidated.

*Limitations of explicit modelling:* An important question concerning the refinement of the K-Model is, which aspects of the system behavior should be explicitly modeled within the adaptation model, and which aspects are encapsulated within the individual services of the model. The obvious answer is, that every aspect which is subject to the frame problem should be modeled explicitly. However, it depends on the designer, the application domain and certain other external factors, which aspects are prone to the frame problem. As a rule of thumb, every service that is prone to change or reconfiguration requests, should be modeled explicitly. Similarly, every aspect that is somehow related to personalization should be modeled explicitly. From a technical point of view, every service that possibly can be realized by external components not contained within the system core (e.g. the mobile device), should also be explicitly modeled in the adaptation model.

The outcome of this step is a refinement/extension of the initial adaptation model. At this stage of maturation, the K-Model should ideally cover all functionalities of the desired system. However, there still might be some obscurities about certain aspects of the system, technical realizations or even coherences of aspects. Such obscurities are treated in subsequent steps and iterations, thus weaving new insights into the model.

### *6.3.5 Step 5: Prototyping*

Prototyping is an appropriate concept for gaining new insights and early feedback concerning the system under construction. Prototypes may implement certain aspects of a system in order to study different technologies, functionalities or even parts of the system model. Therefore, prototyping in combination with software testing serves as an evaluation tool for interfaces (mainly user interfaces) and for providing deeper insights concerning the usability of the considered system. For instance, a prototype may be used to test, if the input/output behavior of some component is correct or if an algorithm fulfills the specified (non-)functional requirements.

In the case of context-adaptive applications, one might also be interested in studying the correlation of certain contexts, their relation to identified situations and their effects on the adaptations of the system. An appropriate prototype may highly contribute to resolving obscurities within a K-Model. In[10], paper prototypes and executable prototypes are discussed in more detail.

### *6.3.6 Step 6: Implementing the System Core*

Due to the development activities carried out in previous steps, a mature adaptation model reflecting all identified requirements and associated contexts is available. The prototypes of step 5 are important indicators for the upcoming implementation of all services included in the K-Model. At first, a decision is made, which of the specified services required by the adaptation model should be realized by components within the system core, and which of them should be moved to the system environment, respectively. Heuristics for guiding this decision are given in the subsequent section.

However, the implementation of each individual component itself is accomplished by conventional concepts of software and systems engineering. The actual implementation is therefore beyond the scope of the Cawar methodology. The only precondition the service fulfilling components must ensure, is to implement the framework specific interfaces for sensors, contexts, interpreters or actuators, respectively. In case the interface cannot be directly implemented due to 3rd party software components, these components must be bound to the framework via intermediary wrapper components (Adapter pattern see [14]) fulfilling the interface imposed by the framework. A short discussion concerning which services should explicitly appear within an adaptation model at all, and which functional behavior should be implicitly encapsulated within a component, respectively, is given in Section 6.3.4.

The initial structuring of the overall system functionality into the system core and its environment is mainly motivated by the lessons learned during the prototyping accomplished in step 5. The decisions are based on the availability of resources and the dependencies between functions. The implementation of certain services may require the addition of further adaptations, resulting in an extension of the adaptation model. This issue is discussed in more detail within the subsequent step. Moreover, the core components implemented during this step should be used for replacing the according simulation components within the associated Cawar prototype.

### *6.3.7 Step 7: Revision of Adaption Design*

To enhance the quality of the context-adaptive system under construction, and hence of the underlying adaptation model, the Cawar methodology requires a certain degree of iterative development. After all services contained in the adaptation model have been implemented as internal or external components, the model usually needs to be iteratively updated. The reason for this revision is, that during the implementation of services often details concerning a better modeling of certain adaptations become visible, which can be conducted in this step. Such modifications result in a modified adaptation model, which in general is augmented by additional, usually more technically oriented adaptations (e.g. discovery of external components, optimizations, autonomic failure recovery etc.). The resulting adaptation model furthermore provides the right stage of maturation for additional evaluations concerning consistency and the appliance of appropriate metrics, respectively. The outlined modifications are described in more detail in [10].

The outcome of this step is a revised K-Model, which includes all secondary adaptations that are necessary due to the implementation of services. The marking of services involved in secondary adaptations is an important input for the subsequent step. In a second iteration, the remaining core system is revised and implemented. Marked services thereby must be implemented in the core system to avoid the emergence of further technical context and associated external service dependencies, respectively.

### *6.3.8 Step 8: Revision of System Core*

In this final step all new services of the modified adaptation model are implemented as internal or external components. The required activities are analog to that already described in step 6: (a) decision for internal or external realization and (b) conventional implementation of components. The only exception is, that all services involved in the technically oriented adaptations added in previous step 7 should be realized *internally*. Otherwise an external implementation of these services can cause single points of failures, which derive from possible external service dependencies.

The revision of the system core is the final step in the overall Cawar methodology. This step produces implementations for all services contained in the adaptation model, which were chosen to be implemented within the system core. Moreover, for all services to be externally realized, we assume the existence of appropriate service fulfilling components. As a result of this step, an augmented prototype is available, in which all simulated components are replaced by actual components of the core system and its environment, respectively.

## 7 Conclusion

The terms *adaptation* and *adaptive systems* are commonly used in order to characterize computer-based systems, which expose a somehow more flexible behavior. However, when it comes to definitively express which part of this behavior constitutes the adaptation and which aspects are adapted, the statements are usually rather vague and the understandings quite differ. Thus, the term adaptation is often used ambiguously. Some approaches provide rather technically motivated definitions or informal categorizations for the notion of adaptation (e.g. [9,31,18]). We aimed at a more general definition, which actually provides a more differentiated viewpoint of this notion and emphasizes its methodological implications.

We argued that adaptation always refers to a certain subject – a human or other technical system – whose interaction with the system is influenced on basis of contextual information usually measured by sensors. W.r.t. the subject we identified three possible perspectives of adaptive system behavior, which differ in the interaction patterns between the subject, the system and its proximate environment.

Adaptive systems are basically interactive systems, and principally can be treated in the same way as the latter. The actual contribution when constructing systems with an adaptive behavior in mind, consists in facilitating the development of complex software systems exposing a certain degree of automation. "Complex" in the sense that those systems fulfill a series of situation dependent requirements concerning their usage. Proposed concepts and programming paradigms such as i) separating the adaptation logic from the actual application logic, ii) modular, component-oriented system design and iii) monitoring and introspection of the system and its environment are examples for enabling technologies in order to realize an adaptive system behavior.

We conclude by identifying relevant research questions, open problems, and blind spots in current research.

*Context Interpretation and Inference:* Which context information is necessary to identify a certain situation of use (e.g. driver is tired)? Which possible situations can be inferred from available contexts?

*Comprehensive Context Model:* What are appropriate concepts for managing the possible contexts available within a complex systems like modern vehicles (e.g. central context repository vs. local contexts in subsystems)?

*Specification of and Reasoning about Adaptive Behavior:* Which modeling techniques are adequate for making the adaptation logic explicit (e.g. system modes)?

*Validation & Verification of adaptive systems:* Adaptive systems promise to release users from certain interactions with the system by reacting context aware. This is often realized by internal reconfigurations of the system. Thereby, a reconfiguration either ensures that the user is provided with a certain functionality despite of changes in the technical infrastructure, such as network connectivity or mobile devices in the proximate environment, or a reconfiguration enables a context-aware reaction by adapting user interface modalities in a situation-dependent fashion. In either case, the verification of such reconfigurations is highly desirable to guarantee safety-critical properties in any context of use. Moreover, certain requirements must only be met in particular contexts. Thus, the verification task needs to take the specific context of a requirement into account.

## References

[1] Agarwal, M., V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia and S. Hariri, *Automate: Enabling autonomic applications on the grid*, in: *Autonomic Computing Workshop* (2003), pp. 48–57.

[2] Broy, M., "Engineering Theories of Software Intensive Systems," Springer Verlag, 2005 pp. 47–81.

[3] Broy, M., *Multifunctional software systems: Structured modeling and specification of functional requirements*, Science of Computer Programming **75** (2010), pp. 1193–1214.
URL http://dx.doi.org/10.1016/j.scico.2010.06.007

[4] Broy, M., C. Leuxner, W. Sitou, B. Spanfelner and S. Winter, *Formalizing the notion of adaptive system behavior*, in: *SAC '09: Proceedings of the 2009 ACM symposium on Applied Computing* (2009), pp. 1029–1033.

[5] Broy, M. and K. Stølen, "Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement," Springer, 2001.

[6] Cheng, B. H., R. de Lemos, H. Giese, P. Inverardi and J. Magee, "Software Engineering for Self-Adaptive Systems: A Research Roadmap," Number 5525 in LNCS, Springer Berlin / Heidelberg, 2009.

[7] Davies, N., J. Landay, S. Hudson and A. Schmidt, *Rapid prototyping for ubiquitous computing*, Pervasive Computing **4** (2005).

[8] Dey, A., "Providing architectural support for building context-aware applications," Ph.D. thesis, College of Computing, Georgia Institute of Technology (2000), director-Gregory D. Abowd.

[9] Fahrmair, M., "Kalibrierbare Kontextadaption für Ubiquitous Computing," Ph.D. thesis, Technische Universität München (2005).

[10] Fahrmair, M., C. Leuxner, W. Sitou and B. Spanfelner, *Adaptation design in ubiquitous computing*, techreport (2008).
URL http://www.in.tum.de/forschung/pub/reports/2008/TUM-I0801.ps.gz;http://www.in.tum.de/forschung/pub/reports/2008/TUM-I0801.pdf.gz

[11] Fahrmair, M., W. Sitou and B. Spanfelner, *An engineering approach to adaptation and calibration*, in: T. R. Roth-Berghofer, S. Schulz and D. B. Leake, editors, *Modeling and Retrieval of Context*, Springer LNCS **3946** (2006), pp. 134 – 147.

[12] Fahrmair, M., W. Sitou and B. Spanfelner, *Unwanted behavior and its impact on adaptive systems in ubiquitous computing*, in: *14th Workshop on Adaptivity and User Modeling in Interactive Systems – LWA/ABIS 2006*, 2006.

[13] Fahrmair, M., W. Sitou and B. Spanfelner, *Seamless development and evaluation of context adaptive systems*, in: *UBICOMP - USE 2007: 9th International Conference on Ubiquitous Computing- Workshop on Ubiquitous Systems Evaluation*, 2007.

[14] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design patterns: abstraction and reuse of object-oriented design," Springer-Verlag, 2002.

[15] Gorlick, M. M. and R. R. Razouk, *Using weaves for software construction and analysis*, in: *ICSE '91: Proceedings of the 13th international conference on Software engineering* (1991), pp. 23–34.

[16] Klein, C., R. Schmid, C. Leuxner, W. Sitou and B. Spanfelner, *A survey of context adaptation in autonomic computing*, in: *Autonomic and Autonomous Systems, 2008. ICAS 2008. Fourth International Conference on*, 2008, pp. 106 –111.

[17] Lieberman, H. and T. Selker, *Out of context: Computer systems that adapt to, and learn from, context*, IBM Systems Journal **39(3-4)** (2000), pp. 617–632.

[18] McKinley, P., S. Sadjadi, E. Kasten and B. Cheng, *Composing adaptive software*, Computer **37** (2004), pp. 56–64.

[19] Medvidovic, N. and M. Mikic-Rakic, *Architectural support for programming in the many*, Technical report, University of Southern California (2001).

[20] Mikic-Rakic, M., N. Mehta and N. Medvidovic, *Architectural style requirements for self-healing systems*, in: *WOSS '02: Proceedings of the first workshop on Self-healing systems* (2002), pp. 49–54.

[21] Mohyeldin, E., M. Dillinger, M. Fahrmair, P. Dornbusch and W. Sitou, *Interworking between link layer and application layer adaptations in a reconfigurable wireless middleware*, in: *15th IEEE International Symposium on Personal, Indoor and Mobile Communications (PIMRC 2004, Barcelona/Spain)*, 2004.

[22] Mohyeldin, E., M. Fahrmair, W. Sitou and B. Spanfelner, *A generic framework for context aware and adaptation behaviour of reconfigurable systems*, in: *The 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications (PIMRC05), 11-14 September 2005, Berlin, Germany*, 2005.

[23] Patterson, D., A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman and N. Treuhaft, *Recovery oriented computing (roc): Motivation, definition, techniques, and case studies*, Technical Report UCB/CSD-02-1175, EECS Department, University of California, Berkeley (2002).

[24] Reiter, R., *The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression* (1991), pp. 359–380.

[25] Roehm, T. and W. Maalej, *Automatically detecting developer activities and problems in software development work*, in: *ICSE'12*, 2012.

[26] Sarter, N., D. Woods and C. Billings, *Automation surprises*, Handbook of Human Factors and Ergonomics **2** (1997), pp. 1926–1943.

[27] Sitou, W., "Requirements Engineering kontextsensitiver Anwendungen," Dissertation, Technische Universität München (2009).

[28] Sitou, W., *Requirements engineering kontextsensitiver systeme*, Softwaretechnik Trends **29** (2009).

[29] Sitou, W. and B. Spanfelner, *Towards requirements engineering for context adaptive systems*, 31st Annual International Computer Software and Applications Conference (COMPSAC 2007) **2** (2007), pp. 593–600.

[30] Taylor, R. N., N. Medvidovic, K. M. Anderson, E. J. W. Jr., J. E. Robbins, K. A. Nies, P. Oreizy and D. L. Dubrow, *A component- and message-based architectural style for GUI software*, Software Engineering **22** (1996), pp. 390–406.

[31] Trapp, M. and B. Schürmann, *On the modeling of adaptive systems*, in: *International Workshop on Dependable Embedded Systems*, 2003.

[32] Weiser, M., *The computer for the 21st century*, Scientific American **265** (September 1991), pp. 94–104.