# A Classification Scheme for Self-adaptation Research

Matthias Rohr*, Simon Giesecke, Wilhelm Hasselbring,
Software Engineering Group, Carl von Ossietzky University of Oldenburg
26111 Oldenburg, Germany

Marcel Hiel, Willem-Jan van den Heuvel, Hans Weigand
InfoLAB, University of Tilburg
5000 LE Tilburg, The Netherlands

## Abstract

The research on self-adaptation is distributed across many sub-disciplines of computer science. This leads to the reinvention of concepts and impairs the ability to establish a structured research program on self-adaptation. This paper contributes a classification scheme that allows to structure self-adaptation research and to compare approaches from different domains. The classification scheme is based on basic concepts that are shared across the research disciplines presented in this paper.

## 1 Introduction

Self-adaptation promises advantages such as increased quality of service, higher flexibility, improved dependability, and lower maintenance costs. On the other hand, self-adaptation can significantly increase the system complexity and the risks for unintended behavior.

It is difficult to define self-adaptation, and therefore it is hard to say whether a system is self-adaptive or not. A typical characteristic of a self-adaptive system is the use of a second conceptual perspective to observe the system or environment, and to adapt the primary operation under certain circumstances. Self-adaptation can be considered as an additional operational unit on top of primary system operation.

Research on self-adaptation in computer science is distributed across many subdisciplines. Each subdiscipline regards self-adaptation from a restricted point of view and fundamental cross-domain research is essentially missing. This leads to the reinvention of concepts and impairs the ability to establish a structured research program on self-adaptation.

This paper contributes a classification scheme that allows to structure self-adaptation research, to compare approaches from different domains, and to identify related research. The classification scheme is based on general concepts of self-adaptation from the study of several related computer science domains, which are discussed within this paper.

This paper is structured as follows: Section 2 briefly discusses computer science subdisciplines that are related to self-adaptation research. Section 3 presents general concepts, goals, and terminology of self-adaptation that are shared among the different domains. In section 4, a classification scheme for self-adaptation research is introduced. The summary and future work follows in section 6.

*This work is supported by the German Research Foundation (DFG), grant GRK 1076/1

## 2 Subdisciplines Related to Self-adaptation Research

A large number of computer science subdisciplines apply and perform research on self-adaptation virtually independent from each other. In this section, some of these subdisciplines are discussed to identify shared basic concepts.

### 2.1 Autonomic Computing

The IBM Autonomic Computing initiative [1] addresses the problem that the number of systems and connected systems are growing rapidly and the accompanying complexity in maintaining them grows with it. The idea behind autonomic computing is that systems become more self-managing and thereby reduce the amount of time and costs put into maintaining them.

Autonomic Computing draws its vision from the autonomous nervous system. The human nervous system is the "controller" in the human body that keeps our vital functions in equilibrium. An example of keeping this balance is that it keeps our blood-sugar level on a certain point and modifies it when necessary.

### 2.2 Self-healing Software Systems

There is no commonly accepted definition of self-healing in the literature. We characterize self-healing in the context of complex software systems as follows: *Self-healing software systems* derive, select, and execute repair operations when unsatisfactory quality of service is monitored and detected. *Self-healing* is the process of maintaining satisfactory quality of service of a primary system by a control component during runtime in the presence of faults.

One typical application domain of self-healing is complex enterprise software systems. For many companies, such systems are mission-critical because they are essential part of the primary business services.

For instance in the domain of complex enterprise software systems, one motivation for self-healing is to reduce repair time by an automation of simple repair tasks. An automatic

repair is usually executed much faster than one performed by an human administrator. As automatic repair is limited to relatively simple problems, it should not be considered as a replacement for an human administrator.

### 2.3 Service-level Agreements

Service Level Agreements (SLAs) provide a established vehicle not only for capturing non-functional requirements but also for monitoring and enforcing them. SLAs are special contractual agreements that encapsulate multiple concerns, and symmetrically reconcile the perspective of service provider and invoker. Besides mutual commitments regarding to-be-delivered services, e.g., timeliness and availability, the SLA should stipulate penalties, contingency plans for exceptional situations, and mechanisms for recovery. Thus, they constitute a instrument to define and enforce Quality of Service (QoS) requirements, that can be formulated in standard SLA-languages, such as WS-Agreement [2].

### 2.4 Software Agents

A commonly used definition for agents is given by Wooldridge [3]: "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives."

From the perspective of self-adaptive software, agents can be regarded a technique for achieving the goals that have been set for self-adaptation. Although a slightly different context is used and therefore a different accent is put on the word self-adaptation. Namely, the reactive and proactive properties of agents are the main characteristics that provide an agent with the ability to adapt or self-adapt. However, the emphasis in the term of self-adaptive software is on the self-part. In agents this is taken more broadly in the sense that the agent could also (try to) change the environment in which they reside.

### 2.5 Ad-hoc Networks

Wireless communication has stimulated research on self-organizing networks. The advantage of these so-called *ad-hoc networks* is that they do not need any pre-established structure.

Self-adaptation and ad-hoc networks are related in that such networks are self-configuring and adaptive. The intent is to realize self-organized networks that have no regulating authority like a service provider. Niemelä and Latvakoski [4] confirm these aspects in their definition of self-organization as "the ability of a system to spontaneously increase its organization without the control of the environment or an encompassing and external system".

## 3 Basic Concepts of Self-Adaptation

As Figure 1 illustrates, a separation can be made between the controller and the adaptable system. A similar distinction is made in control theory between the controller and the controlled object. The controller manages and adapts the adaptable system. The sensors attached to it monitor the possible changes that can occur, which originate from the environment

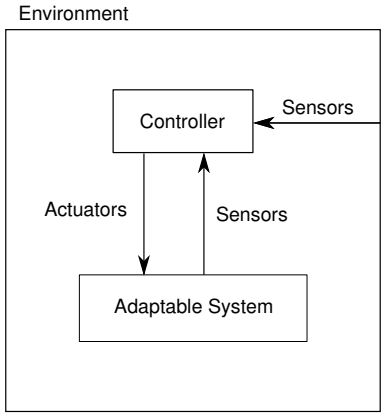


Figure 1: Typical architecture of a self-adaptive system

or from the adaptable system itself. The actuators are used by the controller to apply changes to the adaptable system.

### 3.1 Characterization / Definition

A software entity has the capability of self-adaptation if it can automatically change its structure or behavior, based on observations of its system or environment, with the goal to maintain or improve the Quality of Service.

A slightly different description that does not explicitly consider the environment, is given by Laddaga et al. [5]: "Self adaptive software is software that monitors its own operation, detects faults and opportunities, and repairs or improves itself in response to faults and changes. It effects the improvement by modifying or re-synthesizing its programs and subsystems, using a feedback control-system like behavior."

It should be noted that it is likely that Laddaga et al. [5]'s description refers to 'failures' by the term 'faults' as it is today commonly distinguished. The detection of failures means that some unintended operation is recognized and the term fault is defined as the source of a failure [6]. Usually, the localization and identification of faults is much more challenging than the detection of a failure. For example, finding the reason for a memory leak in the source code is much more difficult than to detect an "out-of-memory" failure.

### 3.2 Typical goals

One possible goal of self-adaptation in software is the reduction of maintenance costs. This goal has also been used in the context of autonomic computing [7]. Software maintenance is largely based on recurring tasks and if these tasks could be automated then this could save much administration effort.

Other typical goals are the improvement of Quality of Service (especially performance), dependability (e.g., reliability, availability and performability [8]) and fault tolerance, design efficiency, and interoperability.

### 3.3 The Adaptation Cycle

Self-adaptation can be structured as a cycle of three activities: Observation, analysis, and adaptation (Figure 2).

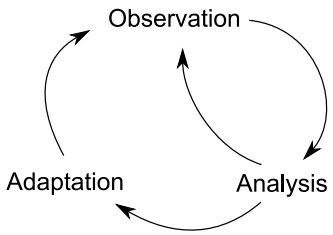


Figure 2: Activities of self-adaptation

#### 3.3.1 Observation

To perform measurements, one has to find a trade-off between information quality (detail and timeliness) and caused performance overhead. Detailed and frequent monitoring, like the recording of the control flow, is often computationally too expensive for normal operation. Additionally, a large overhead of redundant or useless data can make the evaluation more complex.

For a sophisticated maintainable self-adaptive system, it is required to design a monitoring model in a systematic process to ensure that the right measurements are made. The monitoring model should prescribe what, where, and when data is acquired.

**What:** What to be measured can be expressed in terms of metrics (e.g., service response time in ms). A systematic way to derive metrics is the Goal-Question-Metric (GQM) [9] approach.

**Where:** This aspect specifies at which points in the system architecture (or environment) data collection is performed. Both for the interpretation of the monitored data and for the implementation and maintenance of the software, it is important to know where data was and is acquired. Additionally, the precision of the localization of a phenomena that requires adaptation, and hence, the chance of successful adaptation, directly depends on the knowledge about the origin of the data.

**When:** It is specified at which points in time the model is updated.

In practice, monitoring is often implemented manually. This approach is not optimal as the primary application logic will be mixed with monitoring logic, and this reduces the maintainability of the source code. The technique of integrating the monitoring logic into the system (*instrumentation*) should honor that monitoring is a so called *cross-cutting concern* of system behavior. Two major alternative techniques for the instrumentation are middleware interception and source code instrumentation (e.g., via aspect oriented programming (AOP) [10] that allows isolated realization of cross-cutting concerns).

Middleware interception manipulates or extends the middleware platform (for instance the application server, or the virtual machine), so that for example component interaction is automatically observed.

#### 3.3.2 Analysis

The analysis activity summarizes several sub-activities related to the processing of the observations and the identification and selection of adaptation operations. Figure 2 shows an arrow from "Analysis" to "Observation" to indicate that in many cases, no adaptation is required.

A straightforward design for this activity are simple rules (e.g., "if disconnected, then reconnect"). Such rules contain constraints that specify adaptation operations for a particular subset of observations. When an observation violates one of the constraints then a predefined operation will be executed. For the specification of the rules, some approaches from the literature use special first-order predicate logic that are designed to express system behavior (e.g., OCL, Armani). A major problem is that already a few simple rules can result in very complex behavior so that it is not feasible to predict behavior of the self-adaptive system.

Many self-adaptive systems require a more complex analysis during the "Analysis" activity that goes beyond the use of simple rules. Complex decision making could include concepts from artificial intelligence, such as expert systems, ontologies, or data mining.

#### 3.3.3 Adaptation

The adaptation activity is responsible for the execution of the selected adaptation operation(s). Many different kinds of adaptation operations can be distinguished and some might just change single values of the business data, while others affect the structural system architecture.

Changes of the system architecture are often risky. This has two major reasons: (1.) if a non-tested system configuration results from an adaptation, then there might be significant chance that the system will not operate as expected, and (2.) the execution of the operation itself can be technically challenging for the execution environment as it is often desired that the adaptation takes place transparently, this means that the adaptation is executed during runtime, without an interruption of primary system operation.

For many systems with risky adaptation operations, it might not be feasible to test all adaptation scenarios. To provide sufficient dependability in these systems, it is a good strategy to design the system under consideration of adaptation faults. A basic strategy for this would at least supervise the execution of the adaptation operation, test the service, and evaluate whether the adaptation operation achieved its intended benefit or not. More sophisticated approaches could execute the adaptation operations at first on a redundant backup system, or create recovery points to allow for tolerating adaptation faults.

# 4 A Classification Schema for Self-adaptation Research

Figure 3 shows a schema that allows to categorize research approaches according to five major characteristics, which will be discussed in the following subsections.

## 4.1 Origin

The origin is the location of the state change that triggers an adaptation cycle. In biology, this is called the stimulus (state change) and the response (adaptation operation). The stimulus can either originate from inside the system or from the environment.

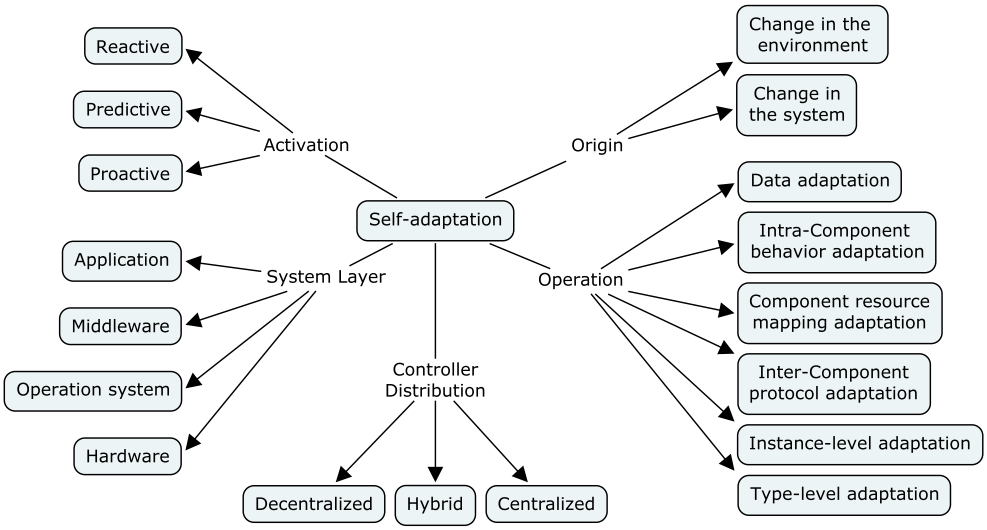


Figure 3: Classification schema for self-adaptation

This separation between system and environment can be made more specific if so desired. In a particular system, it can be beneficial to model elements of the environment explicitly if they are the source of many changes. For instance, in a system with a lot of human interaction, it can be helpful to model the possible changes that an user might desire.

## 4.2 Activation

Regardless of the origin of the change, it is detected through the help of sensors and the system must respond to these changes. We distinguish three different activation types or response types, i.e., reactive, predictive and proactive. To describe the differences between the activation types, we use performance as an illustrative example for the affected system attribute:

**Reactive** If the system adapts only after the performance of the system has dropped under a certain level, then the system can be typed as reactive.

**Predictive** If a certain state or a certain observation occurs before a drop of performance, then the system might adapt to this at once rather than waiting for the actual drop. This is called predictive.

**Proactive** If the system has a normal performance level but the system decides to adapt itself to gain a better performance then this can be considered proactive or possibly goal-directed self-adaptation. This type of adaptation is distinguished from the other two in that it can occur any time. This type of activation is closely related to self-optimization.

## 4.3 System Layer

It is common to distinguish at least four fundamental layers of computing systems: hardware, operation system, middleware, and application. One way to focus and simplify research and development is to limit to one layer and make reasonable assumptions on the other layers. For instance, application programmers are typically concerned with providing a solution on the application layer and assume that a middleware platform supports standard operations, and takes care of the management of lower layers.

Self-adaptation is a general concept that can be applied at any layer, e.g. regarding the adaptation of non-functional system attributes such as timing behavior.

For simplification, the system layer classification should focus on the layers where the adaptation operation is applied. The other parts of the self-adaptation, especially observation, are related to one or more system layers, as well, and additionally, might even operate on different layers. For instance, application behavior could be monitored on the application layer, while the reconfiguration follows on the hardware layer. However, the adaptation activity requires usually more effort during development and operation than just observation. Therefore, it might be required to include the system layers of other activities than adaptation into the classification as well, but only with a lower priority.

## 4.4 Operation

We define a classification scheme for (self-)adaptation operations by combining and extending the dynamism hierarchy of Cuesta et al. [11] with the classification scheme of McKinley et al. [12]. Basically, the operation classification dimension uses an Architectural Description Language (ADL) viewpoint on the adaptation operation. Multiple component instances can be belong to a component type, and the implementation of a component instance can change without effecting the associated component type.

We distinguish six types of adaptation operations according to the architectural perspective:

**Data adaptation** Data values are changed in the system data model without effecting the control flow within or between the system components

**Intra-component behavior adaptation** Changes the behavior (control flow or quality of service) of a component without directly affecting component execution sequences

**Component resource mapping adaptation** Changes the association of software components and resources (data source are here not considered as resources)

**Inter-component protocol adaptation** Changes the dynamic communication between components in a fixed structure

**Instance-level adaptation** Changes the structure by adding or removing component instances (but not component types)

**Type-level adaptation** Changes the component types, which allows to define new component types (the component dependency graph is affected)

### 4.5 Controller Distribution

Three controller distribution styles can be distinguished according to the architectural localization of self-adaptation: centralized, decentralized, and hybrid. The controller distribution style is centralized if the adaptation is controlled and executed from a central point in the system. A decentralized system has no such single point of failure and executes self-adaptation (all activities) locally. Most software agent architectures would be considered as decentralized. Hybrid approaches combine the advantages of a centralized and decentralized architecture by applying parts of the activities locally and other parts globally.

## 5 Related Work

The most related work for the self-adaptation classification scheme can be found in [11, 12]. However, this work only addresses adaptation operations, and this is just one activity of self-adaptation as discussed above (Section 3.3).

McKinley et al. [12] presented a taxonomy and survey on so-called compositional adaptation. Compositional adaptation can be understood as what we have termed architectural reconfiguration of the system. Therefore, it is a subclass of the adaptation operations that are addressed by our work. Their classification scheme use the three perspectives of how, where, when computational adaptation is applied. The taxonomy of McKinley et al. [12] is better suited than our scheme if technical details are of importance (and only adaptation operations are addressed), as we use the implementation- and fine-design-independent view of architectural descriptions. Their "where" dimension distinguishes several system layers, just as we do. They use the more detailed system level hierarchy of Schmidt and Buschmann [13] which distinguishes several middleware layers.

## 6 Conclusions and Future Work

In this paper we have introduced a scheme that allows to classify self-adaptation research. The scheme was developed based on the basic concepts that are shared in the several subdisciplines, which have been discussed above.

The benefit of the classification scheme is that it helps to connect and structure research on the basic principles of self-adaptation and enables to compare the vast number of approaches presented in the literature.

Future work will apply the scheme to classify existing approaches and research from the literature and industry.

## References


[1] Paul Horn. Autonomic computing: IBM's perspective on the state of information technology. Manifesto, IBM Research, October 2001.

[2] A. Andrieux, K. Czajkowski, A. Dan, K. Keahey, H. Ludwig, J. Pruyne, J. Rofrano, S. Tuecke, and M. Xu. Web Services Agreement Specification, Version 1.1, GGF GRAAP Working Group, May 2004.

[3] Michael J. Wooldridge. *Introduction to Multiagent Systems*. John Wiley & Sons, Inc., New York, NY, USA, 2001.

[4] Eila Niemelä and Juhani Latvakoski. Survey of requirements and solutions for ubiquitous software. In *Proceedings of the 3rd international conference on Mobile and ubiquitous multimedia (MUM'04)*, pages 71–78, New York, NY, USA, 2004. ACM Press. doi:10.1145/1052380.1052391.

[5] Robert Laddaga, Paul Robertson, and Howard E. Shrobe. Results of the second international workshop on self-adaptive software. In *Second International Workshop on Self-Adaptive Software (IWSAS'01)*, volume 2614 of *Lecture Notes in Computer Science*, pages 281–290. Springer, 2001.

[6] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. doi:10.1109/TDSC.2004.2.

[7] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *Computer*, 36(1):41 – 50, January 2003. doi:10.1109/MC.2003.1160055.

[8] J.F. Meyer. Performability evaluation: where it is and what lies ahead. In *Proceedings of the International Symposium Computer Performance and Dependability*, pages 334–343. IEEE, April 1995. doi:10.1109/IPDS.1995.395818.

[9] Victor R. Basili. The role of experimentation in software engineering: past, current, and future. In *Proceedings of the 18th international conference on Software engineering (ICSE'96)*, pages 442–449, Washington, DC, USA, 1996. IEEE.

[10] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[11] Carlos E. Cuesta, Pablo de la Fuente, and Manuel Barrio-Solórzano. Dynamic coordination architecture through the use of reflection. In *Proceedings of the 2001 ACM symposium on Applied computing (SAC'01)*, pages 134–140, New York, NY, USA, 2001. ACM Press. doi:10.1145/372202.372298.

[12] Philip K. McKinley, Seyed Masoud Sadjadi, Eric P. Kasten, and Betty H. C. Cheng. A taxonomy of compositional adaptation. Technical Report MSU-CSE-04-17, Department of Computer Science, Michigan State University, East Lansing, Michigan, May 2004.

[13] Douglas C. Schmidt and Frank Buschmann. Patterns, frameworks, and middleware: their synergistic relationships. In *Proceedings of the 25th International Conference on Software Engineering (ICSE '03)*, pages 694–704, Washington, DC, USA, 2003. IEEE Computer Society.