2 Foundations and Related Work

This dissertation is related to multiple research and practice areas of software engineering. In this chapter, the necessary *foundations* of these related areas are introduced. We give an overview of the *related work*, as well.

The main goal of this chapter is to provide readers with the essential background knowledge needed to follow the upcoming chapters. The basics introduced here sketch the background of the author of this thesis which shall help to follow some of the decisions made in the remainder of this work. Experts in the field may skip parts, of course.

The topic introduced in this thesis is related to a broad area of research. Accordingly, we introduce (i) *adaptive software*, (ii) *software product lines*, (iii) *modeling*, (iv) *software language engineering* and (v) *software components* in the remainder of this chapter. We define special terminology the first time it is used. A *glossary* of terms (starting on page 293) and a list of *acronyms* (starting on page 305) are attached to this thesis. This chapter ends with a short *summary*.

2.1 Adaptive Software

The main concept and driving idea especially behind *Self-Adaptive Software* (SAS) [96, 111] and likewise behind *Autonomic Computing* (AC) [71, 86] is to achieve the automation of tedious administration and maintenance tasks. Specifically, software shall take active responsibility for its own *robustness* [97] by adapting its own state and behavior at runtime in response to observed changes (i) in its *context* as well as (ii) in its *self*. While the context covers especially the *operative environment*, the self denotes the whole *body of software* as it is implemented (e.g., logically organized into a stack of layers) [120].

M. Derakhshanmanesh, *Model-Integrating Software Components,*
DOI 10.1007/978-3-658-09646-5_2, © Springer Fachmedien Wiesbaden 2015

Many researchers use the terms self-adaptive, autonomic computing and self-managing interchangeably [76], while some see SAS as the more general area [120]. There is also a community that works on *Organic Computing* (OC) [112, 144]. In the scope of this dissertation, we handle SAS and Autonomic Computing (AC) as two strongly related research areas that target similar goals. Thus, we refer to them collectively as *adaptive software* or "software with adaptivity" (see Definition 2.1) in this thesis.

Definition 2.1: adaptivity

Adaptivity is the ability of a system to make controlled, i.e., meaningful, changes to its own states and behavior to suit changing conditions at runtime.

Adaptivity management (see Definition 2.2) is a very wide and complex topic. The subtopics include *technical* (How to develop adaptive software and its critical parts such as the feedback loop? How to plan and foresee the set of actions that will be most successful in new environments?) and *non-technical* challenges and issues (How to ensure that the adaptive software acts in compliance with the law?).

Definition 2.2: adaptivity management

Adaptivity management is the process of engineering software with adaptivity and controlling its evolution.

The motivation for adaptive software stems from various issues related to engineering *complexity*, *flexibility*, *robustness*, continuous *on-time evolution* and the overall desire for *short turn-around times* required to deal with ever changing customer needs. For example, software is commonly built to operate within (strongly) restricted bounds. Components and their interfaces are being developed for *anticipated* use cases and scenarios. Once placed outside of the context

it was engineered for, software tends to fail miserably and intervention of humans is required for such *unanticipated* situations.

In practice, it is common to perform necessary maintenance tasks on software *offline* and to deploy a new version, eventually. This time-consuming process results in high costs and also in delays during which the system cannot operate and *Service Level Agreements (SLAs)* [91] are violated. The promise of adaptive software is to provide proven methods, tools and techniques that support the automation of large portions of maintenance tasks *online*, i.e., during operation time (runtime). Furthermore, the vision is that software shall be *fault-tolerant* and stable, with minimal human intervention.

Relying on a mix of *sensing* and *effecting* mechanisms that may be implemented by a combination of hardware and software, an *adaptation manager* (see Definition 2.3) is capable of controlling the coupled *managed software*.[1]

Definition 2.3: adaptation manager

An *adaptation manager* is a subject, e.g., a software component, that performs the process of adapting.

Often, controlling is also referred to as *adapting* (see Definition 2.4).

Definition 2.4: adapting

Adapting is a process that operates on a software system to adjust it to varying requirements at runtime by executing adaptations.

Definition 2.5: adaptation

An *adaptation* is a planned sequence of actions that makes a system suitable for a new condition.

[1] Autonomic computing names them *autonomic manager* and *managed element*. In control theory, the related terms are *controller* and *controlled plant*.

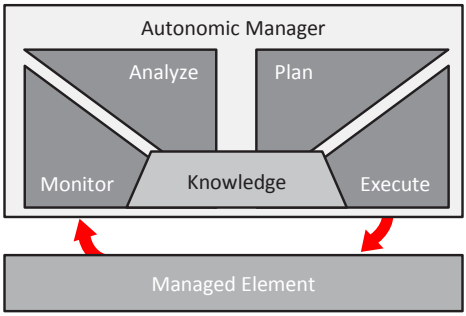

Figure 2.1: IBM's MAPE-K Loop

A common sequence of steps involved is described by the popular *MAPE-K loop* [86]. An illustration is given in Figure 2.1. It consists of (i) *monitoring* changes in the environment, (ii) *analyzing* them for their relevance, (iii) *planning* possible adaptation steps and, finally, (iv) *executing* an adaptation strategy. Additionally, this process is supported by (v) *knowledge*, which is often available in the form of a *self-representation* that can be achieved by following the *reflective architecture pattern* [20].

When designing an adaptation manager, it is common to distinguish between *internal* and *external* approaches [120]. While internal solutions merge the individual steps with the software's (application) logic, external solutions avoid this tight coupling and propose to develop the adaptation manager separately and to connect them with the (existing) application software. This connection is usually established via traditional interfaces, event systems or by injecting code with Aspect-Oriented Programming (AOP) techniques, for instance. Most research seems to focus and rely on the external approach [120, 147].

Most proposed approaches to engineering SAS have in common that they achieve dynamic change by managed, incremental change at the structural level of software, i.e., SAS is also an *architectural challenge* [93]. In Section 2.5, we describe the essentials of software architecture in terms of components and connectors.

2.1.1 Self-* Properties

Software adaptivity is based on a set of properties that facilitate (i) the *observation* of operating state transitions and (ii) the *modification* of the managed application's behavior at runtime [128].[2]

Depending on the domain of the managed software and depending on its requirements, adaptive software needs to possess certain properties to fulfill its requirements, i.e., there are different kinds of adaptivity. This class of properties is widely known as *self-* properties*[3] [6] and can be categorized. Each self-* property focuses on a specific set of system capabilities.

Horn describes eight of such self-* properties that may characterize an autonomic system [71]. In [120], the authors propose a hierarchical view of these properties as illustrated in Figure 2.2. According to this hierarchy, self-adaptiveness and self-organizing are *general* properties of adaptive software. These are further decomposed into *major* and *primitive* properties. This categorization can serve as a starting point for understanding the required and desired properties of adaptive software to be built. In the following, we give an introduction to this hierarchy.

General Level

The *general level* contains global properties of adaptive software such as self-managing, self-governing, self-maintenance [86], self-control [90] and self-evaluating [98]. These terms support the discussion of adaptivity at an abstract level, i.e., without explicitly mentioning concrete capabilities.

Major Level

The *major level* contains all of the four self-* properties introduced in [71]. These were motivated by analogies from biology, where, for

[2] The section on self-* properties is partially based on text from the author's master's thesis [37].

[3] The "*" symbol in "self-*" is a placeholder for different properties.

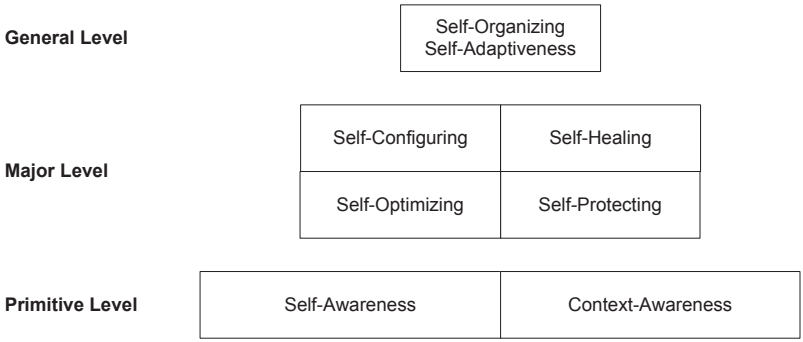

Figure 2.2: Hierarchy of Self-* Properties (Derived from [120])

instance, the autonomous nervous system monitors and controls the heart rate and body temperature [86]. Self-* properties at this level of abstraction can be used to identify and specify the needed adaptivity properties for a concrete software system.

- *Self-configuring* is the ability of adaptive software to adjust its own configuration. Such an adjustment may include installing, updating, integrating, composing and decomposing existing or new software elements.
- *Self-healing* is the ability of adaptive software to discover, diagnose and react to disruptions. Adaptive software with such capabilities is able to detect potential problems and can take proper actions accordingly to prevent failure.
- *Self-optimizing* is the ability of adaptive software to manage performance (e.g., end-to-end response time and throughput) and resource allocation (e.g., CPU and memory consumption).
- *Self-protecting* is the ability of adaptive software to protect itself against threats by detecting security breaches and by recovering from their effects.

Primitive Level

The *primitive level* contains the fundamental properties needed for adaptive software to enable any kind of reasonable adaptation.

- *Self-awareness* refers to the ability of adaptive software to be aware of its *self* (i.e., states and behaviors) based on self-monitoring [68].
- *Context-awareness* refers to the ability of adaptive software to be aware of its *context* (i.e., the operating environment) based on context-monitoring [113].

2.1.2 Selected Publications

The foundations of adaptive software – as needed for this thesis – were described. For a comprehensive list of publications on adaptive software, please check some of the existing surveys [25, 33, 46, 76, 115, 120, 149] and the programs of:

- the International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)[4],
- the International Conference on Cloud and Autonomic Computing (CAC)[5] and
- the Self-Adaptive and Self-Organizing Systems (SASO)[6] community.

Subsequently, we give a summary of *selected publications*[7] from different subareas of the field. The idea is to give the interested reader some more details and background by introducing interesting recently published work.

[4] `http://www.self-adaptive.org/` (accessed July 6th, 2014)
[5] `http://www.autonomic-conference.org/` (accessed July 6th, 2014)
[6] `http://www.saso-conference.org/` (accessed July 6th, 2014)
[7] Please note that the author of this thesis is "influenced" by the SEAMS community.

Subtopics Related to Applications

Pasquale et al. [114] describe the application of SAS to solve the challenge of protecting valuable assets (physical or virtual) within an organization. This area of research is called adaptive security. The assumption is that a protecting system based on hardware and software exists. Because assets change over time, the appropriate countermeasures against attacks need to be decided on at runtime, too. The contributions are a modeling notation for representing assets, the use of models of security requirements and monitoring functions as well as the description of a set of scenarios in the domain of smart grids.

Pasquale et al. combine a domain-specific *model of assets* (containing rooms, energy, CCTV camera, etc.) with an *extended goal model* [119] that also includes vulnerabilities. Each asset has a value and these are recomputed on change based on defined monitoring functions. Countermeasures are injected into the source code using AOP techniques; in this specific work, the AspectJ [8] is used. Different kinds of countermeasures and their implementation are presented, including one solution that waits for the input of a human administrator ("human-in-the-loop").

Ingolfo and Souza [79] propose to use mechanisms from requirements-based approaches to building SAS in order to achieve *regulatory compliance* for software systems. The challenge is to cope especially with changes and variability in the law. For instance, the allowed age of drivers varies across the US from state to state and assisting solutions must adapt accordingly. Google's driver-less car is taken as an example of an autonomous system that shall comply to applicable law. For example, it shall respect stop signs.

The *Zanshin* approach [124] is used to illustrate how to implement an adaptive system using a requirements model consisting of *awareness requirements* (definition of what should be monitored) and *evolution requirements* (definition of actions to face detected failure).

[8] `http://eclipse.org/aspectj/` (accessed July 6th, 2014)

Ingolfo and Souza conclude that, based on their preliminary approach, a piece of law (text) can be systematically converted into a model that can be used by a SAS for decision making. Changes in the law are faced in terms of automated reconfiguration of the software. The SAS is able to identify breaches of the law and to search for variations in the law(s) it can fulfill, essentially resulting in a reconfigured system.

Huang and Knottenbelt [74] propose a framework for the realization of *self-adaptive containers* to support software developers building resource-efficient applications. Assuming that an execution environment exists, these low-level containers (e.g., List, Set, Stack, Queue) adapt their use of algorithms to provide an implementation that satisfies a set of Service Level Objective (SLO) [127] best.

These containers consist of two parts: an API and a *Self-Adaptive Unit*. While the first one is subdivided into *configuration interfaces* and *operation interfaces*, the latter consists of an SLO store, observer analyzer, adaptor and an execution unit. They perform a feedback loop. SLOs are defined by using Web Service Level Agreement (WSLA) [85] in XML format and containers are implemented in C++. Internally, a combination of (probabilistic) data structures is used.

Based on a case study where the author's implementation is compared against C++'s standard implementation of the respective containers, Huang and Knottenbelt demonstrate that adaptivity can also support the needs of software developers as the burden of manual optimization of the used algorithms is removed from their shoulders (in this specific example).

Subtopics Related to Architecture

Weyns et al. [147] investigate if the choice of building SAS with an external control loop (in contrast to an internal control loop) really improves software design or not. The claim that an external control mechanism (i.e., an adaptation manager) is the better solution from

an engineering perspective was especially propagated by *Garlan et al.* [61] and the proposed Rainbow[9] framework [26].

Weyns et al. present their results from a controlled experiment carried out with 24 final-year Students of a Master in Software Engineering program at Linnaeus University in Sweden. Even though external and internal solutions seem to be roughly the same size, external mechanisms simplify the software's design in terms of its control flow complexity. The outcome of the experiment is that, indeed, external feedback loops are the better choice for engineering SAS.

Cámara et al. [21] report on their findings when integrating the Rainbow framework [61] with an existing industrial-scale software solution for monitoring and managing networks of devices. The primary goal is to investigate if and how new frameworks for adaptivity can be integrated with existing industrial solutions (here called *legacy system*). The associated effort is measured, as well. The overall assumption is that a solution like Rainbow can enhance existing software.

The authors describe the changes necessary for a certain part of the Acquisition and Control Service (DCAS) under inspection and illustrate the newly introduced adaptation mechanisms and their realization using the framework's capabilities of placing probes, effectors and expressing adaptation strategies.

Cámara et al. conclude that the Rainbow-based solution performs adaptation better than the existing solution. Moreover, it was not too complicated to integrate both software systems with the exception of an initial learning effort added. Once the integration has been completed, evolution of the system takes minutes, not hours. This facilitates maintenance which is the primary effort associated with software over its full life-cycle [19].

Subtopics Related to Evaluation

Weyns et al. [146] discuss that no systematic study has been performed to check if the claims associated with SAS are actually valid or not.

[9] `http://www.cs.cmu.edu/~able/research/rainbow/` (accessed July 6th, 2014)

http://www.springer.com/978-3-658-09645-8

Model-Integrating Software Components
Engineering Flexible Software Systems
Manesh, M.
2015, XXI, 333 p. 43 illus., Softcover
ISBN: 978-3-658-09645-8