Engineering Self-Adaptive Systems through Feedback Loops

Yuriy Brun[1], Giovanna Di Marzo Serugendo[2], Cristina Gacek[3], Holger Giese[4], Holger Kienle[5], Marin Litoiu[6], Hausi Müller[5], Mauro Pezzè[7], and Mary Shaw[8]

[1] University of Southern California, Los Angeles, CA, USA
ybrun@usc.edu
[2] Birkbeck, University of London, London, UK
dimarzo@dcs.bbk.ac.uk
[3] University of Newcastle upon Tyne, Newcastle upon Tyne, UK
cristina.gacek@ncl.ac.uk
[4] Hasso Plattner Institute at the University of Potsdam, Germany
holger.giese@hpi.uni-potsdam.de
[5] University of Victoria, British Columbia, Canada
{kienle,hausi}@cs.uvic.ca
[6] York University and IBM Canada Ltd., Canada
marin@ca.ibm.com
[7] University of Milano Bicocca, Italy and University of Lugano, Switzerland
mauro.pezze@unisi.ch
[8] Carnegie Mellon University, Pittsburgh, PA, USA
mary.shaw@cs.cmu.edu

Abstract. To deal with the increasing complexity of software systems and uncertainty of their environments, software engineers have turned to self-adaptivity. Self-adaptive systems are capable of dealing with a continuously changing environment and emerging requirements that may be unknown at design-time. However, building such systems cost-effectively and in a predictable manner is a major engineering challenge. In this paper, we explore the state-of-the-art in engineering self-adaptive systems and identify potential improvements in the design process.

Our most important finding is that in designing self-adaptive systems, the feedback loops that control self-adaptation must become first-class entities. We explore feedback loops from the perspective of control engineering and within existing self-adaptive systems in nature and biology. Finally, we identify the critical challenges our community must address to enable systematic and well-organized engineering of self-adaptive and self-managing software systems.

1 Introduction

The complexity of current software systems and uncertainty in their environments has led the software engineering community to look for inspiration in diverse related fields (e.g., robotics, artificial intelligence, control theory, and biology) for new ways to design and manage systems and services [1,2,3,4]. In this endeavor, the capability of the system to adjust its behavior in response to the environment in the form of self-adaptation has become one of the most promising research directions. The “self” prefix indicates that the systems decide autonomously (i.e., without or with minimal interference) how to adapt or organize to accommodate changes in their contexts and environments. While some self-adaptive system may be able to function without any human intervention, guidance in the form of higher-level objectives (e.g., through policies) is useful and realized in many systems.

The landscapes of software engineering domains and computing environments are constantly evolving. In particular, software has become the bricks and mortar of many complex systems (i.e., a system composed of interconnected parts that as a whole exhibits one or more properties (behaviors among the possible properties) not obvious from the properties of the individual parts). The hallmarks of such complex or ultra-large-scale (ULS) systems [5] are self-adaptation, self-organization, and emergence [6]. Engineers in general, and software engineers in particular, design systems according to requirements and specifications and are not accustomed to regulating requirements and orchestrating emergent properties. Ottino argues that the landscape is bubbling with activity and engineers should be at the center of these developments and contribute new theories and tools [6].

In order for the evolution of software engineering techniques to keep up with these ever-changing landscapes, software engineers must innovate in the realm of building, running, and managing software systems. Software-intensive systems must be able to adapt more easily to their ever-changing surroundings and be flexible, fault-tolerant, robust, resilient, available, configurable, secure, and self-healing. Ideally, and necessarily for sufficiently large systems, these adaptations must happen autonomously. The research community that has formed around self-adaptive systems has already generated many encouraging results, helping to establish self-adaptive systems as a significant, interdisciplinary, and active research field.

Self-adaptive systems have been studied within the different research areas of software engineering, including requirements engineering [7], software architecture [8,9], middleware [10], and component-based development [11]; however, most of these initiatives have been isolated. Other research communities that have also investigated self-adaptation and feedback from their own perspectives are even more diverse: control theory, control engineering, artificial intelligence, mobile and autonomous robots, multi-agent systems, fault-tolerant computing, dependable computing, distributed systems, autonomic computing, self-managing systems, autonomic communications, adaptable user interfaces, biology, distributed artificial intelligence, machine learning, economic and financial systems, business and military strategic planning, sensor networks, or pervasive and ubiquitous computing. Over the past decade several self-adaptation-related application areas and technologies have grown in importance. It is important to emphasize that in all these initiatives software has become the common element that enables the provision of self-adaptability. Thus, it is imperative to investigate systematic software engineering approaches for developing self-adaptive systems, which are—ideally—applicable across multiple domains.

Self-adaptive systems can be characterized by how they *operate* or how they are *analyzed*, and by multiple dimensions of properties including *centralized* and *decentralized*, *top-down* and *bottom-up*, *feedback latency* (slow vs. fast), or *environment uncertainty* (low vs. high). A top-down self-adaptive system is often centralized and operates with the guidance of a central controller or policy, assesses its own behavior in the current surroundings, and adapts itself if the monitoring and analysis warrants it. Such a system often operates with an explicit internal representation of itself and its global goals. By analyzing the components of a top-down self-adaptive system, one can compose and deduce the behavior of the whole system. In contrast, a cooperative self-adaptive system or self-organizing system is often decentralized, operates without a central authority, and is typically composed bottom-up of a large number of components that interact locally according to simple rules. The global behavior of the system *emerges* from these local interactions. It is difficult to deduce properties of the global system by analyzing only the local properties of its parts. Such systems do not necessarily use internal representations of global properties or goals; they are often inspired by biological or sociological phenomena.

Most engineered and nature-inspired self-adaptive systems fall somewhere between these two extreme poles of self-adaptive system types. In practice, the line between these types is rather blurred and compromises will often lead to an engineering approach incorporating techniques from both of these two extreme poles. For example, ULS systems embody both top-down and bottom-up self-adaptive characteristics (e.g., the Web is basically decentralized as a global system, but local sub-webs are highly centralized or server farms are both centralized and decentralized) [5].

Building self-adaptive software systems cost-effectively and in a predictable manner is a major engineering challenge. New theories are needed to accommodate, in a systematic engineering manner, traditional top-down approaches and bottom-up approaches. A promising starting point to meet these challenges is to mine suitable theories and techniques from control engineering and nature and to apply those when designing and reasoning about self-adaptive software systems. Control engineering emphasizes feedback loops, elevating them to first-class entities [12,13]. In this paper we argue that feedback loops are also essential for understanding all types of self-adaptive systems.

Over the years, the discipline of software engineering strongly emphasized the static architecture of a system and, to a certain extent, neglected the dynamic aspects. In contrast, control engineering emphasized the dynamic feedback loops embedded in a system and its environment and neglected the static architecture. A notable exception is the seminal paper by Magee and Kramer on dynamic structure in software architecture [14], which formed the foundation for many subsequent research projects [9,15,16,17]. However, while these research projects realized feedback systems, the actual feedback loops were hidden or abstracted.

Feedback loops have been recognized as important factors in software process management and improvement or software evolution. For example, the feedback loops at every stage in Royce's waterfall model [18] or the risk feedback loop in Boehm's spiral model [19] are well known. Lehman's work on software evolution showed that "the software process constitutes a multilevel, multiloop feedback system and must be treated as such if major progress in its planning, control, and improvement is to be achieved." Therefore, any attempt to make parts of this "multiloop feedback system" self-adaptive necessarily also has to consider feedback loops [20].

With the proliferation of self-adaptive software systems, it is imperative to develop theories, methods and tools around feedback loops. Mining the rich experiences and theories from control engineering as well as taking inspiration from nature and biology where we can find systems that adapt in rather complex ways, and then adapting and applying the findings to software-intensive self-adaptive systems is a most worthwhile and promising avenue of research.

In the remainder of this paper, we therefore investigate *feedback loops* as a key aspect of engineering self-adaptive systems. Section 2 outlines basic principles of feedback loops and demonstrates their importance and potential benefits for understanding self-adaptive systems. Sections 3 and 4 describe control engineering and biologically inspired approaches for self-adaptation. In Section 5, we present selected challenges for the software engineering community in general and the SEAMS community in particular for engineering self-adaptive computing systems.

2 The Role of Feedback Loops

Self-adaptation in software-intensive systems comes in many different guises. What self-adaptive systems have in common is that design decisions are moved towards runtime to control dynamic behavior and that an individual system reasons about its state and environment. For example, keeping web services up and running for a long time requires collecting information that reflects the current state of the system, analyzing that information to diagnose performance problems or to detect failures, deciding on how to resolve the problem (e.g., via dynamic load-balancing or healing), and acting to effect the planning decisions made.

Feedback loops provide the generic mechanism for self-adaptation. Positive feedback occurs when an initial change in a system is reinforced, which leads toward an amplification of the change. In contrast, negative feedback triggers a response that counteracts a perturbation. Further, natural environments with synergistic and antagonistic relationships between components sometimes produce more complex forms of feedback loops that can neither be classified as positive nor negative feedback.

2.1 Generic Feedback Loop

A feedback loop typically involves four key activities: collect, analyze, decide, and act. Sensors or probes collect data from the executing system and its context

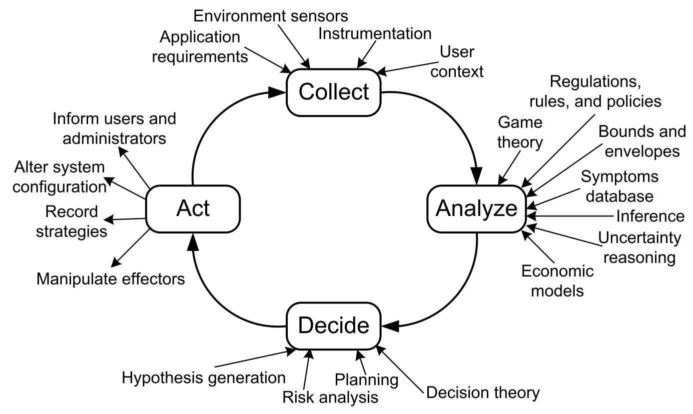


Fig. 1. Autonomic control loop [21]

about its current state. The accumulated data are then cleaned, filtered, and pruned and, finally, stored for future reference to portray an accurate model of past and current states. The diagnosis then analyzes the data to infer trends and identify symptoms. Subsequently, the planning attempts to predict the future to decide on how to act on the executing system and its context through actuators or effectors.

This generic model of a feedback loop, often referred to as the autonomic control loop as depicted in Figure 1 [21], focuses on the activities that realize feedback. This model is a refinement of the AI community's sense-plan-act approach of the early 1980s to control autonomous mobile robots [22,23]. While this model provides a good starting point for our discussion of feedback loops, it does not detail the flow of data and control around the loop. However, the flow of control among these components is unidirectional. Moreover, while the figure shows a single control loop, multiple separate loops are typically involved in a practical system.

When engineering a self-adaptive system, questions about these properties become important. The feedback cycle starts with the *collection* of relevant data from environmental sensors and other sources that reflect the current state of the system. Some of the engineering questions that need be answered here are: What is the required sample rate? How reliable is the sensor data? Is there a common event format across sensors? Do the sensors provide sufficient information for system identification?

Next, the system *analyzes* the collected data. There are many approaches to structuring and reasoning about the raw data (e.g., using models, theories, and rules). Some of the applicable questions here are: How is the current state of the system inferred? How much past state may be needed in the future? What data need to be archived for validation and verification? How faithful will the model be to the real world and whether an adequate model can be obtained from the available sensor data? How stable will the model be over time?

Next, a *decision* must be made about how to adapt the system in order to reach a desirable state. Approaches such as risk analysis help in choosing among various alternatives. Here, the important questions are: How is the future state of the system inferred? How is a decision reached (e.g., with off-line simulation, utility/goal functions, or system identification)? What are the priorities for self-adaptation across multiple feedback loops and within a single feedback loop?

Finally, to implement the decision, the system must *act* via available actuators or effectors. Important questions that arise here are: When should and can the adaptation be safely performed? How do adjustments of different feedback loops interfere with each other? Do centralized or decentralized feedback help achieve the global goal? An important additional applicable question is whether the control system has sufficient command authority over the process—that is, whether the available actuators or effectors are sufficient to drive the system into the desired directions.

The above questions—and many others—regarding the feedback loops should be explicitly identified, recorded, and resolved during the development of a self-adaptive system.

2.2 Feedback Loops in Control Engineering

An obvious way to address some of the questions raised above is to draw on control theory. Feedback control is a central element of control theory, which provides well-established mathematical models, tools, and techniques for analysis of system performance, stability, sensitivity, or correctness [24,25]. The software engineering community in general and the SEAMS community in particular are exploring the extent to which general principles of control theory (i.e., feedforward and feedback control, observability, controllability, stability, hysteresis, and specific control strategies) are applicable when reasoning about self-adaptive software systems.

Control engineers have invented many variations of control and adaptive control. For many engineering disciplines, these types of control systems have been the bread and butter of their designs. While the amount of software in these control systems has increased steadily over the years, the field of software engineering has not embraced feedback loops as a core design element. If the computing pioneers and programming language designers were control engineers—instead of mathematicians—by training, modern programming paradigms might feature process control elements [26].

We now turn our attention to the generic data and control flow of a feedback loop. Figure 2 depicts the classical feedback control loop featured in numerous control engineering books [24,25]. Due to the interdisciplinary nature of control theory and its applications (e.g., robotics, power control, autopilots, electronics, communication, or cruise control), many diagrams and variable naming conventions are in use. The system's goal is to maintain specified properties of the output, y_p, of the process (also referred to as the plant or the system) at or sufficiently close to given reference inputs u_p (often called set points). The process output y_p may vary naturally; in addition external perturbations d may disturb

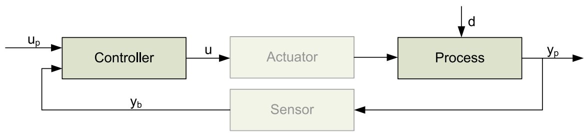


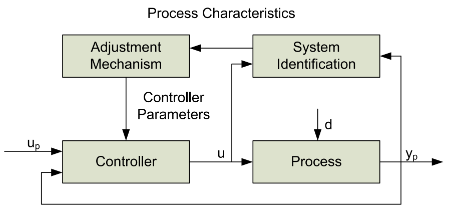
Fig. 2. Feedback control loop

the process. The process output y_p is fed back by means of sensors—and often through additional filters (not shown in Figure 2)—as y_b to compute the difference with the reference inputs u_p. The controller implements a particular control algorithm or strategy, which takes into account the difference between u_p and y_b to decide upon a suitable correction u to drive y_p closer to u_p using process-specific actuators. Often the sensors and actuators are omitted in diagrams of the control loops for the sake of brevity.

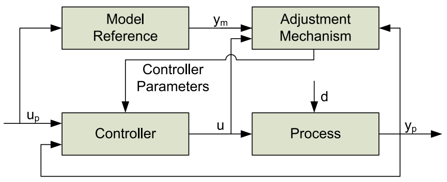
The key reason for using feedback is to reduce the effects of uncertainty which appear in different forms as disturbances or noise in variables or imperfections in the models of the environment used to design the controller [27]. For example, feedback systems are used to manage QoS in web server farms. Internet load, which is difficult to model due to its unpredictability, is one of the key variables in such a system fraught with uncertainty.

It seems prudent for the SEAMS community to investigate how different application areas realize this generic feedback loop, point out commonalities, and evaluate the applicability of theories and concepts in order to compare and leverage self-adaptive software-intensive systems research. To facilitate this comparison, we now introduce the organization of two classic feedback control systems, which are long established in control engineering and also have wide applicability.

Adaptive control in control theory involves modifying the model or the control law of the controller to be able to cope with slowly occurring changes of the controlled process. Therefore, a second control loop is installed on top of the main controller. This second control loop adjusts the controller's model and operates much slower than the underlying feedback control loop. For example, the main feedback loop, which controls a web server farm, reacts rapidly to bursts of Internet load to manage QoS. A second slow-reacting feedback loop



(a) Model Identification Adaptive Control (MIAC)



(b) Model Reference Adaptive Control (MRAC)

Fig. 3. Two standard schemes for adaptive feedback control loops

may adjust the control law in the controller to accommodate or take advantage of anomalies emerging over time.

Model Identification Adaptive Control (MIAC) [28] and *Model Reference Adaptive Control* (MRAC) [27], depicted in Figures 3(a) and 3(b), are two important manifestations of adaptive control. Both approaches use a reference model to decide whether the current controller model needs adjustment. The MIAC strategy builds a dynamical reference model by simply observing the process without taking reference inputs into account. The MRAC strategy relies on a predefined reference model (e.g., equations or simulation model) which includes reference inputs.

This MIAC system identification element takes the control input u and the process output y_p to infer the model of the current running process (e.g., its unobservable state). Then, the element provides the system characteristics it has identified to the adjustment mechanism which then adjusts the controller accordingly by setting the controller parameters. This adaptation scheme has to take also into account that a disturbances d might affect the process behavior and, thus, usually has to observe the process for multiple control cycles before initiating an adjustment of the controller.

The MRAC solution, originally proposed for the flight-control problem [27,29], is suitable for situations in which the controlled process has to follow an elaborate prescribed behavior described by the model reference. The adaptive algorithm compares the outputs of the process y_p which results from the control value u of the Controller to the desired responses from a reference model y_m for the goal u_p, and then adjusts the controller model by setting controller parameters to improve the fit in the future. The goal of the scheme is to find controller parameters that cause the combined response of the controller and process to match the response of the reference model despite present disturbances d.

The MIAC control scheme observes only the process to identify its specific characteristics using its input u and output y_p. This information is used to adjust the controller model accordingly. The MRAC control scheme in contrast provides the desired behavior of the controller and process together using a model reference and the input u_p. The adjustment mechanism compares this to y_p. The MRAC scheme is appropriate for achieving robust control if a solid and trustworthy reference model is available and the controller model does not change significantly over time. The MIAC scheme is appropriate when there is no established reference model but enough knowledge about the process to identify the relevant characteristics. The MIAC approach can potentially accommodate more substantial variations in the controller model.

Feedback loops of this sort are used in many engineered devices to bring about desired behavior despite undesired disturbances [24,27,28,29]. Hellerstein et al. provide a more detailed treatment of the analysis capabilities offered by control theory and their application to computing systems [2,13]. As pointed out by Kokar et al. [30], rather different forms of control loops may be employed for self-adaptive software and we may even go beyond classical or even adaptive control and use reconfigurable control for the software where besides the parameters also structural changes are considered (cf. compositional adaptation [31]).

2.3 Feedback Loops in Natural Systems

In contrast to self-adaptive systems built using control engineering concepts, self-adaptive systems in nature do not often have a single clearly visible control loop. Often, there is no clear separation between the controller, the process, and the other elements present in advanced control schemes. Further, the systems are often highly decentralized in such a way that the entities have no sense of the global goal but rather it is the interaction of their local behavior that yields the global goal as an emergent property.

Nature provides plenty of examples of cooperative self-adaptive and self-organizing systems: social insect behaviors (e.g., ants, termites, bees, wasps, or spiders), schools of fish, flocks of birds, immune systems, and social human behavior. Many cooperative self-adaptive systems in nature are far more complex than the systems we design and build today. The human body alone is orders of magnitude more complex than our most intricate designed systems. Further, biological systems are decentralized in such a way that allows them to benefit from built-in error correction, fault tolerance, and scalability. When encountering malicious intruders, biological systems typically continue to execute, often reducing performance as some resources are rerouted towards handling those intruders (e.g., when the flu virus infects a human, the immune system uses energy to attack the virus while the human continues to function). Despite added complexity, human beings are more resilient to failures of individual components and injections of malicious bacteria and viruses than engineered software systems are to component failure and computer virus infection. Other biological systems, for example worms and sea stars, are capable of recovering from such serious hardware failures as being cut in half (both worms and sea stars regenerate the missing pieces to form two nearly identical organisms), yet we envision neither a functioning laptop computer, half of which was crushed by a car, nor a machine that can recover from being installed with only half of an operating system. It follows that if we can extract certain properties of biological systems and inject them into our software design process, we may be able to build complex and dependable self-adaptive software systems. Thus, identifying and understanding the feedback loops within natural systems is critical to being able to design nature-mimicking self-adaptive software systems.

Two types of feedback in nature are positive and negative feedback. Positive feedback reinforces a perturbation in systems in nature and leads to an amplification of that perturbation. For example, ants lay down a pheromone that attracts other ants. When an ant travels down a path and finds food, the pheromone attracts other ants to the path. The more ants use the path, the more positive feedback the path receives, encouraging more and more ants to follow the path to the food. Negative feedback triggers a response that counteracts a perturbation. For example, when the human body experiences a high concentration of blood sugar, it releases insulin, resulting in glucose absorption, and bringing the blood sugar back to the normal concentration.

Negative and positive feedback combine to ensure system stability: positive feedback alone would push the system beyond its limits and ultimately out of control, whereas negative feedback alone prevents the system from searching for optimal behavior.

Decentralized self-organizing systems are generally composed of a large number of simple components that interact locally — either directly or indirectly. An individual component's behavior follows internal rules based only on local information. These rules can support positive and negative feedback at the level of individual components. The numerous interactions among the components then lead to global control loops.

2.4 Feedback Loops in Software Engineering

For software engineering we have observed that feedback loops are often hidden, abstracted, dispersed, or internalized when the architecture of an adaptive system is documented or presented [26]. Certainly, common software design notations (e.g., UML) do not routinely provide views that lend themselves to describing and analyzing control and reason about uncertainty. Further, we suspect that the lack of a notation leads to the absence of an explicit task to document the control, which leads in turn in failure to explicitly designing, analyzing, and validating the feedback loops.

However, the feedback behavior of a self-adaptive system, which is realized with its control loops, is a crucial feature and, hence, should be elevated to a first-class entity in its modeling, design, implementation, validation, and operation. When engineering a self-adaptive system, the properties of the control loops affect the system's design, architecture, and capabilities. Therefore, besides making the control loops explicit, the control loops' properties have to be made explicit as well. Garlan et al. also advocate to make self-adaptation external, as opposed to internal or hard-wired, to separate the concerns of system functionality from the concerns of self-adaptation [9,16].

Explicit feedback loops are common in software process improvement models [19] and industrial IT service management [32], where the system management activities and products are decoupled by the software development cycle. A major breakthrough in making feedback loops explicit came with IBM‘s autonomic computing initiative [33] with its emphasis on engineering self-managing systems. One of the key findings of this research initiative is the blueprint for building autonomic systems using MAPE-K (monitor-analyze-plan-execute over a knowledge base) feedback loops [34] as depicted in Figure 4. The phases of the MAPE-K loop or autonomic element map readily to the generic autonomic control loop as depicted in Figure 1. Both diagrams highlight the main activities of the feedback loop while abstracting away characteristics of the control and data flow around the loop. However, the blueprint provides extensive instructions on how to architect and implement the four phases, the knowledge bases, sensors, and actuators. It also outlines how to compose autonomic elements to orchestrate self-management.

Software engineering for self-adaptive systems has recently received considerable attention with a proliferation of journals, conferences, workshops (e.g., TASS, SASO, ICAC, or SEAMS). Many of the papers published in these venues dealing with the development, analysis and validation methods for self-adaptive systems do not yet provide sufficient explicit focus on the feedback loops, and their associated properties, that almost inevitably control the self-adaptations.

The idea of increasing the visibility of control loops in software architectures and software methods is not new. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [35]. She introduced a new software organization paradigm based on control loops with an architecture that is dominated by feedback loops and their analysis rather than by the identification of discrete stateful objects. Hellerstein et al. in their ground-breaking book provide a first practical treatment of the design and application of feedback control of computing systems [13]. Recently, Shaw, together with Müller and Pezzè, advocated the usefulness of a design paradigm based on explicit control loops for the design of ULS systems [26]. The preliminary ideas presented in this position paper contributed to ignite the discussion that led to the contribution of this paper.

To manage uncertainty in computing systems and their environments, we need to introduce feedback loops to control the uncertainty. To reason about uncertainty effectively, we need to elevate feedback loops to be visible and first class. If we do not make the feedback loops visible, we also will not be able to identify which feedback loops may have major impact on the overall system behavior and apply techniques to predict their possible severe effects. More seriously, we will neglect the proof obligations associated with the feedback, such as validating that y_b (i.e., the estimate of y_p derived from the sensors) is sufficiently good, that the control strategy is appropriate to the problem, that all necessary corrections can be achieved with the available actuators, that corrections will preserve global properties such as stability, and that time constraints will be satisfied. Therefore, if feedback loops are not visible we will not only fail to understand these systems but also fail to build them in such a manner that crucial properties for the adaptation behavior can be guaranteed.

ULS systems may include many self-adaptive mechanisms developed independently by different working teams to solve several classes of problems at different abstraction levels. The complexity of both the systems and the development processes may result in the impossibility of coordinating the many self-adaptive mechanisms by design, and may result in unexpected interactions with negative effects on the overall system behavior. Making feedback loops visible is an essential step toward the design of distributed coordination mechanisms that can prevent undesirable system characteristics—such as various forms of instability and divergence—due to interactions of competing self-adaptive systems.

3 Solutions Inspired by Explicit Control

The *autonomic element*—introduced by Kephart and Chess [33] and popularized with IBM's architectural blueprint for autonomic computing [34]—is the first architecture for self-adaptive systems that explicitly exposes the feedback control loop depicted in Figure 2 and the steps indicated in Figure 1, identifying

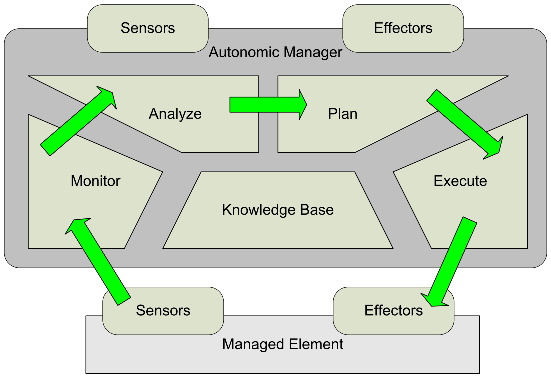


Fig. 4. IBM's autonomic element [34]

functional components and interfaces for decomposing and managing the feedback loop. To realize an autonomic system, designers compose arrangements of collaborating autonomic elements working towards common goals. In particular, IBM uses the autonomic element as a fundamental building block for realizing self-configuring, self-healing, self-protecting and self-optimizing systems [33,34].

An autonomic element, as depicted in Figure 4, consists of a managed element and an autonomic manager with a feedback control loop at its core. Thus, the autonomic manager and the managed element correspond to the controller and the process, respectively, in the generic feedback loop. The manager or controller is composed of two manageability interfaces, the sensor and the effector, and the monitor-analyze-plan-execute (MAPE-K) engine consisting of a monitor, an analyzer, a planner, and an executor which share a common knowledge base. The monitor senses the managed process and its context, filters the accumulated sensor data, and stores relevant events in the knowledge base for future reference. The analyzer compares event data against patterns in the knowledge base to diagnose symptoms and stores the symptoms for future reference in the knowledge base. The planner interprets the symptoms and devises a plan to execute the change in the managed process through its effectors. The manageability interfaces, each of which consists of a set of sensors and effectors, are standardized across managed elements and autonomic building blocks, to facilitate collaboration and data and control integration among autonomic elements. The autonomic manager gathers measurements from the managed element as well as information from the current and past states from various knowledge sources and then adjusts the managed element if necessary through its manageability interface according to its control objective.

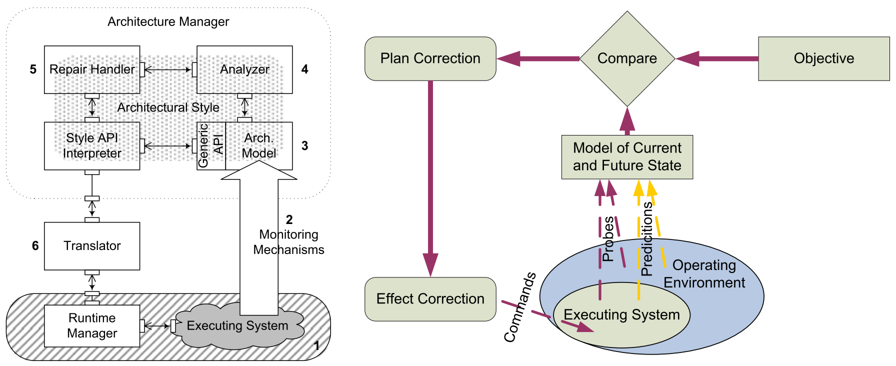
An autonomic element itself can be a managed element [34,36]. In this case additional sensors and effectors at the top of the autonomic manager are used to manage the element (i.e., provide measurements through its sensors and receive control input—rules or policies—through its effectors). If there are no such effectors, then the rules or policies are hard-wired into the control loop. Even if there are no effectors at the top of the element, the state of the autonomic element is typically still exposed through its top sensors. Thus, an autonomic element constitutes a self-adaptive system because it alters the behavior of an underlying subsystem—the managed element—to achieve the overall objectives of the system.

While the autonomic element, as depicted in Figure 4, was originally proposed as a solution for architecting self-managing systems for autonomic computing [33], conceptually, it is in fact a feedback control loop from classic control theory.

Garlan et al. have developed a technique for using feedback for self-repair of systems [9]. Figure 5(a) shows their system. They add an external controller (top box) to the underlying system (bottom box), which is augmented with suitable actuators. Their architecture maps quite naturally to the generic feedback control loop (cf. Figure 2).

To see this, Figure 5(b) introduces two elaborations to the generic control loop. First, we separate the controller into three parts (compare, plan correction, and effect correction). Second, we elaborate the value of y_b, showing that sensors can sense both the executing system and its operating environment and by explicitly adding a component to convert observations to modeled value. In redrawing the diagram, we have arranged the components so that they overlay the corresponding components of the Rainbow architecture diagram. To show that the feedback loop is clearly visible in the Rainbow architecture, we provide the mapping between both architectures in Table 1.

An example for a self-adaptive system following the MIAC scheme applied to software is the robust feedback loop used in self-optimization that is becoming prevalent in performance-tuning and resource-provisioning scenarios (cf. Figure 6(a)) [37,38]. Robust feedback control tolerates incomplete knowledge about the system model and assumes that the system model has to be frequently rebuilt. To accomplish this, the feedback control includes an Estimator



(a) Rainbow system's architecture [9]

(b) Shaw's elaborated feedback control architecture [26]

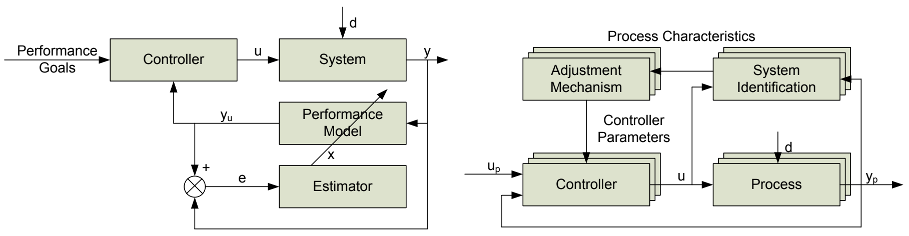
Fig. 5. The Rainbow system and Shaw's feedback control loop

Table 1. Mapping showing the correspondence between the elements of the Rainbow system's architecture and Shaw's architecture

Rainbow system (cf. Figure 5(a))	Model (cf. Figure 5(b))
(1) Executing sys. with runtime manager	Executing sys. in its operating environment
(2) Monitoring mechanisms	Probes
(–) (Rainbow is not predictive)	Predictions
(3) Architectural model	Objective, model of current state
(4) Analyzer	Compare
(5) Repair handler	Plan correction
(6) Translator, runtime manager	Effect correction, commands

that estimates state variables (x) that cannot be directly observed. The variables x are then used to tune a performance model (i.e., queuing network model) online, allowing that model to provide a quantitative dependency law between the performance outputs and inputs (y_u) of the system around an operational point. This dependence is dynamic and captures the influence of perturbations w, as well as time variations of different parameters in the system (e.g., due to software aging, caching, or optimizations). A Controller uses the y_u dependency to decide when and what resources to tune or provision. The Controller uses online optimization algorithms to decide what adaptation to perform. Since performance is affected by many parameters, the Controller chooses to change only those parameters that achieve the performance goals with minimum resource consumption. Since the system changes in time, so does the performance dependence between the outputs and inputs. However, the scheme still works because the Estimator and the Performance Model provide the Controller with an accurate reflection of the system.

Another example of an MIAC scheme is a mechatronics system consisting of a system of autonomous shuttles that operate on demand and in a decentralized manner using a wireless network [39]. Realizing such a mechatronics system makes it necessary to draw from techniques offered by the domains of control engineering as well as software engineering. Each shuttle that travels along a specific track section approaches the responsible local section control to obtain data about the track characteristics. The shuttle optimizes the control behavior for passing that track section based on that data and the specific characteristics of the shuttle. The new experiences are then propagated back to the section control



(a) Self-optimization via feedback loop (b) Multiple intertwined MIAC schemes

Fig. 6. Applications of adaptive control schemes for self-adaptive systems

such that other shuttles may benefit from them (i.e., improve their model). The shuttles implement the MIAC control loop as a group as depicted in Figure 6(b). Each shuttle traveling along a track only realizes the Adjustment Mechanism and Controller while the shuttles, which have reported on the track characteristics before, and the section control collectively realize the System Identification. Note that this constitutes a form of cooperative self-adaptation where multiple elements are involved in a single adaptive control loop.

4 Solutions Inspired by Natural Systems

Because nature-inspired engineering is a younger area of research than control theory, emerging nature-inspired solutions for self-adaptive software have not yet been classified and contrasted against one another. In this section, we present some of the existing nature-inspired solutions for self-adaptive software systems. While some current work deals with building biologically inspired self-adapting software systems [40,41], even more work has gone into studying biological systems to inspire the design of software and hardware systems in robotics [1,42,43,44]. It remains a challenge to employ biological knowledge to develop an understanding of how to build software systems that function the way biological systems do, and to design appropriate architectures, design tools, and programming tools to create such systems.

In nature, the process of crystal growth can result in well-formed regular crystals or high-error irregular crystals. The key aspect that determines which type of crystal will form is the speed at which the crystal grows. If the crystal grows slowly, then badly and weakly attached molecules detach from the crystal and the final result has very few, if any, errors. However, if the crystal grows quickly, badly attached molecules are locked in by other attachments before they can detach, and the final result has many errors. The tile architectural style [40], a software architectural style inspired by crystal growth, leverages the feedback exhibited by crystal growth to allow fault and adversary tolerance [4]. The tile style allows distributing computation of NP-complete problems on a large network in a secure, dependable, and scalable manner [40]. The control loops within tile-style systems are difficult to classify as positive or negative; however, they do fit nicely into the feedback loop described in Figure 2. The individual components attach to one another, *collecting* information on what other components may attach. After a faulty or malicious agent attaches an illegal component, future attachment cannot happen, and *analysis* reveals that the assembly became locally "stuck." (Since the system is only affected locally, the computational resources are rerouted and the system as a whole makes progress, incurring only negligible reduction in computation speed.) The surrounding components *decide* to detach a few most-recently attached neighbors, and resume attaching new components. As this *action* selects prospective components at random, faulty or malicious agents are unlikely to be able to penetrate the assembly two or more times, thus resulting in a fault- and adversary-tolerant software system.

Schools of fish and flocks of birds adapt their behavior by using direct communication. They follow a set of attraction and repulsion rules: maintaining a minimum distance from other objects in the environment, matching own velocity with that of neighbors, and moving toward the perceived center of mass in one's neighborhood. These rules provoke a wave of reactions that are communicated progressively to all components of the school or flock. Certain software systems use similar mechanisms—for example, process schedulers and network routing protocols.

In contrast, ants and wasps use stigmergy as an indirect communication mechanism by leaving clues in the environment for the others. Ants add pheromone to their environments to denote paths to food and wasp-nest construction follows a work-in-progress mechanism (each cell added to the nest creates a new nest configuration and each configuration triggers a particular response in the wasps). Research in swarm robotics has used stigmergy extensively to solve static and dynamic optimization problems. More generally, stigmergy as an indirect communication medium is being used for coordinating unmanned vehicles [45].

Mammalian immune systems provide a defense mechanism by detecting antigens (intruders) and by coordinating a collective decentralized response to destroy them. These distributed, decentralized systems balance detection and removal of malicious agents against interference with normal cell processes and employ learning techniques. Software intrusion detection research already leverages some immune-system ideas [46], but understanding these intricate self-organizing defenses can offer much more insight into engineering self-adaptive systems.

More generally, current practice in engineering self-organizing systems encompass the use of autonomous components (agents), establishment of behavior interactions rules following adaptive mechanisms inspired by nature or use of middleware with built-in features supporting adaptive mechanisms (such as digital pheromone propagation).

5 Challenges Ahead

We have argued that the feedback loop should be a first-class entity when thinking about engineering of self-adaptive systems. We believe that understanding and reasoning about the control loop is key for advancing the construction of self-adaptive systems from an ad-hoc, trial-and-error endeavor towards a more disciplined approach. To achieve this goal, the following issues, possibly among others, have to be addressed.

Modeling: There should be modeling support to make the control loop explicit and to expose self-adaptive properties so that the designer can reason about the system. The models have to capture what can be observed and what can be influenced. It would be desirable to have a widely agreed upon standard (e.g., in the form of reference models with domain-specific notations) for self-adaptive systems including the control loop. Highly decentralized self-organizing systems, such as swarms, need to have proper models of control loops, even though today, that control loop is only implicitly present in the models.

The nature of a self-adaptive system requires to reify properties that would otherwise be encoded implicitly. These reified properties need to be modeled appropriately so that they can be queried and modified during runtime. Examples of such properties are system state that is used to reason about the system's behavior, and policies and business goals that govern and constrain how the system can and will adapt.

Control Loops: We have described a number of types of control loops found in control-engineering, natural, robotics, and software systems. Our list is by no means comprehensive and other types of control loops, and self-adaption techniques that leverage control loops and interactions between control loops, exist. One challenge to advancing the engineering of self-adaptive software systems is creating a reference library of control-loop types and mechanisms of control-loop interactions. To create this library, we must mine, understand, and leverage existing systems and then classify and catalog their self-adaptation mechanisms. In particular, natural systems are rich sources of distinct and novel control loops and control-loop interactions.

Architecture and Design: Decisions concerning feedback loops in the architecture and design of self-adaptive systems can leverage past experience. Control theory research found that systems with a single control loop are easier to reason about than systems with multiple loops, although the latter are far more common. Since good engineering practice calls for simple design, engineers should consider minimizing the number of control loops or decoupling control loops from each other. Such a decoupling can happen with respect to time, ensuring that the loops operate at different time scales, or with respect to space, weakening the dependencies between variables. When complete decoupling is not possible, the design must make the control-loop interactions, and their handling, explicit. Thus, designs containing multiple control loops must be carefully considered and analyzed, as in, for example, the MIAC or MRAC designs (cf. Figures 3(b) and 3(b)).

Control engineering research has also identified that hierarchical organization of control loops reduces the design complexity. In this scheme, the loops influence each other top-down and operate at different time scales, avoiding unexpected interference between the hierarchy levels. Hierarchical organization is of particular interest if it is possible to distinguish different time scales and different controlled variables [37] or different adaptation domains within a software system, such as change management and goal management [17].

Reference architectures for adaptive systems should therefore highlight key aspects of feedback loops, including their number, structural arrangements (e.g., sequential, parallel, hierarchical, decentralized), interactions, data flow, tolerances, trade-offs, sampling rates, stability and convergence conditions, hysteresis specifications, and context uncertainty [36]. It is highly desirable that such architectures can be used to reason about the properties of the system and its control loop. In other words, we must determine whether it is possible to build Attribute-Based Architectural Styles for control loops in self-adaptive systems [47].

Unintended-Interaction Detection: For some systems (e.g., ULS systems), their complexity may limit the possibility of hierarchical organization of control loops or other methods of control-loop decoupling. Control loops developed independently to cope with different problems at various abstraction levels may result in unexpected interactions with negative effects on the system behavior.

Combining distinct subsystems with seemingly independent goals can often result in emergent behaviors, some of which can be desirable, while others are undesirable. Nature suggests two avenues of research toward potential solutions to detecting and avoiding unintended interactions between control loops: (1) mining the abundant systems from nature with control loops that do not interact in undesirable ways to understand how to develop such control loops in engineered systems; and (2) understanding the process that has arrived at systems with only desirably interacting control loops (e.g., evolution) and applying a similar process to select or decouple control loops automatically in engineered systems.

Maintenance: Since maintenance constitutes a significant portion of a software system's life cycle, understanding maintainability concerns specific to self-adaptive systems poses an important challenge. Examples of issues that should be tackled are how the maintainability concerns of self-adaptive systems and traditional systems compare and whether a system designed for dynamic variability or adaptation is easier to maintain than a static system [36,48]. It is reasonable to expect differences in maintenance of the two kinds of systems because some self-adaptive systems add a reflective layer that enables runtime analysis and adaptation. Consequently, a maintenance activity may involve changes to either, or both, the system's *meta level* or *base level* [49].

Middleware Support: Currently, the building of self-adaptive systems is tedious because of the lack of a reusable code base. A dedicated development and execution environment (e.g., in the form of a framework or library) for building systems with self-adaptive features would go a long way in resolving this challenge. As a vision, good middleware support should "allow researchers with different motivations and experiences to put their ideas into practice, free from the painful details of low-level system implementation" [50]. Such an infrastructure should define standardized interfaces and services, support different heterogeneous platforms, allow for the rapid prototyping of self-adaptive features, and involve hybrid architectures (e.g., combining top-down and bottom-up or centralized and decentralized approaches).

Verification and Validation: Development of self-adaptive systems requires techniques to validate the effects of feedback loops. Classical control engineering provides sophisticated solutions for the analysis of continuous-feedback control loops [2,13]. However, some phenomena relevant to software and self-adaptation have a discrete or hybrid nature (e.g., architectural changes). In addition to control engineering, discrete event systems [51], switched systems [52], and hybrid systems [53] may provide mechanisms relevant to self-adaptive systems.

Reengineering: Today, most engineering issues for self-adaptive systems are approached from the perspective of greenfield development. However, many legacy applications can benefit from self-adaptive features. Reengineering of existing systems with the goal of making them more self-adaptive in a cost-effective and principled manner poses an important challenge. Of particular concern is the question of how to inject a control loop into an existing system. Technologies and tools that allow an engineer to (semi-)automatically augment an existing system with sensors and effectors can begin to answer this challenge. Further, existing systems should be gradually migrated towards self-adaptive capabilities (a.k.a. the chicken little approach), for example, by increasing the scope of self-adaptive control from subcomponents towards the entire business infrastructure or by increasing self-adaptive functionality in a single component by substituting high-level-goal-based self-adaptive behavior for manual configuration.

Human-Computer Interaction: Even though self-adaptive systems act autonomously in many respects, they have to keep the user in the loop. Providing the user with feedback about the system state is crucial to establish and keep users' trust. To that effect, a self-adaptive system needs to expose aspects of its control loop to the user. For example, if a web server is reconfigured in response to a load change, the human administrator needs (visual) feedback that the performed adaptation has a positive effect.

Also, users should be given the option to disable self-adaptive features and the system should take care not to contradict explicit choices made by users [54]. Furthermore, users might want feedback from the system about the information collected by sensors and how this information is used to adapt the system. In fact, if the collected information is personal data there might be even a legal obligation to do so [55].

6 Conclusions

We have outlined in this paper that feedback loops are a key factor in software engineering of self-adaptive systems. In the case of top-down self-adaptive architectures employing explicitly-engineered feedback control loops, these loops are of paramount importance to guide engineering of the self-adaptive part of those systems. In case of systems inspired by biological and natural systems, identifying the feedback loops and understanding their impact is essential. While notions of the feedback loop that can be found in the areas of control theory and natural systems can provide valuable insight, software engineering needs to develop its own unique notion of feedback loop that is suitably aligned with its own problem domain. Therefore, we argue for the necessity of well-founded approaches for the models, architectures, design, implementation, maintenance, and verification techniques of self-adaptation, while taking into account the notion of reengineering existing systems to contain self-adaptation. We think that aligning our efforts with the key concept of feedback loops, which has been somewhat ignored in software engineering, will bring our community closer to the goal of building complex self-adapting systems. Satisfying the challenges we outlined above is the first step toward this endeavor.

Acknowledgments

This paper is the result of stimulating discussions among the authors and other participants during the seminar on Software Engineering for Self-Adaptive Systems at Schloss Dagstuhl in January 2008. Some of the ideas developed in this paper have initially been presented elsewhere [26,36,56].

References

1. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Thomas, F., Knight, J., Nagpal, R., Rauch, E., Sussman, G.J., Weiss, R.: Amorphous computing. Communications of the ACM 43(5), 74–82 (2000)
2. Diao, Y., Hellerstein, J.L., Parekh, S., Griffith, R., Kaiser, G., Phung, D.: Control theory foundation for self-managing computing systems. IEEE Journal on Selected Areas in Communications 23(12), 2213–2222 (2005)
3. Di Marzo-Serugendo, G., Gleizes, M.P., Karageorgos, A.: Self-organisation in MAS. Knowledge Engineering Review 20(2), 165–189 (2005)
4. Brun, Y., Medvidovic, N.: Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In: 2nd ACM International Workshop on Engineering Fault Tolerant Systems (EFTS 2007), Dubrovnik, Croatia, pp. 38–43 (2007)
5. Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute (2006), `http://www.sei.cmu.edu/uls/`
6. Ottino, J.M.: Engineering complex systems. Nature 427(6973), 399–400 (2004)
7. Brown, G., Cheng, B.H., Goldsby, H., Zhang, J.: Goal-oriented specification of adaptation requirements engineering in adaptive systems. In: ACM 2006 International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS 2006), Shanghai, China, pp. 23–29 (2006)
8. Richter, U., Mnif, M., Branke, J., Müller-Schloer, C., Schmeck, H.: Towards a generic observer/controller architecture for organic computing. In: Hochberger, C., Liskowsky, R. (eds.) INFORMATIK 2006: Informatik für Menschen. GI-Edition – Lecture Notes in Informatics, vol. P-93, pp. 112–119. Gesellschaft für Informatik (2006)
9. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing system dependability through architecture-based self-repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677. Springer, Heidelberg (2003)
10. Liu, H., Parashar, M.: Accord: a programming framework for autonomic applications. IEEE Transactions on Systems, Man, and Cybernetics 36(3), 341–352 (2006)
11. Peper, C., Schneider, D.: Component engineering for adaptive ad-hoc systems. In: ACM 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), Leipzig, Germany, pp. 49–56 (2008)
12. Tanner, J.A.: Feedback control in living prototypes: A new vista in control engineering. Medical and Biological Engineering and Computing 1(3), 333–351 (1963), `http://www.springerlink.com/content/rh7wx0675k5mx544/`

13. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. John Wiley & Sons, Chichester (2004)
14. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: 4th ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE 1996), San Francisco, CA, USA, pp. 3–14. ACM Press, New York (1996)
15. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14(3), 54–62 (1999)
16. Cheng, S.W., Garlan, D., Schmerl, B.: Making self-adaptation an engineering reality. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 158–173. Springer, Heidelberg (2005)
17. Kramer, J., Magee, J.: Self-managed systems: An architectural challenge. In: Future of Software Engineering (FOSE 2007), Minneapolis, MN, USA, pp. 259–268. IEEE Computer Society, Los Alamitos (2007)
18. Royce, W.W.: Managing the development of large software systems. In: 9th ACM/IEEE International Conference on Software Engineering (ICSE 1970), pp. 328–338 (1970)
19. Boehm, B.W.: A spiral model of software development and enhancement. IEEE Computer 21(5), 61–72 (1988)
20. Lehman, M.M.: Software's future: Managing evolution. IEEE Software 15(1), 40–44 (1998)
21. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions Autonomous Adaptive Systems (TAAS) 1(2), 223–259 (2006)
22. Nilsson, N.J.: Principles of Artificial Intelligence. Tioga Press, Palo Alto (1980)
23. Gat, E.: Three-layer Architectures, pp. 195–210. MIT/AAAI Press, Cambridge (1997)
24. Burns, R.: Advanced Control Engineering. Butterworth-Heinemann (2001)
25. Dorf, R.C., Bishop, R.H.: Modern Control Systems, 10th edn. Prentice-Hall, Englewood Cliffs (2005)
26. Müller, H.A., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), Workshop at 30th IEEE/ACM International Conference on Software Engineering (ICSE 2008), Leipzig, Germany (May 2008)
27. Astrom, K., Wittenmark, B.: Adaptive Control, 2nd edn. Addison-Wesley, Reading (1995)
28. Söderström, T., Stoica, P.: System Identification. Prentice-Hall, Englewood Cliffs (1988)
29. Dumont, G., Huzmezan, M.: Concepts, methods and techniques in adaptive control. In: 2002 IEEE American Control Conference (ACC 2002), Anchorage, AK, USA, vol. 2, pp. 1137–1150 (2002)
30. Kokar, M.M., Baclawski, K., Eracar, Y.A.: Control theory-based foundations of self-controlling software. IEEE Intelligent Systems 14(3), 37–45 (1999)
31. McKinley, P.K., Sadjadi, M., Kasten, E.P., Cheng, B.H.: Composing adaptive software. IEEE Computer 37(7), 56–64 (2004)
32. Brittenham, P., Cutlip, R.R., Draper, C., Miller, B.A., Choudhary, S., Perazolo, M.: IT service management architecture and autonomic computing. IBM Systems Journal 46(3), 565–581 (2007)
33. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)

34. IBM Corporation: An architectural blueprint for autonomic computing. White Paper, 4th edn., IBM Corporation, `http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`
35. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
36. Müller, H.A., Kienle, H.M., Stege, U.: Autonomic computing: Now you see it, now you don't. In: Lucia, A.D., Ferrucci, F. (eds.) Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures. LNCS, vol. 5413, pp. 32–54. Springer, Heidelberg (2009)
37. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical model-based autonomic control of software systems. In: ACM ICSE Workshop on Design and Evolution of Autonomic Software, St. Louis, MO, USA, pp. 1–7 (2005)
38. Litoiu, M., Mihaescu, M., Ionescu, D., Solomon, B.: Scalable adaptive web services. In: Development for Service Oriented Architectures (SD-SOA 2008), Workshop at 30th IEEE/ACM International Conference on Software Engineering (ICSE 2008), Leipzig, Germany (2008)
39. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool support for the design of self-optimizing mechatronic multi-agent systems. International Journal on Software Tools for Technology Transfer (STTT) 10(3) (2008)
40. Brun, Y., Medvidovic, N.: An architectural style for solving computationally intensive problems on large networks. In: Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007), Workshop at 29th IEEE/ACM International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA (2007)
41. Di Marzo-Serugendo, G., Fitzgerald, J., Romanovsky, A., Guelfi, N.: A generic framework for the engineering of self-adaptive and self-organising systems. Technical report, School of Computer Science, University of Newcastle, Newcastle, UK (2007)
42. Nagpal, R.: Programmable Self-Assembly: Constructing Global Shape Using Biologically-Inspired Local Interactions and Origami Mathematics. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2001)
43. Clement, L., Nagpal, R.: Self-assembly and self-repairing topologies. In: Workshop on Adaptability in Multi-Agent Systems, First RoboCup Australian Open (AORC 2003), Sydney, Australia (2003)
44. Shen, W.M., Krivokon, M., Chiu, H., Everist, J., Rubenstein, M., Venkatesh, J.: Multimode locomotion via superbot reconfigurable robots. Autonomous Robots 20(2), 165–177 (2006)
45. Sauter, J.A., Matthews, R., Parunak, H.V.D., Brueckner, S.A.: Performance of digital pheromones for swarming vehicle control. In: 4th International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS 2005), The Netherlands, pp. 903–910. ACM, New York (2005)
46. Hofmeyr, S., Forrest, S.: Immunity by design: An artificial immune system. In: Genetic and Evolutionary Computation Conference (GECCO 1999), Orlando, Florida, USA, pp. 1289–1296. Morgan-Kaufmann, San Francisco (1999)
47. Klein, M., Kazman, R.: Attribute-based architectural styles. Technical Report CMU/SEI-99-TR-022, Software Engineering Institute (SEI) (1999), `http://www.sei.cmu.edu/pub/documents/99.reports/pdf/99tr022.pdf`
48. Zhu, Q., Lin, L., Kienle, H.M., Müller, H.A.: Characterizing maintainability concerns in autonomic element design. In: 24th IEEE International Conference on Software Maintenance (ICSM 2008), Beijing, China, pp. 197–206 (2008)

49. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: 2009 International Workshop on Self-Adaptation and Self-Managing Systems (SEAMS 2009), Vancouver, BC, Canada (to be published, 2009)
50. Babaoglu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A.P.A.: The self-star vision. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 1–20. Springer, Heidelberg (2005)
51. Passino, K.M., Burgess, K.L.: Stability analysis of discrete event systems. Adaptive and Learning Systems for Signal Processing Communications, and Control. John Wiley & Sons, Inc., New York (1998)
52. Liberzon, D., Morse, A.: Basic problems in stability and design of switched systems. IEEE Control Systems Magazine 19(5), 59–70 (1999)
53. Decarlo, R.A., Branicky, M.S., Pettersson, S., Lennartson, B.: Perspectives and Results on the Stability and Stabilizability of Hybrid Systems. Proceedings of the IEEE 88(7), 1069–1082 (2000)
54. Lightstone, S.: Seven software engineering principles for autonomic computing development. Innovations in Systems and Software Engineering 3(1), 71–74 (2007)
55. Sackmann, S., Strüker, J., Accorsi, R.: Personalization in privacy-aware highly dynamic systems. Communications of the ACM 49(9), 32–38 (2006)
56. Cheng, B.H., de Lemos, R., Giese, H., et al.: Software engineering for self-adaptive systems: A research roadmap. In: Cheng, B.H., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)