# Autonomic Computing
# Now You See It, Now You Don't

## Design and Evolution of Autonomic Software Systems

Hausi A. Müller, Holger M. Kienle, and Ulrike Stege

Department of Computer Science
University of Victoria
{hausi,kienle,stege}@cs.uvic.ca
http://webhome.cs.uvic.ca/~{hausi,stege}

**Abstract.** With the rapid growth of web services and socio-technical ecosystems, the management complexity of these modern, decentralized, distributed computing systems presents significant challenges for businesses and often exceeds the capabilities of human operators. Autonomic computing is an effective set of technologies, models, architecture patterns, standards, and processes to cope with and reign in the management complexity of dynamic computing systems using feedback control, adaptation, and self-management. At the core of an autonomic system are control loops which sense their environment, model their behavior in that environment, and take action to change the environment or their own behavior. Computer science researchers often approach the design of such highly dynamical systems from a software architecture perspective whereas engineering researchers start with a feedback control perspective. In this article, we argue that both design perspectives are needed and necessary for autonomic system design.

**Keywords:** Continuous evolution, software ecosystems, software complexity management, autonomic computing, self-managing systems, self-adaptive systems, feedback loops, autonomic element, autonomic computing reference architecture, autonomic patterns.

## 1 Introduction

Two important trends are dominating the computing world in the new millennium. First, many companies are moving from a goods-centric way to a service-centric way of conducting business [34]. In this service-oriented world, we perform everyday tasks, such as communication, banking, or shopping, without human-to-human interaction from the comfort of our living rooms. This apparently seamless integration of services and computing power has put enormous demands on its underlying information technology (IT) infrastructure. Second, with the proliferation of computing devices and enterprise systems, software systems have evolved from software intensive systems to systems of systems [56], and now to ultra-large-scale systems and socio-technical ecosystems [51]. With the rapid growth of web services and socio-technical ecosystems, the management complexity of these modern, decentralized, distributed computing systems presents significant challenges for businesses and often exceeds the capabilities of human operators who manage these systems.

The continuous evolution from goods-centric to service-centric businesses and from software intensive systems to socio-technical ecosystems requires new and innovative approaches for building, running, and managing software systems. Understanding, managing, and controlling the run-time dynamics of these computing systems—given the onslaught of ever-changing IT infrastructure—is nowadays a crucial requirement for the survival and success of many businesses. Therefore, end-users increasingly demand from business that they provide software systems that are versatile, flexible, resilient, dependable, robust, service-oriented, mashable, inter-operable, continuously available, decentralized, energy-efficient, recoverable, customizable, self-healing, configurable, or self-optimizing.

One of the most promising approaches to achieving some of these properties is to equip software systems with feedback control to address the management of inherent system dynamics. The resulting self-managing and self-adapting computing systems, which embody multiple feedback control loops at their core, are better able to cope with and even accommodate changing contexts and environments, shifting requirements, and computing-on-demand needs.

Over the past decade, many industrial and academic research initiatives have emerged to attack these problems and challenges head-on and produced impressive and profitable solutions for many businesses. One of the best-documented and most successful research initiatives is IBM's Autonomic Computing initiative, launched in 2001 with Horn's grand challenge for the entire IT sector [26]. The rest of the computing industry followed quickly with major initiatives to take on this formidable challenge. For example, the goal of Sun's N1 management software is to enable application development across heterogeneous environments in a consistent and virtualized manner [58]. The Dynamic Systems Initiative (DSI) is Microsoft's technology strategy for products and solutions to help businesses meet the demands of a rapidly changing and adaptable environment [44]. Hewlett-Packard's approach to autonomic computing is reified in its Adaptive Enterprise Strategy [25]. Intel is deeply involved in the development of standards for autonomic computing [60].

The goal of this paper is to illustrate the broad applicability of autonomic computing techniques from IT complexity problems to continuous software evolution problems, to motivate the benefits of mining the rich history of control theory foundations for autonomic, self-managing and self-adaptive systems, and to present selected challenges for the future. Section 2 characterizes the problem of continuous software evolution using different perspectives. Section 3 discusses selected approaches, issues, and challenges when designing dynamic computing systems using feedback control. Section 4 introduces the field of autonomic computing and illustrates how autonomic computing technology can be used to reduce complexity and solve continuous evolution problems of software-intensive systems. Section 5 relates traditional feedback control, such as model reference adaptive control (MRAC) or model identification adaptive control (MIAC), to feedback control for dynamic computing systems. Sections 6 and 7 describe cornerstones of IBM's autonomic computing technology, including the autonomic computing reference architecture (ACRA) and two key architecture components, autonomic element and autonomic manager. Section 8 presents some lessons learned and outlines selected research challenges for this exciting field, and Section 9 draws some conclusions and points out selected entries in the references as good starting points for newcomers to the autonomic computing field.

## 2 Continuous Evolution of Software Systems

In a world where continuous evolution, socio-technical ecosystems, and service-centric businesses are prevalent, we are forced to re-examine some of our most fundamental assumptions about software and its construction. For the past 40 years, we have embraced a traditional engineering perspective to construct software in a centralized, top-down manner where functional or extra-functional requirements are either satisfied or not [51]. Today, many software-intensive systems of systems emerge through continuous evolution and by means of regulation rather than traditional engineering. For example, firms are engineered—but the structure of the economy is not; the protocols of the Internet were engineered—but not the web as a whole [51]. While software ecosystems, such as the web, exhibit high degrees of complexity and organization, it is not built through traditional engineering. However, individual components and subsystems of such ecosystems are still being built using traditional top-down engineering [51].

Three independent studies, conducted in 2006, seem to confirm that this notion of continuous evolution is taking hold [3,5,51]. In particular, the acclaimed report of Carnegie Mellon Software Engineering Institute (SEI) on ultra-large-scale (ULS) systems suggests that traditional top-down engineering approaches are insufficient to tackle the complexity and evolution problems inherent in decentralized, continually evolving software [51].

While there has been recent attention to the problem of continuous evolution, several researchers have already articulated this problem at the end of the nineties. In his dissertation, Wong envisioned the *Reverse Engineering Notebook* to cope with and manage continuous evolution issues of software systems [62]. At the same time, Truex et al. recognized that the traditional assumption where "software systems should support organizational stability and structure, should be low maintenance, and should strive for high degrees of user acceptance" might be flawed [61]. They suggested an alternate view where "software systems should be under constant development, can never be fully specified, and are subject to constant adjustment and adaptation".

Having investigated software systems and their engineering methods for many years, we have come to realize that their alternate, continuous evolution view is as important today—or even more important for certain application domains—than the traditional view. For narrow domains, well-defined applications, or safety-critical tasks, the traditional view and engineering approach still applies.

For highly dynamic and evolving systems, such as software ecosystems, the continuous evolution view clearly applies. This is good news for the software engineering research community at large since this shift guarantees research problems for many years to come. The not so good news is that most software engineering textbooks, which only treat and advocate the traditional view, will have to be rewritten or at least updated to incorporate the notion of continuous evolution [46].

In a 2004 Economist article, Kluth discussed how other industrial sectors previously dealt with complexity [36]. He and others have argued that for a technology to be truly successful, its complexity has to disappear. He illustrates this point by showing how industries such as clocks, automobiles, and power distribution overcame complexity challenges. For example only mechanics were able to operate early automobiles successfully and in the early days of the 20th century, companies had a prominent position of Vice President of Electricity to deal with power generation and consumption issues. In both cases, the respective industries managed to reduce the need of human expertise and simplify the usage of the underlying technology with traditional engineering methods. However, usage simplicity comes with an increased complexity of the overall system complexity (e.g., what is under the hood). Basically for every mouse click or key stroke we take out of the user experience, 20 things have to happen in the software behind the scenes [36]. Given this historical perspective with this predictable path of technology evolution, there may be hope for the information technology sector [45].

Today, there are several research communities (cf. Section 1) which deal with highly dynamic and evolving systems from a variety of perspectives. For example, Inverardi and Tivoli argue that the execution environment for future software systems will not be known a priori at design time and, hence, the application environment of such a system cannot be statically anticipated [32]. They advocate reconciling the static view with the dynamic view by breaking the traditional division among development phases by moving some activities from design time to deployment and run time. What the approaches of these different communities have in common is that the resulting systems push design decisions towards run-time and exhibit capabilities to reason about the systems' own state and their environment. However, different communities emphasize different business goals and technological approaches.

The goal of autonomic computing or self-managing systems is to reduce the total cost of ownership of complex IT systems by allowing systems to self-manage by combining a technological vision with on-demand business needs [16,17]. Autonomic communication is more oriented towards distributed systems and services and the management of network resources [13]. Research on self-adaptation spans a wide range of applications from user-interface customization and web-service composition, to mechatronics and robotics systems, to biological systems, and to system management. As a result, distinct research areas and publication venues have emerged, including adaptive, self-adaptive, self-managing, autonomic, autonomous, self-organizing, reactive, and ubiquitous systems.

Feedback control is at the heart of self-managing and self-adaptive systems. Building such systems cost-effectively and in a predictable manner is a major engineering challenge even though feedback control has a long history with huge successes in many different branches of engineering [59,63]. Mining the rich experiences in these fields, borrowing theories from control engineering [1,6,23,12,50], and then applying the findings to software-intensive self-adaptive and self-managing systems is a most worthwhile and promising avenue of research. In the next section we introduce a generic model of an autonomic control loop that exposes the feedback control of self-managing and self-adaptive systems, providing a first step towards reasoning about feedback control (e.g., properties of the control loop during design, and implications of control loops during maintenance).

## 3 Design and Maintenance Issues of Feedback-Based Systems

Self-adaptive and self-managing systems have several properties in common. First and foremost such systems are "reflective" in nature and are therefore able to reason about their state and environment. Secondly these systems address problems for which selected design decisions are necessarily moved towards run-time [55,9,2]. Müller et al. proposed a set of problem attributes which suggest considering self-adaptive and self-managing solutions [47]:

- Uncertainty in the environment, especially uncertainty that leads to substantial irregularity or other disruption or may arise from external perturbations, rapid irregular change, or imprecise knowledge of the external state.
- Nondeterminism in the environment, especially of a sort that requires significantly different responses at different times.
- Requirements, especially extra-functional requirements, which can best be satisfied through regulation of complex, decentralized systems (as opposed to traditional, top-down engineering) especially if substantial trade-offs arise among these requirements.
- Incomplete control of system components (e.g., the system incorporates embedded mechanical components, the task involves continuing action, or humans are in the operating loop).

The reasoning about a system's state and environment typically involves feedback processes with four canonical activities—*collect, analyze, decide,* and *act*—as depicted in Figure 1 [13]. Sensors, or probes, "collect" data from the executing process and its context about their current states. The accumulated data is then cleaned, filtered, and finally stored for future reference to portray an accurate model of past and current states. The diagnosis engine then"analyzes" the data to infer trends and identify symptoms. The planning engine then attempts to predict the future to "decide" on how to "act" on the executing process and its context through effectors.

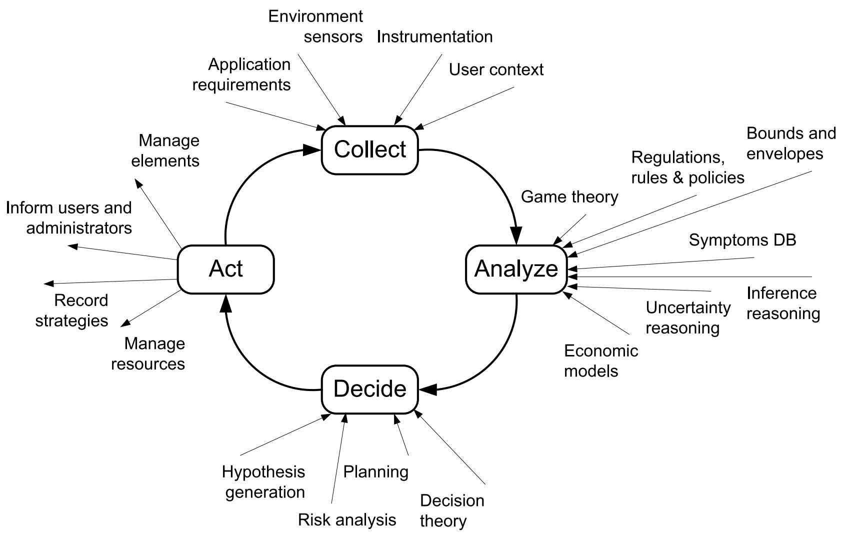

**Fig. 1.** Generic control loop [13]

This generic model provides a good overview of the main activities around the feedback loop, but ignores many properties of the control and data flow around the loop. In the following we give examples of control loop properties for each activity (cf. Figure 1) [9]:

**Collect**
What kinds of data and events are collected from which sources, sensors, or probes? Are there common event formats? What is the sampling rate and is it fixed or varying? Are the sampled sources fixed or do they change dynamically? What are appropriate filters for the data streams?

**Analyze**
How are the collected data represented and stored? What are appropriate algorithms or diagnosis methods to analyze the data? How is the current state of the system assessed? How much past state needs to be kept around? How are critical states archived? How are common symptoms recognized (e.g., with the help of a symptoms database)?

**Decide**
How is the future state of the system inferred and how is a decision reached (e.g., with off-line simulation, quality of service (QoS) objectives, or utility/goal functions)? What models and algorithms are used for trade-off analysis? What are the priorities for adaptation across multiple control loops and within a single control loop? Under what conditions should adaptation be performed (e.g., allow for head-room or avoid system thrashing while considering timing issues relating to the required adaptations)?

**Act**
What are the managed elements and how can they be manipulated (e.g., by parameter tuning or by injecting new algorithms)? Are changes of the

system pre-computed (e.g., switching between known configurations) or opportunistically assembled, composed, or generated?

Only a system designed to be self-adaptive is able to self-manage by monitoring and responding to changes in its own state or its external operating environment. As a result, designers and maintainers of such systems must take uncertainty into account because the environment may change in unexpected ways and cause the system to adapt in such a way that was not foreseeable at design time. Introducing uncertainty requires trade-offs between flexibility and assurance. For a maintainer it is critical to know which parts of the environment are assumed to be fixed and which are expected to introduce uncertainty.

Two key maintenance questions arise in the context of self-adaptive and self-managing systems [64]:

– How different are the maintainability concerns for self-adaptive and self-managing systems compared to static systems?
– Is a system that is designed for dynamic variability or adaptation easier to maintain?

It is reasonable to expect that differences in maintenance exist between the two kinds of systems because self-managing systems, for instance, have to (1) introduce pervasive monitoring functionality, (2) reify part of the system's state in an executable meta-model (i.e. reflectivity), and (3) include functionality that performs reasoning and adaptations at run-time. This means that system properties of a self-managing system have to be verified during run-time whenever the system dynamically changes or evolves. In contrast, for a static system the properties can be often assured with off-line analyses before deployment.

Self-adaptation in software-intensive systems comes in many different guises, including static and dynamic forms. In static self-adaptation all possible adaptations are explicitly defined by the designers for the system to choose from during execution whereas, in dynamic self-adaptation possible adaptations are determined and selected by the system at run-time. One way to characterize self-adaptive software systems is by the implementation mechanism used to achieve self-adaptation [52,53]. Static and dynamic mechanisms include design-time (e.g., architectural alternatives or hard-wired decision trees), compile-time (e.g., C++ templates or aspects), load-time (e.g., Java beans or plug-ins), and run-time (e.g., feedback loops or interpreters of scripting languages). Self-adaptive and self-managing systems are typically implemented using feedback mechanisms to control their dynamic behavior—for example, to keep web services up and running, to load-balance storage and compute resources, to diagnose and predict system failures, and to negotiate an obstacle course in a warehouse.

McKinley et al. distinguish between *parameter adaptation* and *compositional adaptation* [43]. Parameter adaptation modifies program variables that determine behavior whereas compositional adaptation exchanges algorithmic or structural system components with others to adapt to its environment. With compositional adaptation, an application can adopt new algorithms for addressing concerns that were unforeseen during development.

In this article, we mostly concentrate on a particular approach for realizing self-adaptive and self-managing systems, namely autonomic computing. In the next section we briefly outline the origins of autonomic computing and the community that has emerged around it. We then discuss selected issues of autonomic computing, including feedback control (cf. Section 5), autonomic element design (cf. Section 6), and architectures for autonomic systems (cf. Section 7). Even though this paper concentrates on autonomic computing, we believe that most of the findings also apply to most solutions dealing with dynamic and evolving computing systems in general. As Huebscher and McCann in their recent survey article on autonomic computing, in this article we basically use the terms autonomic system, self-Managing system, and self-adaptive system interchangeably [27].

## 4 Autonomic Computing Systems

A software system is called *autonomic*[1] if it operates mostly without human or external intervention or involvement according to a set of rules or policies. Such a system can, for example, self-configure at run-time to meet changing operating environments, self-tune to optimize its performance, recognize symptoms, determine root causes, and self-heal when it encounters unexpected obstacles during its operation.

The autonomic computing community often refers to the human Autonomic Nervous System (ANS) with its many control loops as a prototypical example [35]. The ANS monitors and regulates vital signs such as body temperature, heart rate, blood pressure, pupil dilation, digestion blood sugar, breathing rate, immune response, and many more involuntary, reflexive responses in our bodies. Furthermore the ANS consists of two separate divisions called the parasympathetic nervous system, which regulates day-to-day internal processes and behaviors and sympathetic nervous system, which deals with stressful situations. Studying the ANS might be instructive for the design of autonomic software systems; for example, physically separating the control loops which deal with normal and out-of-kilter situations might be a useful design idea for designing autonomic computing systems. Over the past decade distinct architectural patterns have emerged for specific autonomic problem domains. For example, Brittenham et al. have distilled common patterns for IT service management based on the best practices in the IT Infrastructure Library [4,31].

Autonomicity may be addressed at different points along a system-application dimension. Specifically, it can be built into a design at the hardware level (e.g., self-diagnosing hardware), the operating system level (e.g., self-configuring upgrades or security management), middleware (e.g., self-healing infrastructure), or the application level (e.g., self-tuning storage management, or optimization of application performance and service levels). Autonomicity is not an all-or-nothing property. One can conceive of software applications that require none, little, or considerable user input in order to self-manage at run-time. Kephart and Chess

[1] The Greek origin for autonomic ($\alpha\upsilon\tau o\nu o\mu o\varsigma$), literally means "self-governed".

in their seminal paper state that the essence of autonomic computing is system self-management, freeing administrators from low-level task management while delivering more optimal system behavior [35]. In many cases however, users and operators usually remain in the loop by assisting and approving self-management processes.

The IBM autonomic computing initiative has generated an impressive research community spanning academia and industry. Researchers have founded journals such as *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, produced special issues for established journals on the subject [24,8], and founded several conferences, including *IEEE International Conference on Autonomic Computing (ICAC), ACM/IEEE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), ACM International Conference on Autonomic Computing and Communication Systems (Autonomics), IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO),* and *IBM CASCON Workshop on Engineering Autonomic Systems.* The produced research results are significant and have influenced and penetrated many commercial products in different companies. The results have recently also been embraced by architectural strategists and standards communities [60], including web services, service-oriented architecture (SOA) [14], and ubiquitous computing [28].

IBM has defined the widely applicable autonomic computing reference architecture (ACRA) [28] and the highly practical autonomic toolkit [30] which together comprise a collection of components, tools, scenarios, and documentation designed for users wanting to learn, adapt, and develop autonomic behavior in their products and systems. Thus, after almost a decade of intense research and development in this realm, autonomic computing constitutes an effective set of technologies, models, architecture patterns, standards, and processes to cope with and reign in the management complexity of dynamic computing systems using feedback control, adaptation, and self-management.

## 5 Feedback Control for Autonomic Computing

While the term autonomic computing was coined at the beginning of this decade, many of the foundations of autonomic computing have a rich history in engineering [59], operations research [63], and artificial intelligence [54,11]. Feedback control is really the heart of a self-managing, autonomic or self-adaptive system. Feedback control, with its control theory history in many branches of engineering and mathematics, is about regulating the behavior of dynamical systems as depicted in Figure 2.

A feedback control system consists of a set of components that act together to maintain actual system attribute values close to desired specifications. The main idea is to sense output measurements to adjust control inputs to meet the specified goals. For example, a thermostat controls the temperature of a house by sensing the air temperature and by turning the heater or air conditioner on and off. Another canonical example frequently analyzed and implemented in control theory textbooks is the automotive cruise-control system [1,6,50]. Control

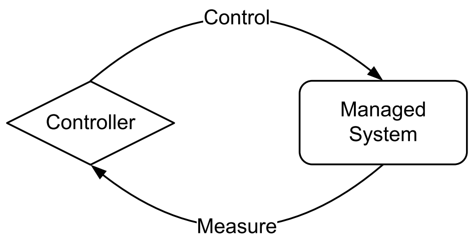

**Fig. 2.** Ubiquitous feedback loop

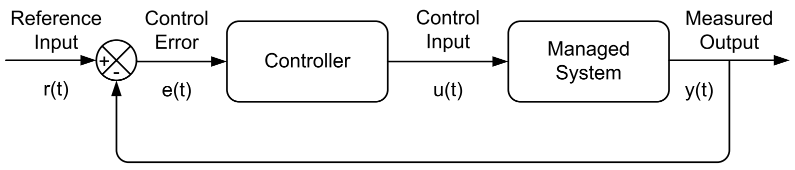

**Fig. 3.** Block diagram of a general feedback system [1,6,50]

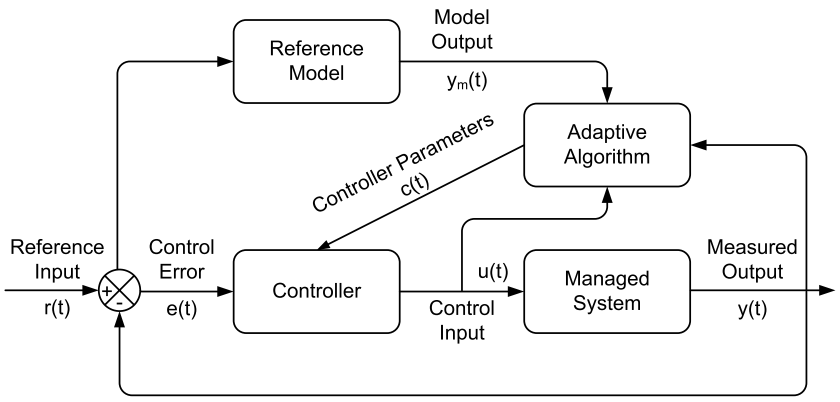

**Fig. 4.** Basic structure of model reference adaptive control (MRAC)

theory literature describes control systems using block diagrams and well-defined terminology as shown in Figure 3.

Control engineering textbooks contain many reference architectures for different application domains. One of the most common models is the model-reference adaptive control (MRAC) architecture depicted in Figure 4, which was originally proposed for the flight control problem [1,15]. MRAC is also known as model reference adaptive system (MRAS). MRAC implementations can be used to realize compositional adaptation.

Significant flexibility and leverage is achieved by defining the model and algorithm separately. Because of the separation of concerns, this solution is ideal for software engineering applications in general and self-adaptive or self-managed software-intensive systems in particular. While feedback control of this sort is common in the construction of engineered devices to bring about desired behavior despite undesired disturbances, it is not as frequently applied as one might hope for realizing dynamic computing systems—in part due to the inaccessibility of control theory for computing practitioners. With the publication of IBM's architectural blueprint for autonomic computing [28] and Hellerstein et al.'s book on *Feedback Control of Computing Systems* [23], computing practitioners are now in a much better position to address the dynamics of resource management and other related problems of autonomic computing.

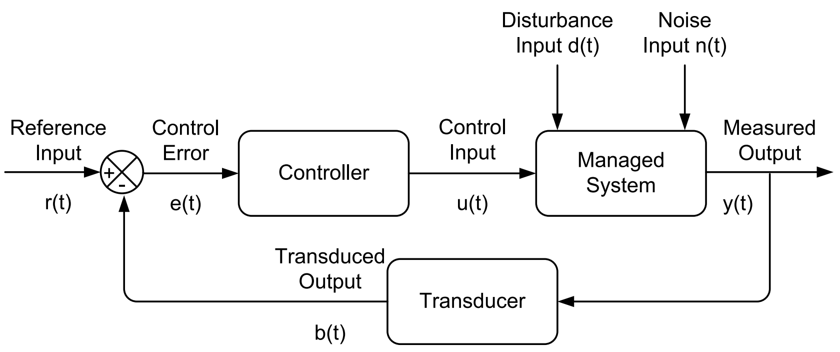

**Fig. 5.** Block diagram of a feedback control system with disturbance input for computing systems [23,12]

The essential components of a feedback control system suitable for dynamic computing systems according to Hellerstein et al. are depicted in Figure 5 [23]. The reference input is the desired value of the system's measured output. The controller adjusts the setting of control input to the managed system so that its measured output converges towards to the referenced input [23,12]. This feedback control encapsulates functionality identified in the generic control loops depicted in Figures 1 and 3, and in Figure 4 the MRAC reference model and adaptive algorithm are realized in the controller. The main components are as follows [23,12]:

- Control input is a parameter that affects the behavior of the managed system and can be adjusted dynamically.
- The controller determines the setting of the control input needed to achieve the reference input. The controller computes values for the control input based on the history (i.e., current and past values) of the control error (i.e., the difference between the reference input and the measured output).
- Disturbance input is any change that affects the way in which the control input influences the measured output. These are factors that affect the measured output but for which there is no direct governing control input (e.g., workload variations such as running back-ups, and downloading and updating virus definitions).
- Noise input is any effect that changes the measured output produced by the managed system. This is also called sensor noise or measurement noise.
- Measured output is a measurable characteristic of the managed system (e.g., response time).

– Reference input (also referred to as set point) is the desired value of the measured outputs.
– Managed system (also referred to as process or plant) is the computing system to be controlled.
– The transducer transforms the measured output so that it can be compared with the reference input (e.g., relating and converting output units to input units).

The application determines the control objective. Hellerstein et al. concentrate on regulatory control (e.g., to maintain reserve capacity), disturbance and noise rejection (e.g., to maximize throughput subject to guaranteed response times for different customers), and optimization (e.g., optimizing operations according to service level agreements and compliance requirements) [23]. The crux of the matter is the construction of a suitable model to quantify the control objective. Fields such as performance engineering and queuing theory have arrived at mature models. However, in many application domains are still at their infancy of modeling the relationships between controlled inputs and outputs.

The feedback loop is a central element of control theory, which provides well-established mathematical techniques to analyze system performance and correctness. It is not clear if general principles and properties established by this discipline (e.g., observability, controllability, stability, and hysteresis) are applicable for reasoning about autonomic systems. Generally, systems with a single control loop are easier to reason about than systems with multiple loops—but multi-loop systems are more common and have to be dealt with.

Good engineering practice calls for reducing multiple control loops to a single one—a common approach in control engineering, or making control loops independent of each other. If this is impossible, the design should make the interactions of control loops explicit and expose how these interactions are handled. Another typical scheme from control engineering is organizing multiple control loops in the form of a hierarchy where due to the employed different times unexpected interference between the levels can be excluded. This scheme seems to be of particular interest if we separate different forms of adaptation such as component management, change management, and goal management as proposed by Kramer and Magee [37,38].

In order to structure an autonomic system's control loops and to reason about them, the concept of autonomic element is helpful because it encapsulates a single control loop and provides an explicit abstraction for it.

## 6 The Autonomic Element

IBM researchers introduced the notion of an *autonomic element* as a fundamental building block for designing self-adaptive and self-managing systems [28,35]. An autonomic element consists of an autonomic manager (i.e., controller), a managed element (i.e., process), and two manageability interfaces.

The core of an autonomic manager constitutes a feedback loop, often referred to as monitor-analyze-plan-execute (MAPE) or monitor-analyze-plan-execute-knowledge (MAPE-K) loop, as depicted in Figure 6. The manager gathers measurements from the managed element as well as information from the current and past states from various knowledge sources via a service bus and then adjusts the managed element if necessary through a manageability interface (i.e., the sensors and effectors at the bottom of this figure) according to the control objective. Note that an autonomic element itself can be a managed element—the sensors and effectors at the top of the autonomic manager in Figure 6 are used to manage the element (i.e., provide measurements through its sensors and receive control input (e.g., rules or policies) through its effectors). If there are no such effectors, then the rules or policies are hard-wired into the MAPE loop. Even if there are no effectors at the top of the element, the state of the element is typically still exposed through its top sensors.

One of the greatest achievements and most compelling benefits of the autonomic computing community is the standardization of the manageability interfaces and manageability endpoints (i.e., a manageability endpoint exposes the state and the management operations for a resource or another autonomic element) across a variety of managed resources and the standardization of the information (e.g., events or policies) that is passed through these interfaces [60].

IN 2006 OASIS Web Services Distributed Management Technical Committee (WSDM TC) approved and published two sets of specifications on Web Services Distributed Management entitled Management Using Web Services (MUWS) and Management of Web Services (MOWS) [49]. The standardization of WSDM 1.0 is an important milestone for autonomic computing because it defines the web services endpoints. Web services endpoints are a necessary technology for autonomic managers to bring self-managing capabilities to hardware and software resources of the IT infrastructure.

The autonomic manager is a controller, which controls the managed element (i.e., a set of resources or other autonomic elements). Thus, the autonomic manager and the managed element of an autonomic element correspond to the controller and the managed system in the general feedback system (cf. Figure 3). This connection is depicted in Figure 7 below.

The autonomic element structure can also be characterized as MIAC or MRAC structure (cf. Section 5). The MIAC or MRAC reference model is stored in the knowledge base and the adaptive algorithm is decomposed into the four MAPE components.

The controller, with its MAPE loop, operates in four phases over a knowledge base to assess the current state of the managed elements, predict future states, and bring about desired behavior despite disturbances from the environment and changes in the process. 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 [29]. The planner interprets the symptoms and devises a plan to execute the change in the managed process through the effectors.

While there is plenty of existing literature for designing and implementing each MAPE component (e.g., artificial intelligence planning algorithms), the

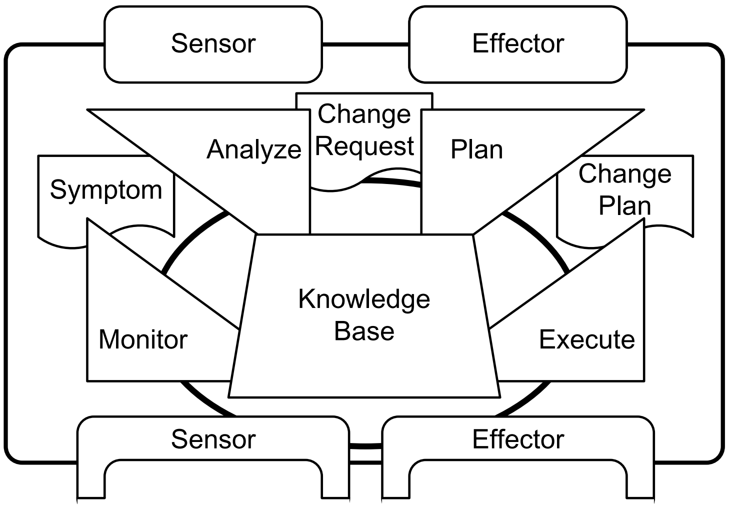

**Fig. 6.** Autonomic Manager [28,35]

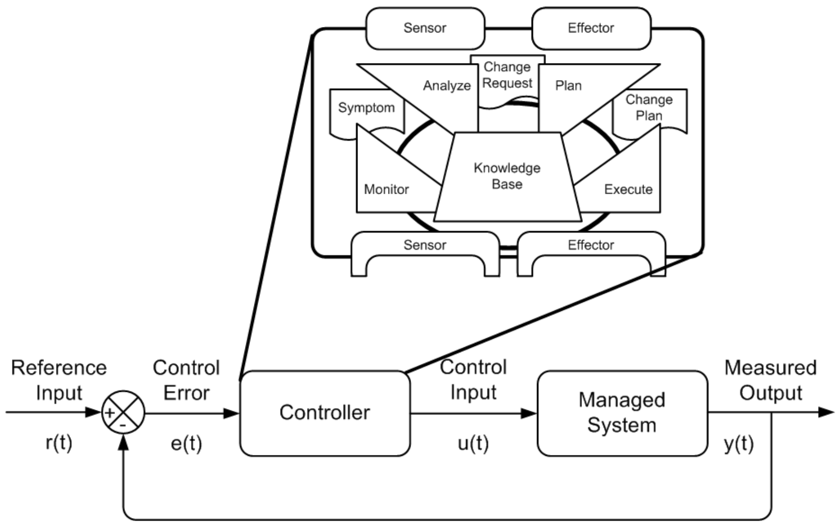

**Fig. 7.** Autonomic Manager as Controller in Feedback Loop

modeling of control objectives is hard. Therefore, human operators are often kept in the loop (i.e., performing some of the tasks of the MAPE loop manually). For example, for root cause analysis in enterprise systems, the monitor and analyze tasks are frequently fully automated whereas the plan and execute tasks are completed by experienced human operators. Even if all components of a MAPE loop can be executed without human assistance, the autonomic manager can typically be configured to perform only a subset of its automated functions.

The four components of a MAPE loop work together by exchanging knowledge to achieve the control objective. An autonomic manager maintains its own knowledge (e.g., information about its current state as well as past states) and has access to knowledge which is shared among collaborating autonomic managers (e.g., configuration database, symptoms database, business rules, provisioning policies, or problem determination expertise).

To realize an autonomic system, designers build an arrangement of collaborating autonomic elements working towards a common goal. In this case, the maintainability concerns increase significantly due to the interactions of the control loops of the different autonomic elements. To synthesize arrangements of autonomic elements, designers employ various techniques including goal models [39,40] and architectural patterns [64].

## 7 Autonomic Reference Architecture and Patterns

Autonomic system solutions consist of arrangements of interdependent, collaborative autonomic elements. A hierarchy of autonomic managers where higher level managers orchestrate lower level managers or resources is the most common and natural arrangement. For example, it reflects how organizations work or how IT professionals organize their tasks [4].

The autonomic computing reference architecture (ACRA) is a three-layer hierarchy of orchestrating managers, resource managers, and managed resources which all share management data across an enterprise service interface [28] as depicted in Figure 8. The ACRA also includes manual managers or operators, who have access to all levels through dashboards or consoles [42]. Hierarchical arrangements of autonomic managers also afford separation of concerns (e.g., managing performance, availability, or capacity relatively independently).

Brittenham et al. define a set of autonomic architecture and process patterns called delegation for progressing from manual management, to autonomic management [4]. Some of the intermediate stages include automated assistance, supervised delegation, conditional delegation, task delegation, and full loop delegation. For example, in conditional delegation the IT professional trusts an autonomic manager to perform some but not all requests whereas in task delegation the IT professional trusts an autonomic manager to perform a complete task.

Kramer and Magee proposed a three layer reference architecture for self managed systems consisting of goal management, change management, and component management [38]. They follow Gat's architecture for autonomous robotics systems [20] and address different time scales of management or control with the three levels. Immediate feedback actions are controlled at the component level and the longest actions requiring deliberation are at the goal level.

Litoiu et al. use three levels autonomic management for provisioning, application tuning, and component tuning [41]. Another typical scheme from control engineering is organizing multiple control loops in the form of a hierarchy where, due to the employed different time periods, unexpected interference between the levels can be excluded.

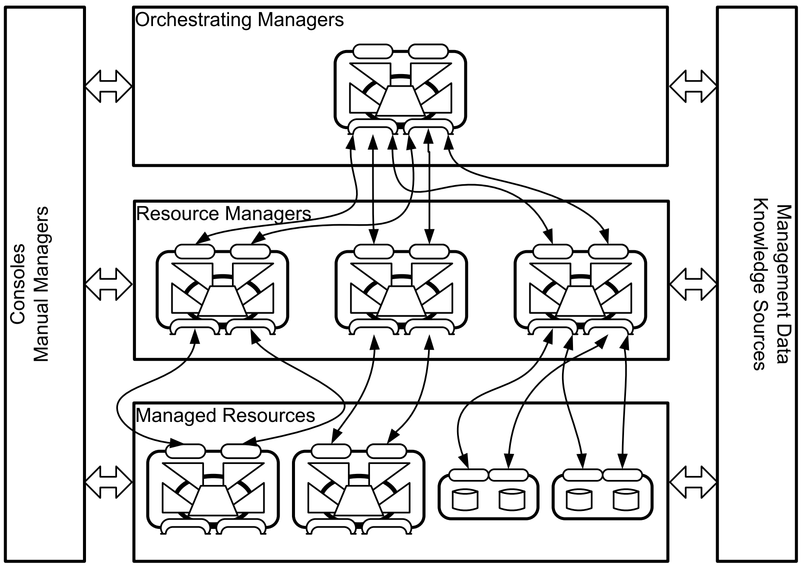

**Fig. 8.** Autonomic computing reference architecture (ACRA) [28]

Lapouchnian et al. proposed a novel design technique for autonomic systems based on goal-driven requirements engineering which results in more complicated autonomic element topologies [39,40]. Goal models capture a spectrum of alternative system behaviors and/or system configurations, which are all delivering the same functionality and are able to select at run-time the best behavior or configuration based on the current context. Goal models provide a unifying intentional view of the system by relating goals assigned to autonomic elements to high-level system objectives and quality concerns. This method produces an arrangement of autonomic elements that is structurally similar to the goal hierarchy of the corresponding goal model. One extreme solution is to realize the entire goal model using a single autonomic element. On the other side of the spectrum is a solution that allocates an autonomic element for each node in the goal model. A combination of these two extreme mappings appears to be most practical. Realizing a subtree (i.e., rather than a single node) of a goal model using an autonomic element seems more appropriate for many applications. Trade-offs among goal subtrees can be tuned and optimized using higher-level autonomic elements. Thus, a goal model affords a first architectural decomposition of a self-managing system into components. Of course, much design information has to be added to implement the four phases of each autonomic element.

Out of a need to include autonomic features into legacy applications, Chan and Chieu proposed an autonomic design that treats the monitoring of a software application as a separate and independent concern [7]. This approach utilizes aspect-oriented programming to intercept, analyze, and decompose the application states of a software application before linking appropriate non-invasive constructs into the software application code to provide application state information to an autonomic manager through sensors. Kaiser et al. proposed a similar approach to externalizing monitoring logic [33].

Garlan's Rainbow framework, which is based on reusable infrastructure, also uses external adaptation mechanisms to allow the explicit specification of adaptation strategies for multiple system concerns. [18,19]. The Rainbow project investigated the use of software architectural models at runtime as the basis for reflection and dynamic adaptation.

Zhu et al. [64] identify and categorize types of common forms of autonomic element patterns grounded in Hawthorne and Perry's architectural styles [21,22], and Neti and Müller's attribute based architectural styles (ABASs) [48]. They also reveal the inherent relationships among these patterns and assess their particular maintainability concerns.

The field of autonomic computing has produced control-centric as well as architecture-centric reference models and patterns. To this point little research has been devoted to compare and evaluate the advantages and disadvantages of these two strategies. The selected research challenges outlined in the next section present good steps in this direction.

## 8 Lessons Learned and Research Challenges

While much progress has been made over the past decade, many open research problems and challenges remain in this exciting field of autonomic computing.

**Model construction**
The process of designing feedback-based computing systems requires the construction of models which quantify the effect of control inputs on measured outputs [23]. Fields such as performance engineering and queuing theory have developed advanced models for many different applications. However, performance, albeit important, constitutes just one dimension of the modeling problem. There are many other quality criteria that come into play in dynamic autonomic applications. For some of these criteria (e.g., trust) quantification is difficult [48]. In addition, models are needed to design trade-off analyses schemes for combinations of quality criteria. Thus, developing feedback models using quality criteria for various application domains is a major challenge and critical for the success of this field. Models and quality criteria related to governance, compliance, and service-level agreements are of particular importance for service-oriented autonomic business processes and applications.

**Managing and leveraging uncertainty**
When we model potential disturbances from the environment of an autonomic system (e.g., unexpected saturation of the network) or satisfy requirements by regulation (i.e., trade-off analysis among several extra-functional requirements), we introduce some uncertainty. Therefore, designers and maintainers of such dynamical systems should manage uncertainty because the environment may change in unexpected ways and, as a result, the system may adapt

in such a way that was not foreseeable at design time. Introducing uncertainty requires trade-offs between flexibility and assurance [9]. For a maintainer it is critical to know which parts of the environment are assumed to be fixed and which are expected to introduce uncertainty [64]. Moreover, assurance and compliance criteria should be continuously validated at run-time (i.e., not just at system acceptance time). Hence, understanding, managing, and leveraging uncertainty is important for delivering autonomic systems with reliability and assurance guarantees.

**Making control loops explicit**

Software engineers are trained to develop abstractions that hide complexity. Hiding the complexity of a feedback loop seems obvious and natural. In contrast, Müller et al. advocate that feedback loops, which are the bread and butter of self-adaptive and autonomic systems, should be made first class design elements [47]. Designers of autonomic systems will realize significant benefits by raising the visibility of control loops and specifying the major components and characteristics of the control loops explicitly. Further benefits could be realized by identifying common forms of adaptation and then distilling design and V&V obligations for specific patterns. When arrangements of multiple control loops interact as in the ACRA (cf. Section 7), system design and analysis should cover their interactions. As control grows more complex, it is especially important for the control loops to be explicit in design and analysis. Hence, it is useful to investigate the trade-offs between hiding the complexity of feedback loops and treating feedback loops as first class objects with respect to the construction and operation of autonomic systems.

**Characterizing architectural patterns and analysis frameworks**

Another worthwhile avenue of research is to investigate patterns, quality criteria, and analysis frameworks for self-managing applications for different business and evolution contexts [48,64,10]. Quality attributes should not only include traditional quality criteria such as variability, modifiability, reliability, availability and security, but also autonomicity-specific criteria such as dynamic adaptation support, dynamic upgrade support, support for detecting anomalous system behavior, support for how to keep the user informed and in the loop, support for sampling rate adjustments in sensors, support for simulation of expected or predicted behavior, support for differencing between expected and actual behavior, or support for accountability (e.g., how users can gain trust by monitoring the underlying autonomic system).

## 9 Conclusions

On the one hand, dealing with software-intensive ecosystems seems rather daunting given that there are still many challenges in the top-down construction of traditional software systems and their subsequent maintenance. On the other hand, to be able to observe and control independent and competing processes in a dynamic and changing environment is a necessity in this new world of service-oriented, decentralized, distributed computing ecosystems.

After almost a decade of intense research and development, autonomic computing constitutes an effective set of technologies, models, architecture patterns, standards, and processes to cope with and reign in the management complexity of dynamic computing systems using feedback control, adaptation, and self-management. The feedback loops, which are at the core of an autonomic system, sense their environment, model their behavior in that environment, and take action to change the environment or their own behavior. In this article, we argued that we not only need architecture-centric views, but also control-centric views to construct, reason about, and operate highly dynamic autonomic systems. We illustrated that autonomic computing technology is not only useful for managing IT infrastructure complexity, but also to mitigate continuous software evolution problems. We also discussed how to leverage the rich history of control theory foundations for autonomic and self-adaptive systems. Finally, we presented some valuable lessons learned and outlined selected challenges for the future.

Previously, we advocated to teach the notion of a control loop, together with the concept of an autonomic element, early in computer science and software engineering curricula [46]. In contrast to traditional engineering disciplines, computing science programs seem to have neglected the control loop. If we assume that the two important trends dominating the computing world over the last decade and outlined at the beginning of this article—moving towards a service-centric way of conducting business and evolving towards socio-technical software ecosystems,—it might be time to give the control loop more prominence in undergraduate computing curricula so that the next generation of software engineers and IT professionals is intimately familiar with the foundations of self-adaptation and self-management. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [57]. Not surprisingly, the process control pattern described in that paper resembles an autonomic element. This method could be used to teach simple feedback loops in an introductory programming course.

The papers listed in the bibliography give a good indication of how quickly the field of autonomic computing has grown since 2001, but also show that there is a rich history for many of its foundations. The references contain a number of papers that can serve as good starting points for newcomers to the field (i.e., [4,9,12,13,16,23,28,35,38,51,60]).

## Acknowledgments

This work grew out of collaboration with colleagues at IBM Toronto Center for Advanced Studies, CA Labs, and Carnegie Mellon Software Engineering Institute. We are also deeply indebted to many of friends and students, who contributed significantly to our appreciation and understanding of autonomic computing. In particular, we would like to thank John Mylopoulos, Marin Litoiu, Anatol Kark, Dennis Smith, Grace Lewis, Patricia Oberndorf, Mary Shaw, Linda Northrop, Ken Wong, Peggy Storey, Kostas Kontogiannis, Gabby Silbermann, Serge Mankovski, Joanna Ng, Kelly Lyons, Cheryl Morris, Scott Tilley, Andrea

De Lucia, Filomena Ferrucci, Qin Zhu, Piotr Kaminski, Sangeeta Neti, Sweta Gupta, and Dylan Dawson for their continued collaboration and support. This work was funded in part by the National Sciences and Engineering Research Council (NSERC) of Canada (CRDPJ 320529-04 and CRDPJ 356154-07) as well as IBM Corporation and CA Inc. via the CSER Consortium.

## References

1. Astrom, K.J., Wittenmark, B.: Adaptive Control, 2nd edn. Addison-Wesley, Reading (1995)
2. 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)
3. Boehm, B.: A View of 20th and 21st Century Software Engineering. In: 28th ACM/IEEE International Conference on Software Engineering (ICSE 2006), pp. 12–29. ACM, New York (2006)
4. 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)
5. Broy, M., Jarke, M., Nagl, M., Rombach, D.: Manifest: Strategische Bedeutung des Software Engineering in Deutschland. Informatik-Spektrum 29(3), 210–221 (2006)
6. Burns, R.S.: Advanced Control Engineering. Butterworth-Heinemann (2001)
7. Chan, H., Chieu, T.C.: An Approach to Monitor Application Status for Self-Managing (Autonomic) Systems. In: 18th ACM Annual SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2003), pp. 312–313. ACM, New York (2003)
8. Chao, L.: Special Issue on Autonomic Computing. Intel Technology Journal 10(4), 253–326 (2006)
9. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, J., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Malek, S., Mirandola, R., Müller, H.A., Park, S., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: A Research Roadmap: Software Engineering for Self-Adaptive Systems. Schloss Dagstuhl Seminar 08031 Report on Software Engineering for Self-Adaptive Systems. Schloss Dagstuhl Seminar 08031 Report on Software Engineering for Self-Adaptive Systems, Wadern, Germany, 12 pages (2008); ACM/IEEE International ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008). ACM, New York (2008), `http://www.dagstuhl.de/08031/`
10. Dawson, R., Desmarais, R., Kienle, H.M., Müller, H.A.: Monitoring in Adaptive Systems Using Reflection. In: 3rd ACM/IEEE International ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2008), pp. 81–88. ACM, New York (2008)
11. De Kleer, J., Mackworth, A.K., Reiter, R.: Characterizing Diagnosis and Systems. Artificial Intelligence 56(23), 197–222 (1992)
12. Diao, Y., Hellerstein, J.L., Parekh, S., Griffith, R., Kaiser, G.E., Phung, D.: A Control Theory Foundation for Self-Managing Computing Systems. IEEE Journal on Selected Areas in Communications 23(12), 2213–2222 (2005)

13. Dobson, S., Denazis, S., Fernández, A., Gaiti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A Survey of Autonomic Communications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1(2), 223–259 (2006)
14. Draper, C.: Combine Autonomic Computing and SOA to Improve IT Management. IBM Developer Works (2006), `http://www.ibm.com/developerworks/library/ac-mgmtsoa/index.html`
15. Dumont, G.A., Huzmezan, M.: Concepts, Methods and Techniques in Adaptive Control. In: American Control Conference (ACC 2002), vol. 2, pp. 1137–1150. IEEE Computer Society, Washington (2002)
16. Ganek, A.G., Corbi, T.A.: The Dawning of the Autonomic Computing Era. IBM Systems Journal 42(1), 5–18 (2003)
17. Ganek, A.G.: Overview of Autonomic Computing: Origins, Evolution, Direction. In: Parashar, M., Hariri, S. (eds.) Autonomic Computing: Concepts, Infrastructure, and Applications. CRC Press, Boca Raton (2006)
18. Garlan, D., Cheng, S., 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)
19. Garlan, D., Cheng, S., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. IEEE Computer 37(10), 46–54 (2004)
20. Gat, E.: On Three-layer Architectures. In: Kortenkamp, D., Bonasso, R., Murphy, R. (eds.) Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems. MIT/AAAI Press (1998)
21. Hawthorne, M.J., Perry, D.E.: Exploiting Architectural Prescriptions for Self-Managing, Self-Adaptive Systems: A Position Paper. In: 1st ACM SIGSOFT Workshop on Self-Managed Systems (WOSS 2004), pp. 75–79 (2004)
22. Hawthorne, M.J., Perry, D.E.: Architectural Styles for Adaptable Self-healing Dependable Systems. Technical Report, University of Texas at Austin, USA (2005), `http://users.ece.utexas.edu/~perry/work/papers/MH-05-Styles.pdf`
23. Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D.M.: Feedback Control of Computing Systems. John Wiley & Sons, Chichester (2004)
24. Herger, L., Iwano, K., Pattnaik, P., Davis, A.G., Ritsko, J.J.: Special Issue on Autonomic Computing. IBM Systems Journal 42(1), 3–188 (2003), `http://www.research.ibm.com/journal/sj42-1.html`
25. Hewlett-Packard Development Company: HP Unveils Adaptive Enterprise Strategy to Help Businesses Manage Change and Get More from Their IT Investments (2003), `http://www.hp.com/hpinfo/newsroom/press/2003/030506a.html`
26. Horn, P.: Autonomic Computing. Online Whitepaper (2001), `http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf`
27. Huebscher, M.C., McCann, J.A.: A Survey of Autonomic Computing—Degrees, Models, and Applications. ACM Computing Surveys 40(3), 7, 1–28 (2008)
28. IBM Corporation: An Architectural Blueprint for Autonomic Computing, 4th edn. (2006), `http://www-03.ibm.com/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf`
29. IBM Corporation: Symptoms Reference Specification V2.0 (2006), `http://download.boulder.ibm.com/ibmdl/pub/software/dw/opensource/btm/SymptomSpec_v2.0.pdf`
30. IBM Corporation: Autonomic Computing Toolkit User's Guide, 3rd edn. (2005), `http://download.boulder.ibm.com/ibmdl/pub/software/dw/library/autonomic/books/fpu3mst.pdf`

31. Information Technology Infrastructure Library (ITIL). Office of Government Commerce, UK (2007), `http://www.itil.org.uk/`
32. Inverardi, P., Tivoli, M.: The Future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008, University of Salerno, Italy. LNCS, vol. 5413, pp. 1–31. Springer, Heidelberg (2009)
33. Kaiser, G., Parekh, J., Gross, P., Valetto, G.: Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems. In: 5th IEEE Annual International Active Middleware Workshop (AMS 2003), pp. 22–30. IEEE Computer Society, Washington (2003)
34. Kazman, R., Chen, H.-M.: The Metropolis Model: A New Logic for the Development of Crowdsourced Systems. Communications of the ACM, 18 pages (submitted, 2008)
35. Kephart, J.O., Chess, D.M.: The Vision of Autonomic Computing. IEEE Computer 36(1), 41–50 (2003)
36. Kluth, A.: Information Technology—Make It Simple. The Economist (2004), `http://www.economist.com/surveys/displaystory.cfm?story_id=E1_PPDSPGP&CFID=17609242&CFTOKEN=84287974`
37. Kramer, J., Magee, J.: Dynamic Structure in Software Architectures. ACM SIGSOFT Software Engineering Notes 21(6), 3–14 (1996)
38. Kramer, J., Magee, J.: Self-managed Systems: An Architectural Challenge. In: Future of Software Engineering (FoSE 2007), pp. 259–268. IEEE Computer Society, Washington (2007)
39. Lapouchnian, A., Liaskos, S., Mylopoulos, J., Yu, Y.: Towards Requirements-Driven Autonomic Systems Design. In: ICSE Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005), pp. 45–51. ACM, New York (2005)
40. Lapouchnian, A., Yu, Y., Liaskos, S., Mylopoulos, J.: Requirements-Driven Design of Autonomic Application Software. In: ACM IBM Center for Advanced Studies Conference on Collaborative Research (CASCON 2006), pp. 80–93. ACM, New York (2006)
41. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical Model-based Autonomic Control of Software Systems. In: ICSE Workshop on Design and Evolution of Autonomic Application Software (DEAS 2005), pp. 34–40. ACM, New York (2005)
42. Marcus, A.: Dashboards in Your Future. ACM Interactions 13(1), 48–49 (2006)
43. McKinley, P.K., Sadjadi, M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. IEEE Computer 37(7), 56–64 (2004)
44. Microsoft Corporation: Dynamic Systems 2007: Get Started With Dynamic Systems Technology Today (2007), `http://www.microsoft.com/business/dsi/dsiwp.mspx`
45. Müller, H.A., O'Brien, L., Klein, M., Wood, B.: Autonomic Computing. Technical Report, Software Engineering Institute, Carnegie Mellon University, CMU/SEI-2006-TN-006, 61 pages (2006), `http://www.sei.cmu.edu/pub/documents/06.reports/pdf/06tn006.pdf`
46. Müller, H.A.: Bits of History, Challenges for the Future and Autonomic Computing Technology (Keynote). In: 13th IEEE Working Conference on Reverse Engineering (WCRE 2006), pp. 9–18. IEEE Computer Society, Washington (2006)
47. Müller, H.A., Pezzè, M., Shaw, M.: Visibility of Control in Adaptive Systems. In: 2nd ACM/IEEE International ICSE Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), pp. 23–26. ACM, New York (2008)
48. Neti, S., Müller, H.A.: Quality Criteria and an Analysis Framework for Self-Healing Systems. In: 2nd ACM/IEEE International ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007), pp. 39–48. IEEE Computer Society, Washington (2007)

49. OASIS: Web Services Distributed Management: Management of Web Services (WSDM-MOWS) 1.1 OASIS Standard (2006), http://docs.oasis-open.org/wsdm/wsdm-mows-1.1-spec-os-01.htm
50. Ogata, K.: Discrete-Time Control Systems, 2nd edn. Prentice-Hall, Englewood Cliffs (1995)
51. Northrop, L., Feiler, P., Gabriel, R., Goodenough, J., R., L., Longstaff, T., Kazman, R., Klein, M., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-Large-Scale Systems—The Software Challenge of the Future. Technical Report, Software Engineering Institute, Carnegie Mellon University, 134 pages (2006), http://www.sei.cmu.edu/uls
52. Oreizy, P., Medvidovic, N., Taylor, R.N.: Architecture-Based Runtime Software Evolution (Most Influential Paper Award at ICSE 2008). In: ACM/IEEE International Conference on Software Engineering (ICSE 1998), pp. 177–186. IEEE Computer Society, Washington (1998)
53. Oreizy, P., Medvidovic, N., Taylor, R.N.: Runtime Software Adaptation: Framework, Approaches, and Styles. In: ACM/IEEE International Conference on Software Engineering (ICSE 2008), pp. 899–910. ACM, New York (2008)
54. Reiter, R.: A Theory of Diagnosis from First Principles. Artificial Intelligence 32(1), 57–95 (1987)
55. Schloss Dagstuhl Seminar 08031. Software Engineering for Self-Adaptive Systems, Wadern, Germany (2008), http://www.dagstuhl.de/08031/
56. SEI Software-Intensive Systems (ISIS). Integration of Software-Intensive Systems (ISIS) Initiative: Addressing System-of-Systems Interoperability (2007), http://www.sei.cmu.edu/isis/
57. Shaw, M.: Beyond Objects: A Software Design Paradigm Based on Process Control. ACM SIGSOFT Software Engineering Notes 20(1), 27–38 (1995)
58. Sun Microsystems: Sun N1 Service Provisioning System (2007), http://www.sun.com/software/products/service_provisioning/index.xml
59. 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/
60. Tewari, V., Milenkovic, M.: Standards for Autonomic Computing. Intel. Technology Journal 10(4), 275–284 (2006)
61. Truex, D., Baskerville, R., Klein, H.: Growing Systems in Emergent Organizations. Communications of the ACM 42(8), 117–123 (1999)
62. Wong, K.: The Reverse Engineering Notebook. Ph.D. Thesis, Department of Computer Science, University of Victoria, Canada, 149 pages (1999) ISBN:0-612-47299-X
63. Tsypkin, Y.Z.: Adaptation and Learning in Automatic Systems. Mir. Publishing, Moscow (1971)
64. Zhu, Q., Lin, L., Kienle, H.M., Müller, H.A.: Characterizing Maintainability Concerns in Autonomic Element Design. In: IEEE International Conference on Software Maintenance (ICSM 2008), 10 pages. IEEE Computer Society, Washington (2008)