On Patterns for Decentralized Control in Self-Adaptive Systems

Danny Weyns[1], Bradley Schmerl[2], Vincenzo Grassi[3], Sam Malek[4], Raffaela Mirandola[5], Christian Prehofer[6], Jochen Wuttke[7], Jesper Andersson[1], Holger Giese[8], and Karl Göschka[9]

[1] Linnaeus University, Växjö, Sweden
[2] Carnegie Mellon University, Pittsburgh, PA, USA
[3] Università di Roma TorVergata, Italy
[4] George Mason University, Washington DC, USA
[5] Politecnico di Milano, Milan, Italy
[6] LMU München and Fraunhofer ESK, Germany
[7] University of Washington, WA, USA
[8] Univerisität Potsdam, Germany
[9] Technische Universität Wien, Austria

Abstract. Self-adaptation is typically realized using a control loop. One prominent approach for organizing a control loop in self-adaptive systems is by means of four components that are responsible for the primary functions of self-adaptation: Monitor, Analyze, Plan, and Execute, together forming a MAPE loop. When systems are large, complex, and heterogeneous, a single MAPE loop may not be sufficient for managing all adaptation in a system, so multiple MAPE loops may be introduced. In self-adaptive systems with multiple MAPE loops, decisions about how to decentralize each of the MAPE functions must be made. These decisions involve how and whether the corresponding functions from multiple loops are to be coordinated (e.g., planning components coordinating to prepare a plan for an adaptation). To foster comprehension of self-adaptive systems with multiple MAPE loops and support reuse of known solutions, it is crucial that we document common design approaches for engineers. As such systematic knowledge is currently lacking, it is timely to reflect on these systems to: (a) consolidate the knowledge in this area, and (b) to develop a systematic approach for describing different types of control in self-adaptive systems. We contribute with a simple notation for describing interacting MAPE loops, which we believe helps in achieving (b), and we use this notation to describe a number of existing patterns of interacting MAPE loops, to begin to fulfill (a). From our study, we outline numerous remaining research challenges in this area.

1 Introduction

Self-adaptive systems have the ability to adapt themselves to changes in their execution environment and internal dynamics, such as response to failure, variability in available resources, or changing user priorities, to continue to achieve their goals.

Examples of self-adaptive systems are those that optimize their performance under changing operating conditions, and systems that heal themselves when certain components fail. Feedback control loops have been identified as crucial elements in realizing self-adaptation of software systems [46,28,11]. One prominent approach to organizing a control loop in self-adaptive sytems is by means of four components that are responsible for the primary functions of self-adaptation: Monitor, Analyze, Plan, and Execute, often referred to as the MAPE loop [28]. When systems are large, complex, and heterogeneous, a single MAPE loop may not be sufficient for managing adaptation [9,1]. In such cases, multiple MAPE loops may be employed that manage different parts of the system. In self-adaptive systems with multiple MAPE loops, the functions for monitoring, analyzing, planning, and effecting may be made by multiple components that coordinate with one another. That is, the functions may be *decentralized* throughout the multiple MAPE loops. An example is a self-adaptive system in which multiple planning components coordinate with one another to prepare a plan for adaptation.

Different patterns of interacting control loops have been used in practice by centralizing and decentralizing the functions of self-adaption in different ways. For example, in the Rainbow framework [17], monitoring and execution are delegated to the different nodes of the controlled system, whereas analyzing and planning are centralized. The IBM architectural blueprint [25] organizes MAPE loops hierarchically, where each level of the hierarchy contains instances of all four MAPE components. In this setting, higher level MAPE loops determine the set values for the subordinate MAPE loops. In fully decentralized settings, relatively independent MAPE components coordinate with one another and adapt the system when needed. An example of this approach is discussed in [18], in which local component managers on different nodes coordinate with one another to (re-)configure the structure of the managed system according to the overall architectural specification.

The existing self-adaptive literature and research, in particular those with a software engineering perspective, have by and large tackled the problem of managing either local or distributed software systems in a centralized or hierarchical fashion, e.g., [40,17,25]. While increasing attention is given to decentralized control of self-adaptive software, e.g., [12,18,36,5,56,48,53], we believe that there is a dearth of practical and effective techniques to build systems in this fashion. However, there is an opportunity to build on the work of decentralized self-adaptation to understand the recurrent coordination patterns and trade-offs, so that systematic design of self-adaptive systems can be acheived.

To foster comprehension of self-adaptive systems with multiple control loops, and support reuse of known solutions in this area, it is crucial that we document common design approaches for engineers [47]. However, systematic knowledge about interacting control loops for self-adaptive systems is lacking. Therefore, it is timely to reflect on these systems to: (a) consolidate the knowledge in this area, and (b) develop a systematic approach for describing different types of control in self-adaptive systems. In this chapter, we contribute with a simple notation for describing multiple interacting MAPE loops, which we believe helps in achieving (b), and we use this notation to describe a number of well-known patterns of interacting MAPE loops, to begin to fulfill (a). Patterns are an established approach for documenting systematic knowledge in a particular area. A pattern describes a generic solution for a recurring design problem. The patterns we present are derived from common knowledge in the field of self-adaptation and experiences of the authors with building self-adaptive systems. In reflecting about these patterns and the different ways of organizing self-adaptive control loops, we have identified a number of further research challenges that together form a roadmap for achieving a more principled approach to designing decentralized self-adaptive systems. This roadmap is outlined in the conclusion of this chapter.

2 Terminology

Before we elaborate on the notation for interacting MAPE loops and the patterns, we first clarify terminology. In particular, we (1) explain the distinction between managed and managing subsystems, the two constituent parts of a self-adaptive system, and (2) clarify how we use the terms distribution and decentralization in this chapter, two terms that are often mixed up by software engineers in the community of self-adaptive systems, leading to a lot of confusion.

2.1 Managed and Managing Subsystem

As shown in Figure 1, a self-adaptive system is situated in an environment. We use the general terms *managed* subsystem and *managing* subsystem to denote the constituent parts of a self-adaptive software system. Other authors make a similar distinction. For example, in the Rainbow framework [17], the managed subsystem maps to the system layer and the managing subsystem to the architecture layer. The authors in [43] use core function to refer to the managed subsystem and adaptation engine to refer to the managing subsystem. In FORMS [57], the managed subsystem corresponds to the base-level subsystem, and the managing subsystem to the reflective subsystem.

The environment refers to the part of the external world with which the self-adaptive system interacts, and in which the effects of the system will be observed and evaluated [26]. The environment may correspond to both physical and software entities. For example, the environment of a robotic system includes physical entities like obstacles on the robot's path and other robots, as well as external cameras and corresponding software drivers. The distinction between the environment and the self-adaptive system is made based on the extent of control. For instance, in the robotic system, the self-adaptive system may interface with the mountable camera sensor, but since it does not manage (adapt) its functionality, the camera is considered to be part of the environment.

The managed subsystem comprises the application logic that provides the system's domain functionality. For instance, in the case of robots, navigation of a robot or transporting loads is performed by the managed subsystem. To realize its functionality, the managed subsystem monitors and affects the environment.

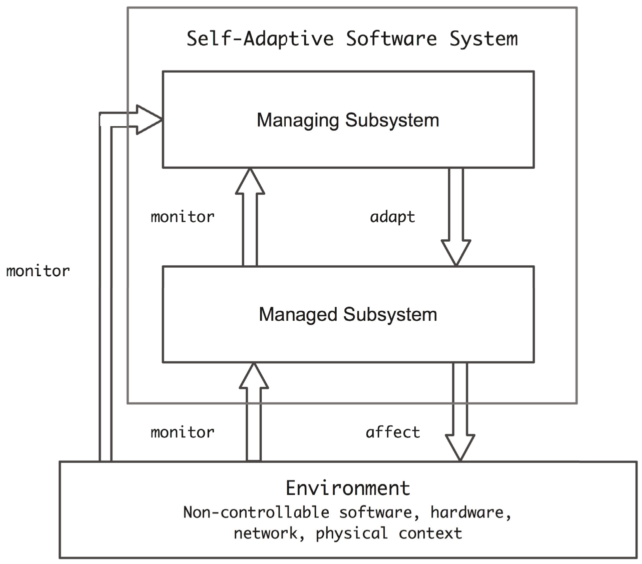
Self-Adaptive Software System
Managing Subsystem
monitor
adapt
monitor
Managed Subsystem
monitor
affect
Environment
Non-controllable software, hardware, network, physical context

Fig. 1. Constituent parts of a self-adaptive software system

To support adaptations, the managed subsystem has to provide support for monitoring and executing adaptations.

The managing subsystem manages the managed subsystem. The managing subsystem comprises the adaptation logic that deals with one or more concerns. For instance, a robot may be equipped with a managing subsystem that allows adaption of its navigation strategy based on the changing operation conditions, e.g., changing task load, or reduced bandwidth for communication. To realize its goals, the managing subsystem monitors the environment and the managed subsystem and adapts the latter when necessary.

Other layers can be added to the system where higher-level managing subsystems manage underlying subsystems, which can be managing subsystems themselves. For instance, consider a robot that not only has the ability to adapt its navigation strategy, but also to adapt the way such adaptation decisions are made, e.g., based on remaining energy level of the battery. In such an instance, the subsystem responsible for managing the battery level of the robot must coordinate with the subsystem for managing navigation and other robotics tasks, so that the robot does not fail entirely.

It is important to note that the managed and managing subsystems can be interwoven, as is the case when adaptation logic is dispersed throughout the functional logic the system. In such systems, it is not possible to easily reason about adaptation logic separately from system logic, meaning it is difficult to provide assurances or guarantees on the behavior of the system to changes in the environment. Another emerging approach to self-adaptation is in the field of self-organizing systems, where adaptation comes entirely from decisions made locally by components of the system. In such systems, the global properties of the adaptation (e.g., performance, utility to the user, or failure properties) are also difficult to reason about, though there is some research that attempts to address these concerns (for example, in [51], the authors present a statistical model that allows the convergence of global system objectives based on local agent behaviors to be analyzed, predicted, and controlled). In this chapter, we focus on how to organize self-adaptive systems where both subsystems are separate entities, following the principle of disciplined split [34] (or separation of concerns), which has been a main focus in the self-adaptive research community [55].

2.2 Distribution and Decentralization

Textbooks on distributed systems, e.g., [49], typically differentiate between: (1) centralized data in contrast to distributed, partitioned, and replicated data, (2) centralized services in contrast to distributed, partitioned, and replicated services, and (3) centralized algorithms in contrast to decentralized algorithms.

In this chapter, we use *distribution* to refer to the deployment of the software of a self-adaptive system to hardware. As such, distribution of a self-adaptive system refers to the deployment of the software of both the managed subsystem and the managing subsystem. A distributed self-adaptive system consists of multiple software components that are deployed on multiple nodes connected via some network. The opposite of a distributed self-adaptive system is software that is deployed on a single node. The managed and managing subsystems can be deployed on the same or on different nodes. For example, the software components of a managed subsystem may be deployed on a set of nodes, while the software of the managing system may be deployed on one dedicated node. Thus, while the managed system may be distributed, it is possible that the managing system is not.

With *decentralization*, we refer to how control decisions in a self-adaptive software system are coordinated among different components, independent of how those control components are physically distributed. In particular, we consider decentralization at the level of the four activities of self-adaption: monitoring, analyzing, planning, and execution. Decentralization implies a type of control in which multiple components responsible for one of the activities of self-adaption perform their functionality locally, but coordinated with with peers. Typically, such decentralized coordination is organized as follows: monitoring components coordinate with other monitoring components to collect the knowledge required for subsequent analysis; analysis components coordinate to decide whether the conditions for a particular adaptation hold; multiple planning components coordinate to plan an adaptation; and multiple execution components coordinate to execute an adaptation, e.g., they have to synchronize their adaptation actions. Decentralized control contrasts with central control. In central control, a single component exists (for one of the activities of self-adaptation) that performs its function. For example, analysis and planning is centralized in a self-adaptive system if this system has one analysis and one planning component that decides about when and how to perform an adaptation.

From this perspective, the functions of adaptation in a self-adaptive system (monitoring, analysis, planning, execution) can in principle be centralized or decentralized, independently of how the software of the managed and managing subsystems are deployed. However, in practice, when the managed software is deployed on a single node, the managing software will often also be deployed on that node and the adaptation functions are typically centralized. Similarly, fully decentralized adaptation functions typically go hand in hand with distribution of the software of the managed and managing subsystems. Between these two extremes, a variety of different ways to organize the functions of adaptation exist. The next two sections of this chapter elaborate on this.

3 A Notation for MAPE Patterns

As mentioned in the introduction, the adaptation logic (managed subsystem) typically involves feedback control loops with four key activities: Monitor (collect), Analyze (determine), Plan (prepare), and Execute (act), defining the classic MAPE control loop [28]. Given the central role control loops play in the way we conceptualize, design, and implement self-adaptive systems, [9] argues that "the design [of self-adaptive systems] must make the interactions of control loops explicit and expose how these interactions are handled".

Several authors, e.g. [46,38,9], have argued that existing approaches to describe software models are not well suited to represent control loops in the design. In [23], the authors introduce a UML profile for modeling control loops that extends UML modeling concepts. This approach allows control loops to become first-class elements of the design. The proposed UML profile supports modeling and reasoning about interactions between coarse-grained "controllers," while in this work we aim to model finer-grained interactions between the components of control loops.

In this section, we introduce a graphical notation to explicitly capture interacting MAPE loops by considering the control loop components M, A, P, and E, and their interactions. We call a recurring structure of interacting MAPE components a *MAPE pattern*.

In order to describe different MAPE patterns properly and overcome complexity, we introduce a unified, simple graphical representation based on a condensed notation of a MAPE loop as depicted in Figure 2. The key for this figure and the other figures with interacting MAPE loops in this chapter is described in Figure 3.

We distinguish between a MAPE pattern and an instance of the pattern. The former describes the abstract structure of the MAPE pattern in terms of abstract groups of MAPE components, the type of interactions between MAPE components between groups, and the interactions with the managed subsystem. The latter describes the concrete structure of the pattern for one particular configuration.

A group of MAPE components expresses a logical collection of MAPE components that may occur once or more in the pattern. The annotated cardinalities of the interactions between the groups of MAPE components determine the allowed occurrences of the different groups in the pattern. For the example pattern shown in Figure 2, there is only one occurrence of the group with four MAPE components allowed, while there are many occurrences possible for the group with only the M and E component (in the example shown at the bottom of Figure 2, there are two

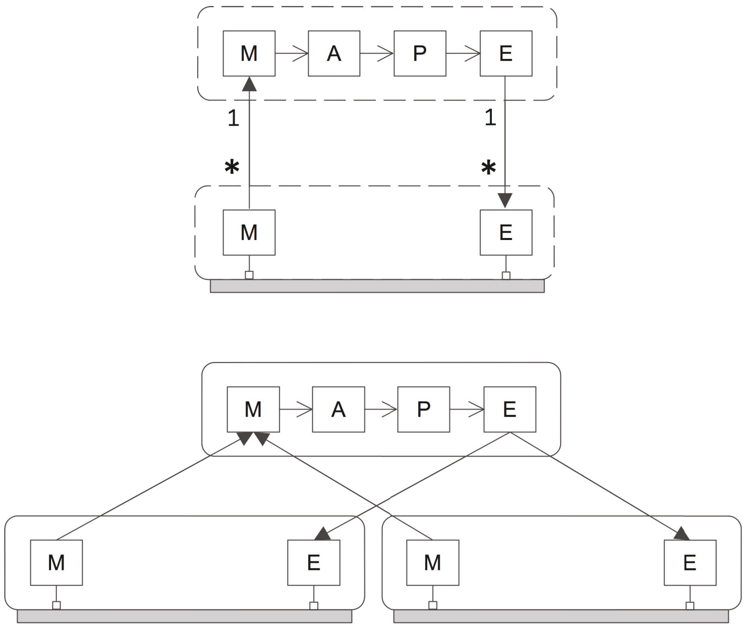

Fig. 2. Top: An example of a MAPE pattern. Bottom: an instance of the pattern in one concrete configuration.

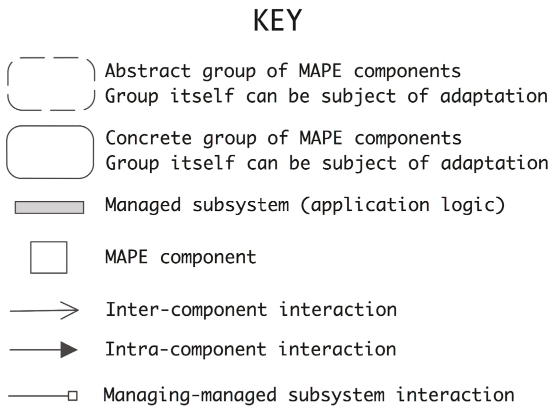
KEY
Abstract group of MAPE components
Group itself can be subject of adaptation
Concrete group of MAPE components
Group itself can be subject of adaptation
Managed subsystem (application logic)
MAPE component
Inter-component interaction
Intra-component interaction
Managing-managed subsystem interaction

Fig. 3. Key for patterns and instances

such occurrences). Notationally, groups partition a pattern so that it is easier to see which parts of the MAPE loop are being decentralized.

We differentiate the following types of interactions:

– Managed-managing subsystem interactions: these are the interactions between M components and the managed subsystem for monitoring purposes, and between E components and the managed subsystem for performing adaptations. Managed subsystem is the application logic that provides the

system's domain functionality, or it can be a group of MAPE components that itself is subject of adaptation. Note that not every M and E component has to interact with the managed system. For example, in the instance shown in Figure 2, the M component in the top group is responsible for providing the required information about the managed subsystem to allow the A component to decide about adaptations. However, the actual collection of this information is delegated to M components that may reside at the different nodes where the managed subsystem is deployed.
- Inter-component interactions: these are the interactions between different types of MAPE components. In a typical MAPE loop, M interacts with A, A with P, and P with E. However, other interactions paths may be possible, such as subloops within a MAPE loop as discussed in [53].
- Intra-component interactions: these are the interactions between MAPE components of the same type, e.g., interactions between M components. Two important subtypes of this kind of interactions are delegation (as in the example pattern of Figure 2) and coordination. Coordination is used when components of the same type, but from different MAPE loops, interact with one another. Examples are two A components that have to coordinate to decide whether an adaptation should be applied, and two E components that have to synchronize the actions of an ongoing adaptation.

There are a number of important aspects of interacting MAPE loops that we do not consider explicitly in the notation and the patterns described in this chapter. First, we abstract away the knowledge aspect of MAPE components and how this knowledge is used and shared by the MAPE components. It is clear that knowledge exchange is an important design aspect of interacting MAPE loops and may have an impact on the applicability of the pattern. Second, we do not consider the distribution of the MAPE components and the communication resulting from actual deployment based on a particular network topology and supporting communication infrastructure (message oriented, publish-subscribe, etc.). Different deployments of the MAPE components may be possible, combined with different types of communication infrastructure, each with its particular benefits and trade-offs. We refrained from including these concerns in the patterns for the following reasons: (1) the treatment of knowledge heavily depends on the characteristics of the domain (e.g., the degree of cooperation or competition in the system, the sensitivity of particular knowledge, etc.), (2) the deployment of MAPE loops depends on constraints imposed by the underlying infrastructure (e.g., type of network, use of a particular middleware, etc.), and (3) including knowledge and deployment as first-class in the patterns would significantly expand the design space for each pattern and increase the complexity, and make less clear the interactions between the MAPE components, within and across MAPE loops. We touch upon a number of aspects of knowledge storage and exchange in section 6.

4 Patterns for Decentralized Control

We now present a selection of MAPE patterns that model different types of interacting MAPE loops with different degrees of decentralization. These patterns are not intended to be a complete enumeration of all possible configurations. In fact, the presented patterns emerged from the experiences of the authors with building self-adaptive systems, and discussions at a Dagstuhl seminar [44]. We use a standard template to present the patterns, consisting of the following parts: problem/motivation, solution, consequences, and examples.

We start by presenting two patterns, *coordinated control* and *information sharing*, both based on a fully decentralized approach that represents the antithesis with respect to a single centralized control loop. Both these patterns are based on a "flat" distribution model, where a multiplicity of peer MAPE loops operates in parallel to manage the overall system self-adaptation. Then, we present three other patterns, *master/slave*, *regional planning*, and *hierarchical control*, that are based instead on a "hierarchical" distribution model, where higher level MAPE components control subordinate MAPE components. The hierarchy generally reflects a separation of concerns among different control loops. These three patterns can be considered as intermediate points between fully decentralized and centralized control, as the root of the hierarchy basically constitutes a centralization point.

4.1 Coordinated Control Pattern

Problem/Motivation. In many cases, centralizing control for self-adaptation is simply not feasible. Among the possible reasons for this include: a) an inherent distribution of information in the system makes it too costly or even infeasible to collect all the data required for adaptation; b) due to the scale of the system the cost to process all the information at one place may be too high; and c) the system spans multiple ownership domains with no trustworthy authority to control adaptations. However, support for adaptation to achieve certain quality attributes is still desired. For example, multiple data centers still require guarantees that service-level agreements and legal regulations can be met, or control systems for managing traffic in a metropolitan area must coordinate to grant passage to emergency vehicles. In such cases, there may be no obvious way to organize control so that one part of the system has authority over another. In such a case, each control loop must coordinate with its peers to reach some joint decision about how to adapt.

Solution. A possible solution to overcome these problems is to decentralize the four MAPE activities. A MAPE loop is associated with each part of the managed system that is under its direct control. What characterizes this pattern is that basically all the M, A, P and E components of each loop coordinate their operation with corresponding peers of other loops. For example, A components exchange information to make a decision about the need for an adaptation, E components exchange messages to synchronize adaptation actions, etc. The

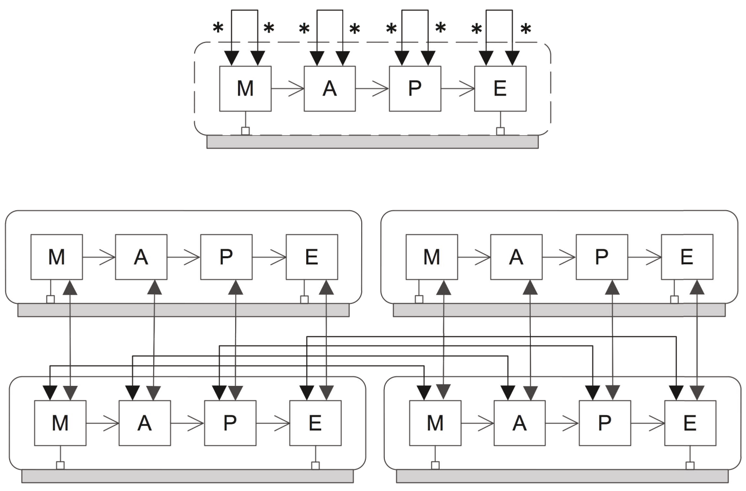

Fig. 4. Top: coordinated control pattern. Bottom: a possible instance.

interactions are typically localized, so that each component directly interacts with only a subset of its peers.

Figure 4 shows the decentralized control pattern and illustrates it for a concrete configuration. The pattern consists of one abstract group of MAPE components that contains all four components. The abstract group can be instantiated an arbitrary number of times. The pattern uses the standard sequence of interactions between the components within a MAPE loop, but variations may be possible. MAPE components of different MAPE loops can interact with peers to share particular information or coordinate their actions. The cardinalities of the intra-component interactions define the connectivity among MAPE components of the same type, that is, each component can interact with an arbitrary number of components of its own type.

The instance diagram at the bottom of Figure 4 shows a concrete instantiation of the pattern with four groups of MAPE components. Even if the pattern allows for a full connection among all peers, in a typical scenario, interactions among the same type of MAPE components will be localized.

Consequences. Decentralized control has the potential of good scalability with respect to communication and computation, depending on the coupling degree among peer components, and the number of other peers each MAPE component has to explicitly interact with. For systems in which adaptations can be performed based on local interactions between MAPE components, the communication overhead is limited to interactions with local peers. Furthermore, the computational burden is spread over the nodes. Decentralization may also contribute to improving robustness as there is no single point of failure. Decentralization of control may be the only option in cases where no single entity has the knowledge or authority to coordinate adaptations across a set of managed subsystems. There are a number of potential downsides of decentralized control as well. When coordination is required between MAPE components of many nodes, scalability may be compromised. The cost for reaching consensus about suitable adaptation actions may be high (in terms of communication and/or timing). Decentralized control may cause problems with ensuring consistency of adaptations. Furthermore, it may lead to sub-optimal adaptation decisions and actions, from the overall system viewpoint.

Examples. An example application that can be characterized as an implementation of the coordinated control pattern is presented in [18]. A component manager located at each node of a distributed application implements a logical control loop. The set of component managers cooperate to preserve some architectural constraints under certain events. All component managers rely on a group membership service and reliable broadcast to achieve a consistent view of the knowledge accumulated by their local M and A activities. Moreover, adaptation actions planned and executed by local P and E activities are globally coordinated by means of a totally ordered broadcast that implements a distributed locking scheme. As pointed out by the authors, the adopted mechanism to achieve global coordination requires explicit interaction among all MAPE loops. The resulting overhead thus limits the scalability of the proposed control architecture.

4.2 Information Sharing Pattern

Problem/Motivation. A software system consisting of a (potentially large) set of loosely connected components requires support for adaptation to maintain a particular concern or quality attribute. The components of the system are deployed on different nodes. Each part of the system can adapt locally, but requires information about the state of other nodes in the system because a local adaptation may impact these other nodes (e.g., on some quality attribute of those operations). However, apart from information sharing, nodes do not need to coordinate other adaptation activities. For example, in a sensor network for environmental monitoring (e.g., habitat monitoring), certain nodes may be equipped with sensors to detect a fire. In case a node detects a fire, it produces an alarm signal that can be spread effectively through the network using a smart gossip algorithm. Upon receiving the signal, nodes can activate a local adaptation procedure to anticipate disaster.

Solution. In contrast to the coordinated control pattern, the information sharing pattern restricts the inter-component type interactions of decentralized control to monitor (M) components only, as depicted in Figure 5. In particular, in this pattern only M components communicate with one another, while the A, P, and E components of each loop operate independently of their peers. The interactions are typically localized, that is, M components exchange information only with nodes in their (physical or logical) context. Thus, some information collected about the status of the managed systems is shared among the MAPE loops that allows local analysis, planning, and execution of adaptations

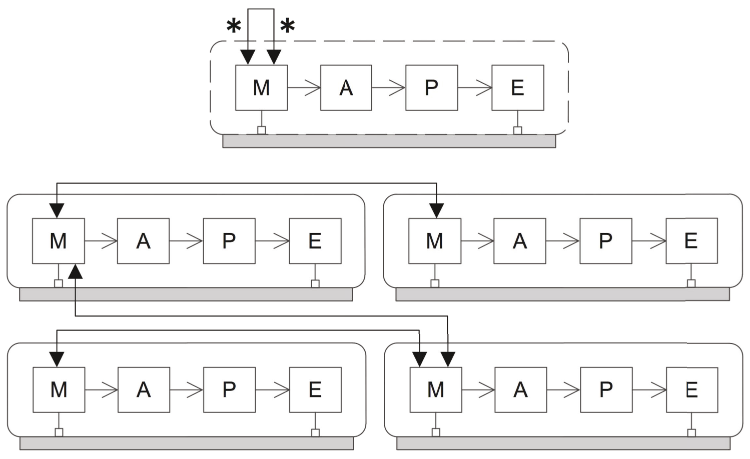

Fig. 5. Top: information sharing pattern. Bottom: concrete instance of the pattern.

without further coordination. Information sharing about the system state may be realized by explicit interactions among peer M components, or by implicit interactions where each M component independently monitors state information that is affected by the behavior of other nodes.

Figure 5 shows the information sharing pattern and illustrates it for a concrete configuration. The pattern consists of one abstract group of MAPE components containing all four components that can be instantiated an arbitrary number of times. At the inter-loop level, only M components can interact with an arbitrary number of peers to share particular information.

The instance diagram at the bottom of Figure 5 shows a concrete instantiation of the pattern with four groups of MAPE components. In this particular example, two M components interact with two peers (top left and bottom right), while the two other M components only interact with one peer.

Consequences. From a scalability perspective, information sharing may produce potentially higher benefits than coordinated control. Indeed, the less stringent interaction requirements (limited to M components only) may result in solutions that scale even better with respect to communication. However, this requires that the traffic between M components, in particular in case of explicit interactions, is limited in scope and volume. Another potential benefit is that since P, A, and E components can act locally without the need for coordination, this may lead to more timely decisions and execution of adaptations. On the other hand, the reduced coordination may increase locally optimal objectives, but at the cost of globally optimal ones. In the worst case, local decisions may conflict with one another, resulting in perpetual adaptation of the system, thus wasting resources and having adverse effect on the system's availability and stability.

Information sharing can be considered a special case of the coordinated control pattern discussed above, where the interactions among peer A, P and E components have been completely dropped. However, while the coordinated control pattern aims at directly achieving some regional or global objective through explicit cooperation among all types of MAPE components, the information sharing pattern adopts a different perspective, where achieving the global objective is less direct, because decisions are made locally rather than in a coordinated fashion. For this reason we prefer to give a "first-class citizen" status to this pattern in our list of alternative patterns.

Examples. The self-healing traffic monitoring system presented in [56] is an example in which the information sharing pattern is used to support self-healing in a traffic monitoring system by means of explicit state information sharing. The overall system consists of a set of cameras distributed along roads that are used to detect and report traffic jams (for example to a traffic light control system). A local traffic monitoring system deployed on each camera (i.e., the managed subsystem) monitors the traffic conditions in its viewing range. When a traffic jam is detected the local traffic monitoring systems form a dynamic organization with neighboring local traffic monitoring systems that span the range of the traffic jam. One of the monitoring systems is responsible for reporting the traffic jam to interested clients. To make the system resilient to camera failures, a self-healing subsystem is added to each local traffic monitoring system. To this end, the self-healing subsystems exchange information with self-healing subsystems on local cameras about their status using a ping-echo protocol. When a failure is detected (one of the self-healing subsystems does not respond with an echo message), the self-healing subsystem locally performs some analysis and planning activities that trigger local adaptation actions. Examples are removing the reference to a failed camera from the set of neighbors, and changing the dynamic organization of a set of monitoring cameras.

Another example of information sharing is described in [48] that aims to tackle the scalability problem of the group membership service and reliable broadcast used to preserve architectural constraints among distributed nodes as described in [18]. In this work, a gossip protocol is used to exchange information between nodes to support local adaptations. The authors show that the approach achieves a fault-tolerant and scalable solution to exchange information regarding the component configuration.

The two patterns identified and described so far are characterized by the introduction of different degrees of decentralization, but are both driven by a *flat separation of concerns* model, which places the different MAPE loops at the same conceptual/abstraction level. In other words, they play analogous roles on the different parts of the overall managed system they are directly responsible for. In the following we describe three hierarchical control patterns, where MAPE loops at different levels play different roles, with different responsibility levels.

4.3 Master/Slave Pattern

Problem/Motivation. There is a need to adapt a distributed software system for some concern. Monitoring and adaptations of the software needs to be done locally at each node, for example because of the high cost of transferring monitored data or because of the specificity of local adaptations. On the other hand, there is a need to provide global guarantees, predictability, and consistency about the state of the distributed system and its adaptations. For example, a central controller in an automated logistic systems (with cranes, conveyor belts, etc.) may rely on locally collected knowledge of machine software or their environment (which may include complex processing performed by M components) to trigger some of the machines to change their work mode (which may involve complex manipulations of the machine software performed by E components).

Solution. This pattern organizes the adaptation logic by creating a hierarchical relationship between one (centralized) master component that is responsible for the analysis and planning of adaptations (A and P activities) and multiple slave components that are responsible for monitoring and execution (M and E activities), see Figure 6. The pattern consists of two abstract groups of MAPE components. There is a single instance of the group with a P and an A component, and there can be an arbitrary number of instances of the group with an M and an E component. Each M interacts with the A component and P interacts with each E component. As such the pattern supports the typical flow of inter-component interactions of a MAPE loop, but with multiple instances of M and E components.

The M components of the slaves monitor the status of the local managed subsystems and possibly their execution environment and send the relevant information to the A component of the master. A, in turn, examines the collected information and coordinates with the P component, when a problem arises that requires an adaptation of local managed systems. The P component then puts together a plan to resolve the problem and coordinates with the E components on the slaves to execute the actions to the local managed subsystems.

The instance diagram at the bottom of Figure 6 shows a concrete instantiation of the pattern with three slaves.

Consequences. The master/slave pattern is a suitable solution for application scenarios in which slave control components need to process monitored information to derive the required data allowing centralized decision making, and execute local adaptation (probably based on higher-level adaptation instructions). On the positive side, centralizing the A and P components facilitates the implementation of efficient algorithms for analysis and planning aimed at achieving global objectives and guarantees. However, sending the collected information to the master component and distributing the adaptation actions may impose a significant communication overhead. Moreover, the solution may be problematic in case of large-scale distributed systems where the master may become a bottleneck. Finally, the master component continues to represent a single point of failure.

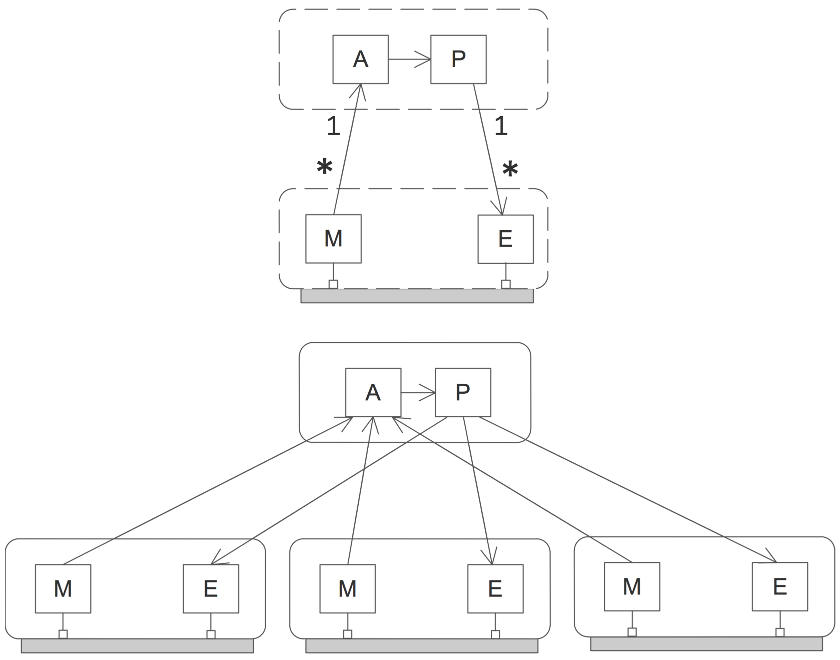

Fig. 6. Top: master/slave pattern. Bottom: concrete instance of the pattern.

Examples. The control architecture proposed in the RESERVOIR project [50,16] is an example of the master/slave pattern. The architecture is proposed in the context of a virtualized data center, where virtual execution environments are offered on top of a set of distributed physical servers. To meet the SLAs negotiated with the data center users, the control system monitors the system status (e.g., utilization degree of physical resources) through a set of monitors located at the different servers. A central master controller collects and analyzes these data, and plans suitable adaptation actions (that include, for example, changing the balance of the time slice among different virtual environments hosted by the same server, or live-migrating processes among physically separated servers).

The master/slave pattern is also at the basis of the control architecture for the *Znn.com* example system [10], developed according to the Rainbow framework [17]. The Znn.com system implements a news service that provides multimedia news content to its users, and is architected as a dynamically variable number of servers that serve clients requests by accessing a backend database. According to the Rainbow framework, the control system of Znn.com uses a distributed set of *probes* and *gauges* to monitor the system status. Collected data is centrally analyzed by an *architecture evaluator* that detects possible problems, while an *adaptation manager* decides on the best adaptation whose goal, in this example, is to keep the response time within a given threshold. The execution of adaptation actions (that include changing the number of active servers, and varying the "fidelity" of the provide responses) is then delegated to a set of distributed *effectors* driven by a *strategy executor*.

4.4 Regional Planning Pattern

Problem/Motivation. Different loosely coupled parts of software (regions) of a complex integrated software system want to realize local adaptations (within a region) as well as adaptations that cross the boundaries of the different parts (between regions). A typical scenario is a federated cloud infrastructure where adaptations within regions may aim to optimize resource allocation, while the objective of adaptations between regions may be delegation of certain loads under particular conditions (that owners of regions may not want to expose). Another setting is a supply chain management system where partners in the chain have certain local adaptation objectives, while adaptations between partners or system-wide adaptations may aim to achieve some global utility objective.

Solution. Regional planning provides one P component (a regional planner) for each region. A regional planner collects the necessary information from the underlying subsystems under its supervision to plan adaptations. Regional planners interact with one another to coordinate adaptions that span multiple regions.

Figure 7 shows the regional planner pattern and illustrates it for a concrete configuration. The pattern consists of two abstract groups of MAPE components, which both can occur an arbitrary number of times. The first group contains M, A, and E components. The second group contains only a P component. Inter-component type interactions follow the logical flow of a MAPE loop. Intra-component type interactions are restricted to P components.

The instance diagram at the bottom of Figure 7 shows a concrete instantiation of the pattern with two regional planners. For each region, the M components monitor the status of local managed subsystems and possibly the execution environment, the local A components analyze the collected information, and report the analysis results to the associated regional planner. The regional planner may then decide to perform a local adaptation (i.e., within the region), or regional planners may interact with one another to plan adaptations that span the two regions. Once the planners agree on a plan they can put the adaptations to action by activating the E components of the respective component groups involved in the adaptation.

Consequences. Regional planner enables a *layered separation of concerns* among different MAPE loops within a single ownership domain, where several MAPE loops delegate the planning function to a higher level component. For systems that cross the boundaries of ownership domains, regional planner enables a further (flat) separation of concerns for the planning function, where each planner is responsible for the planning of adaptations in its region. Local analysis of monitored data may reduce the amount of data and frequency of interactions with the planner. A downside of regional planner may be a lack of efficient adaptations. Aggregating the results of local analysis and coordinating the planning of adaptations may incur considerable overhead. Moreover, the pattern may require very detailed planning of the execution of adaptations as it does not support runtime coordination between E components.

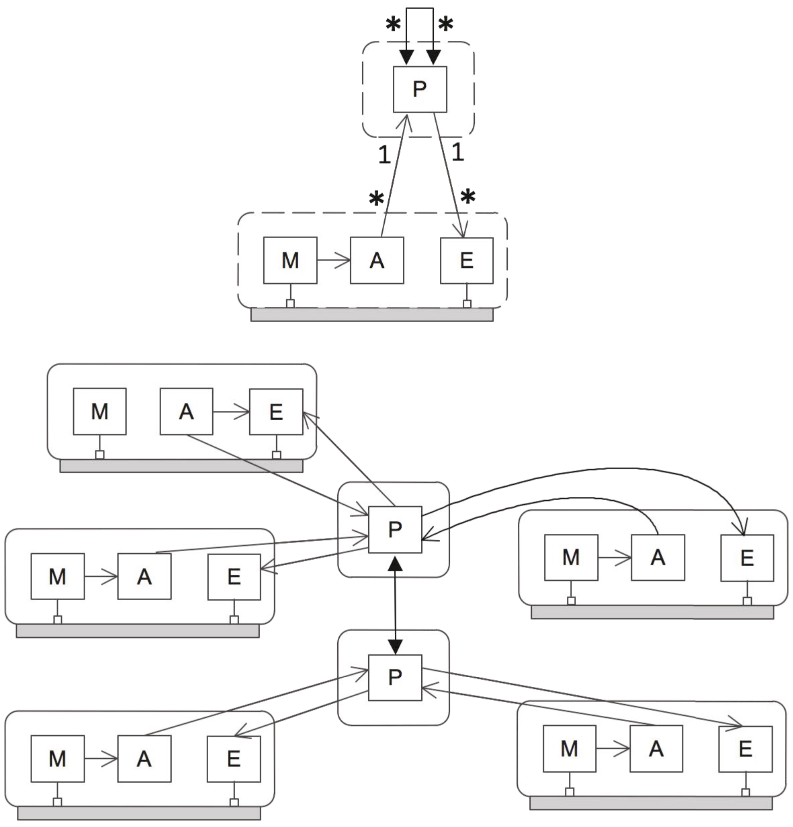

Fig. 7. Top: regional planner pattern. Bottom: concrete instance of the pattern with two regions.

Examples. The MOSES framework [8] is an example that instantiates the regional planner pattern. Within the context of service-oriented systems, the goal of MOSES is to provide a brokering service that supports runtime adaptation of composite services offered to multiple users with different service levels. The MOSES framework consists of a set of distributed monitoring components (*WS Monitor* and *QoS Monitor* components) that collect data about the availability and quality of service of different pools of candidate services that can be used to build the composite service managed by MOSES. Collected data are locally analyzed. The result of the analysis can trigger the calculation of a new plan by a centralized planning component (*Optimization engine*) that calculates a new abstract-to-concrete services binding policy, that is then realized at the endpoints.

The Deployment Improvement Framework [35] provides the ability to determine the optimal deployment of a software system at runtime and effecting it through runtime redeployment and adaptation of its components. This framework has been realized using a regional planner pattern, supporting redeployment in mobile and pervasive computing environments. In this particular case, each host has a *decentralized planner* (i.e., a regional planner) that only manages a single instance of a group of M, A, E components. Furthermore, each host has a *local monitor*, *local analyzer*, and *local effector* that are responsible for the monitoring, assessing changes in the monitored parameters, and redeployment of the components on the host they reside. Each host has a *model* that contains some subset of the system's overall model, populated by the data received from the local monitor and the model of the hosts to which this host is connected. The local analyzer on each host determines when the conditions for an improved deployment architecture occur, based on the local model. The decentralized planner then synchronizes with its remote counterparts to find a common solution. If the planners agree, the improved deployment architecture is effected by the local effectors.

4.5 Hierarchical Control Pattern

Problem/Motivation. The control architecture for a complex distributed system may itself become a complex system that needs to be adapted. In this case it is often necessary to consider multiple control loops within the same application. The loops can work at different time scales and manage different kind of resources, and resources with different localities. However, in this context, control loops need to interact and coordinate actions to avoid conflicts and provide certain guarantees about adaptations. The problem is then how to separate concerns to manage this complexity? Examples of such systems are: a) within a single data center, higher level control loops are responsible for achieving power consumption or workload goals, whereas local control loops manage workflow distribution between localized subsets of the nodes, and b) adaptation in pervasive computing environments could be organized into controllers that manage adaptation of human tasks as a user's goals change (in the order of minutes) and controllers that manage particular instances of these tasks to provide fault tolerance (in the order of seconds).

Solution. The hierarchical control pattern provides a *layered separation of concerns* to manage the complexity of self-adaptation. This pattern structures the adaptation logic as a hierarchy of MAPE loops. Different layers typically focus on different concerns at different levels of abstraction, and may operate at different time scales. Loops at lower layers operate at a short time scale, guaranteeing timely adaptation concerning the part of the system under their direct control. Higher levels operate at a longer time scale with a more global/strategic vision. MAPE loops at the bottom layer are directly concerned with different parts of the managed subsystem. MAPE loops at intermediate layers are concerned with the adaptation layers beneath. Finally, the MAPE loop at the top is concerned with the overall adaptation objectives of the system.

Figure 8 shows the hierarchical control pattern and illustrates it for a concrete configuration. The pattern is shown for a hierarchy of three layers, but more intermediate layers are possible. The M and E components of abstract groups at the bottom layer directly interact with the managed subsystem. M and E

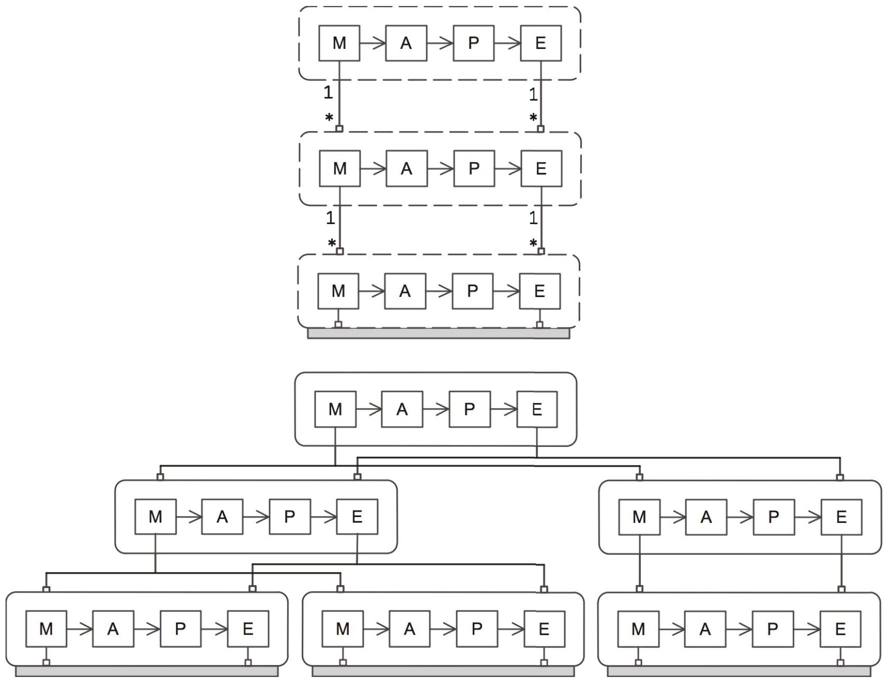

Fig. 8. Top: hierarchical control pattern. Bottom: concrete instance of the pattern.

components of abstract groups of higher-level layers interact with groups at the layers beneath.

The instance diagram at the bottom shows a concrete instantiation of the hierarchical control pattern. In this particular example, the hierarchy consists of three layers with two intermediate MAPE loops, one of them managing two subordinate loops, the other one managing a single loop.

Consequences. The hierarchical control pattern enables adaptation logic to be structured so that the complexity of self-adaptation can be managed. The hierarchical structure allows bottom layer control loops to focus on concrete adaptation objectives while higher level control loops can take increasingly broader perspectives. This corresponds to the layered organization of self-adaptation as proposed in [31]. However, there are a number of potential trade-offs with hierarchical control. The hierarchical decomposition of the adaptation concerns and the allocation of these concerns to different control loops might be difficult to achieve, in particular when goals interfere with one another. Moreover, it is known from behavior-based architectures [2] that the design and management of hierarchies with multiple layers can become very complex. As a result, there might be no guarantee that the overall solution meets the specifications.

Examples. A classic example of hierarchical control of adaption is the IBM architectural blueprint [25]. This approach consists of autonomic managers that add self-* properties to resources and these managers are, in turn, managed by other autonomic managers. At the highest level a manager takes high-level policies from users and delegates these throughout the hierarchy of autonomic managers. [4] discusses how the Autonomic Computing Reference Architecture (ACRA) can be used to orchestrate a set of autonomic managers that share knowledge sources to realize adaptations of managed resources.

The use of multiple control loops is proposed also in [33], where Litoiu et al. propose a hierarchical framework to deal with autonomic systems where it is possible to consider different time scales and different kind of managed resources.

Another example of the application of the hierarchical pattern can be found in [27]. In this work, the authors present Mistral, a multi-level hierarchical self-adaptive system. Specifically, Mistral is presented for a large data center environment and deployed in the form of a hierarchical control scheme with multiple instances of Mistral controllers managing different subsets of hosts and applications and operating at different time-scales. The controllers at the lower level manage a small number of machines and the applications hosted on them, while at the next higher level, a controller manages machines owned by multiple lower level controllers. Mistral reconfigures the system when variations in the monitored workload are detected and the adaptation actions are selected according to a predefined utility function.

5 Drivers for Selecting Control Schemas for Adaptation

So far we have outlined a set of patterns for decentralizing self-adaptive control loops, discussed forces that express conflicts among concerns when applying the patterns, and described how the patterns have been used by existing self-adaptive systems. Based on these insights, we discuss some of the drivers that should be considered by designers of self adaptive systems when choosing a MAPE pattern. As with any design, it is not possible to fulfill all requirements of all stakeholders with any one pattern. This means that choosing a pattern will depend on the relative importance of the requirements that stakeholders place on the managed system.

In the literature, it is usually quality concerns such as fault handling, efficiency, resource consumption, and load balancing that are the main goal of, and thus the main drivers for, self-adaptive solutions [24,52,55]. Due to the variability of domains and requirements, we cannot give an exhaustive list of how requirements may influence the choice of control mechanisms. Rather, we discuss in a few examples how certain kinds of requirements may impact this choice.

Optimization of one or more system properties is easier in centralized approaches where all measurement data is collected in one place, and only one entity makes control decisions based on that data. In decentralized approaches where several entities make local control decisions likely it will be more difficult to find a global optimum for system reconfiguration, since it is possible for control decisions to adversely influence each other, leading to frequent antagonistic adaptations. Ensuring other global properties is also easier to achieve with a centralized controller, when all data relevant to decision making is directly accessible. However, ensuring that this data is consistent in a distributed system poses significant challenges in itself [15,19].

The *scalability* of systems with respect to communication can be impacted significantly by the choice of a centralized or decentralized solution for self-adaptation. The larger and more complex an adaptive system becomes, the more data has to be processed to make control decisions. This data may also have to be transmitted from the node in the network where it is gathered to the node that hosts the decision logic. Scalability is thus impacted by at least two factors: the amount of data that has to be processed to make control decisions, and the amount of data that has to be transmitted across networks. In both cases, the more data there is, the less scalable the system will be. Decentralized systems can improve scalability if decisions can be made locally, based on data collected from the local context, or possibly subsets of the global monitoring data. Thus decentralization of self-adaptation functions may reduce the amount of data that has to be transmitted, and the amount of data that has to be processed to make decisions about adaptations. Effectively, this parallelizes adaptation decisions at the cost of making it hard to ensure global optima.

Robustness against node and link failure is the classical domain of distributed, replicated systems. A system with centralized control has a central control node as a bottleneck and potential single point of failure. Decentralized systems on the other hand will still be able to function even when some nodes and links fail. Only the nodes affected directly by link failures or controlled by a crashed controller will be affected in this scenario.

Responsiveness to changes needs to be considered. Different MAPE patterns have different reaction characteristics and MAPE loops in particular patterns may work at different time scales. For example, Rainbow (master/slave) can act on a system within seconds, but in some cases reactions in less than a second may be needed. To soundly organize MAPE loops hierarchically means that a loop must act at a time scale greater than its subordinate loops. Decentralizing control may make an adaptive system more responsive, but at the cost of producing subobtimal adaptations.

Different *administrative domains* may force particular types of adaptation control on a designer. For example, building a centralized model of the entire system may be infeasible if the knowledge of parts of the system has to be kept hidden (e.g., for strategic reasons). Similarly, an adaptive system may not be able to exercise control over some parts of the system. Consider a globally distributed data center network, where each data center may control itself, but cannot request reconfiguration in sibling centers because they are owned by different companies, or are under regulations of different governments. In such a case, the particular patterns that can be used for decentralization of control will be affected by the amount of information that is shared between the domains, and the amount of control that one domain can influence on another.

Domain constraints may also impose restrictions on the choice of a MAPE pattern for controlling adaptation. For example, in certain domains (e.g. banking), security or confidentiality requirements might prevent the sharing of data needed for control decisions with a central entity. In such a case, it might be feasible for subsystems to summarize and filter data so that no confidential data leaks, and then pass that data on to a centralized controller. Alternatively, a regional planning solution where each part of the overall system only deals with its own confidential data is conceivable. In some domains, for example mobile network applications, network interruptions and topology changes are so frequent that centralized solutions would be infeasible. In both the above scenarios it is infeasible or at least impractical to collect all relevant data at a central node and thus in these scenarios a decentralized solutions are more likely to be effective.

6 Discussion

The focus of the patterns described in this chapter is on the structures of MAPE loop components and their interactions. We have abstracted away the representation of knowledge in the patterns, how this knowledge is used and shared among the MAPE components, and how the system components are actually deployed on hardware. However, the ways in which knowledge is stored in the system and exchanged among MAPE components and the actual deployment of the system are important design concerns that will affect the applicability of the patterns. As explained in Section 3, we have refrained from including these concerns in the patterns since the way knowledge is treated and components are deployed heavily depend on the characteristics of the domain. Considering these concerns explicitly would increase the complexity of the descriptions of the patterns significantly. Instead, we consider the way knowledge is stored and exchanged between MAPE components and the distribution of the various components as two different *views* in the design of a self-adaptive system, complementary to the structured, interaction-oriented view of the patterns presented in this chapter.

In this section, we touch upon some aspects of knowledge in the design of MAPE loops of self-adaptive systems. It is our aim to give some initial ideas about such design decisions and their implications. Clearly, extensive research is required to treat the aspects of knowledge and deployment in a systematic manner. Concretely, we will look at two alternative approaches to deal with knowledge in the hierarchical control pattern.

As we explained in Section 4, one particular objective of the hierarchical control pattern is to manage complexity of self-adaptation by separating concerns of the adaptation logic in the form of a hierarchy of MAPE loops. Figure 8 shows the interactions among MAPE loops in consecutive layers. Here, we show two possible approaches to share knowledge among MAPE components in this pattern.

Figure 9 shows an instance of the hierarchical control pattern with individual knowledge repositories for each MAPE loop. In this configuration, knowledge can only be exchanged via the interactions of MAPE loops of consecutive layers. Figure 10 shows an alternative configuration with additional knowledge repositories that are shared among MAPE loops within layers.

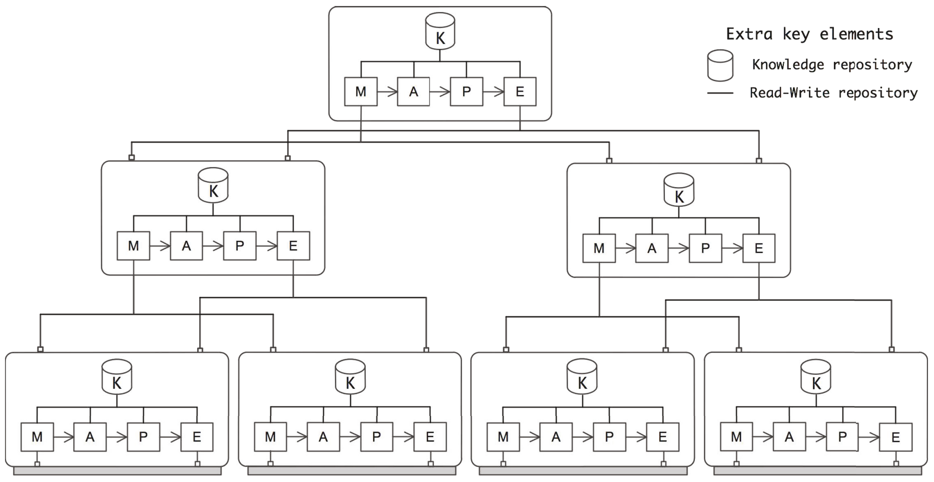

Fig. 9. Instance of the hierarchical control pattern with knowledge repositories per MAPE loop

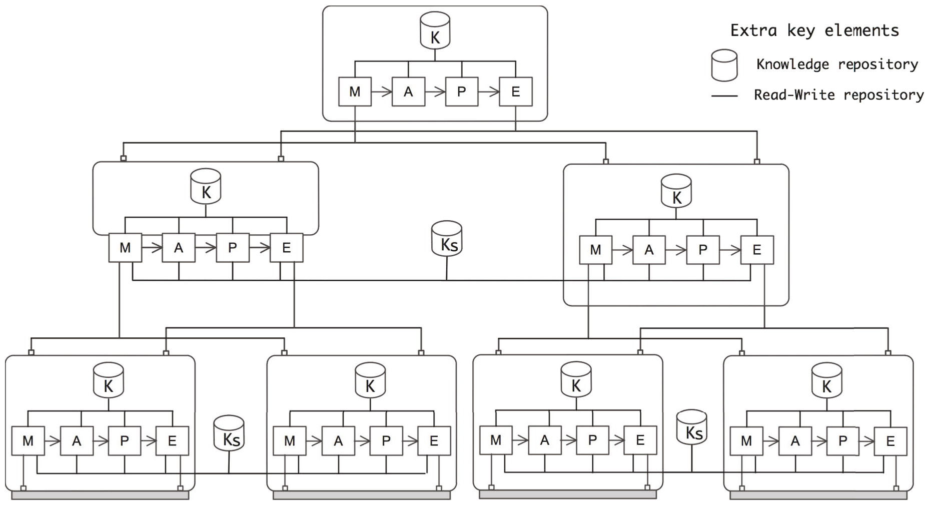

Fig. 10. Instance of the hierarchical control pattern with additional shared knowledge repositories within layers

In the first approach, each MAPE loop maintains knowledge in a local repository. This approach restricts the exchange of knowledge between MAPE loops of consecutive layers. Such knowledge exchange is important, for example, to enable higher level MAPE loops to make decisions about adaptations at lower levels. In the second approach, MAPE loops can also exchange knowledge with siblings using a shared knowledge repository. This approach enables MAPE loops at one layer to coordinate adaptations without direct interference of MAPE loops at the layer above. Shared knowledge in the form of a shared tuple space, for example, creates a loose coupling between MAPE loops at one layer. Such an organization may be a solution to situations where adaptations have to be realized between managed subsystems that are connected in a very dynamic manner, e.g., in a mobile setting.

These two example scenarios illustrate that the aspect of knowledge can be treated in (potentially many) different ways for this particular pattern, resulting in specific variants of the pattern that are useful for different domains with different characteristics and specific requirements. Study of these variants for different types of MAPE patterns is an interesting area of future research.

7 Related Work

The work on software architecture and design patterns is extensive. The series on Pattern-Oriented Software Architecture (POSA) by Buschmann et al. [7,45,29,6], covers fundamental patterns [7], like *Reflection* and patterns specific for a domain, e.g., resource management [29], concurrency [45], and distribution [6]. These patterns provide concrete strategies and mechanisms to address specific architectural or implementation problems. The patterns proposed and described in this chapter are different in that they are considering the structure and interaction of MAPE loops and their components at an abstract level. On a more concrete level, various POSA patterns are premier candidates to realize such patterns.

Research in self-organizing systems have brought forward a number of patterns for distributed decentralized computing, for instance to support replication, which are inspired from biology [3]. Compared to the architecture-centric perspective presented in this chapter, the biology inspired patterns are described from an algorithmic perspective with more precise behavioral semantics. Such patterns are more related to the aspect of knowledge (see section 6), and are candidates to realize particular types knowledge exchange in some of the MAPE patterns.

There is a large body of work in designing and implementing self-adaptive systems, and subsequent recent reflection by researchers to develop advice and patterns for them. Gomaa and Hussein [21] have developed several software reconfiguration patterns for dynamic evolution of software architectures. They define a *reconfiguration pattern* to be a set of recurring sequences of adaptation steps (e.g., stopping/starting, (un)linking, adding/removing) necessary for ensuring *consistent* adaptation of a software system. To ensure the dynamic replacement of a component does not jeopardize the systems consistency, a reconfiguration pattern first places the component in the *quiescent* state [30], before replacing it at runtime. Subsequent to this work, several approaches have shown the utility of reconfiguration patterns to achieve consistency during adaptation. In [22], Gomaa et al. employ reconfiguration patterns in the context of self-managed service-oriented software systems, while in [14], Esfahani et al. show their utility in the design of architecture-based middleware solutions. The patterns described in this chapter are different, as we have aimed to distill patterns that result from the different compositions of MAPE components in the managing system, while reconfiguration patterns deal with ensuring the consistency of the managed system during adaptation.

Ramirez and Cheng [42] describe a set of design patterns for building dynamic software systems. Their patterns are at the level of software design, and aim to facilitate the construction of a self-adaptive software system. The purpose of patterns proposed in their work is to help engineers to better understand alternative means of achieving runtime adaptation in the system's design. The patterns proposed in this chapter are at a higher level of granularity, as we adopt an architecture-centric perspective with the aim of better understanding the impact of decentralization on self-adaptive software systems.

Some of the co-authors have defined a formal reference model (FORMS) that can be used to understand and reason about self-adaptive systems, formally defining the relationships among the environment, managing system, and managed system [57]. This model provides three perspectives of self-adaptive systems, among one is a distribution perspective that offers an abstract representation of interacting MAPE loops in terms of coordination mechanisms. Work in this chapter concentrates on the relationships between MAPE components (which refines the discussion in FORMS), and does not really consider in detail the relationship between these elements and the managed system and environment.

Explicit representation of control loops in self-adaptive systems has been discussed in [23]. A UML profile for modeling control loops is presented which allows the modeling of sensors, actuators, controller, and their interactions as parts of the adaptation logic. They are able to model a variety of instances of self-organizing systems with mutliple control loops. The work in this chapter translates the abstract concept of controller in [23] into concrete patterns of interacting MAPE loops, and provides a platform for discussing the trade-offs of applying different patterns.

8 Conclusions and Challenges Ahead

In this chapter, we have laid the groundwork for consolidating knowledge on decentralized control in self-adaptive systems in the form of patterns of interacting MAPE loops. We derived these patterns from their use in practice, introduced a notation for describing them, and discussed their ramifications with respect to certain quality attributes. This work can be used as a basis for understanding different patterns of decentralized control by software engineers of self-adaptive systems, and for comparing work in the field.

As this chapter represents only the start of the work on decentralization of control in self-adaptive systems, we conclude this chapter with a number of research challenges ahead, to contribute to the research road-map in the field. We start with more concrete challenges and move towards long term visions at the end.

Include state/knowledge. Currently, the patterns cover only structural aspects of decentralization of control in self-adaptive systems. As an important future challenge, data/knowledge aspects should also be covered in the patterns, including the differentiation between global and local knowledge. In particular, different forms of partitioning and/or (full/partial/lazy) replication of knowledge should be seamlessly included in the MAPE patterns, as they provide another path of indirect interaction between the MAPE elements. For example, one could let the components in a hierarchical MAPE loop interact by shared knowledge as described in section 6 or by introducing a new hybrid control pattern by using the coordinated control pattern in the middle layer of the hierarchical control pattern for achieving a similar interaction. One particularly interesting approach to including knowledge in the patterns for decentralized control is by defining a complementary view of the managed system that focuses on the knowledge concern.

Adding behavior and communication. The patterns presented in this chapter focus on centralization vs. decentralization of the primary functions of self-adaptation with MAPE loops. Future research should focus on identifying and classifying (i) the behavior of each MAPE component (for example, filtering or preprocessing monitored information to minimize data exchange; using decentralized or self-organized planning algorithms), (ii) the communication paradigms used for the various interactions in the patterns (for example, direct message exchange; use of a blackboard for coordination), and (iii) the specific protocols used for communication between the MAPE components (for example, push-pull, request-reply, negotiation). The pattern notation introduced in this chapter could be improved by adding different connector types between the elements to take care of items (ii) and (iii).

MAPE activities beyond sequence. In this chapter, we assume the activities in the MAPE loop follow in sequence (i.e., Monitoring followed by Analysis, Planning, and finally Execution). It is conceivable that there may be interactions that do not follow this logical sequence. For example, analysis and planning may coordinate, or analysis might coordinate with monitoring to insert new monitors or request information more or less frequently. [53] is an example in which coordination between MAPE components organized in sub-loops within a MAPE-loop is studied. Nested loops [20] are another approach where the managed system of the outer loop comprises the managing inner loop plus the system managed by the inner loop. A systematic study of MAPE activities beyond a traditional sequence is an interesting area that should be studied further. A related challenge is to study how the style of the managed system might have implications on the architecture of the managing system.

Extending the architectural expressiveness of our patterns. The notation used in this paper could be extended with a formal foundation. This would enhance the expressiveness of the patterns and allow precise expression and reasoning about different configurations of the patterns. Additionally, a formal model would enable analysis of certain properties of systems modeled with the patterns. Such analysis is particularly important for decentralized self-adaptive systems in which global properties are often a critical aspect of the design. One effort in this direction is FORMS [57] that provides formally defined modeling elements (in the Z language) to specify architectures of managing subsystems, allowing to reason about the architectural characteristics of distributed self-adaptive software systems. However, this approach does not support fine-grained specifications of interacting MAPE loops.

Dealing with uncertainty. To perform proper adaptations, the managing subsystem needs runtime models, including models of (the relevant parts) of the managed subsystem and the environment in which the self-adaptive system is deployed. Such models may introduce uncertainty, for example caused by non-determinism in the environment, inconsistencies between the managed subsystem and its runtime representation, etc. Tackling the problems related to uncertainty is challenging [13], as the causes of uncertainty are often not under control of the designer. The situation is exacerbated in decentralized self-adaptive systems, where there is no central authority, and adaptation decisions have to taken based on partial knowledge. Dealing with uncertainty in self-adaptive systems that have multiple control loops is a challenging area for future research.

Standardization. So far, the research community has focused on standardizing the notification interfaces of sensors and effectors of managed subsystems, but ignored communication *within* the MAPE loop. For instance, the Oasis standard [39] defines events that are broadly understood by vendors of system management tools. Our position of making the decentralization of control loops explicit underlines the need for standardizing the interactions between the MAPE loop components. That includes interactions among MAPE components within a control loop as well as interactions between MAPE loop components of the same type of different control loops. This will comprise interface definitions (signatures and APIs), message formats, and protocols. The necessity of this standardization has already been appreciated in the past, e.g., in [32] the authors standardize the communication from the A to the P component by using a standard data exchange format (e.g., SOAP), but no comprehensive approach exists so far.

Control Theory. There are substantial theoretical foundations for understanding control systems in other engineering domains, embodied in control theory. In this chapter, we have defined patterns for how to assemble a particular kind of control loop (i.e., MAPE), but we have not discussed how theories and techniques from control theory apply to the control of self-adaptative systems. For example, it would be desirable to describe the transfer function of a managed system. A transfer function defines the relation between a controlled system's input and output, in particular how effectors affect subsequent sensor readings. Having such a function for the control loops of self-adaptive systems would mean that we could reason about the properties (such as stability) of the control loop being designed. Even more so, if the transfer function is available during run-time in a machine-processable way, this reasoning can be subject to run-time adaptation as well. The forms that a transfer function takes in different software domains has received scant attention. In the context of service-oriented computing, the transfer function could relate to service level agreements (SLA): sensor readings would be mapped to SLA values, so that the transfer function describes how the generic control loop needs to be controlled at the effector in order to result in the desired SLA behavior. This will probably include the mapping of certain SLAs to particular MAPE patterns that are proven to be effective with respect to these SLAs. Investigating how research from the models@runtime community can inform the definition of transfer functions for software would be a good starting point.

Adaptive coupling with mutable control patterns. Complexity theory [37] shows that the overall properties of a complex software system are largely determined by the internal structure and interaction of its parts and less by the function of its individual constituents [54]. Even more so, the internal structure of a system is formed by relationships of differing strengths between constituents. Components with tighter connections (or coupling) cluster to sub-systems, while other components may remain more loosely-coupled. Hence, a complex software system provides a mixture of tightly and loosely coupled parts. As an important consequence, the overall system properties (e.g., scalability) are determined not only by the structure but even more so by the strength of coupling of its relationships [20]. Our control patterns support different forms of coupling. For example, the information sharing pattern provides a much looser form of coupling compared to the decentralized control pattern, thus the former potentiality scales much better than the latter.

In order to use the full potential of the extended architectural expressiveness, e.g., with nested control loops, the outer loop should be able to control the strength of coupling of the inner loop. This means nothing else than "switching" from one pattern to the other during operation. Future research should investigate approaches like [41] to allow for mutable control patterns.

Pattern enumeration and application. The patterns described in the chapter do not fully enumerate all possible decentralization patterns, and in fact the patterns could potentially be combined in any number of ways (for example, in federated data centers the information sharing pattern could be used to manage adaptation between data centers, while a hierarchical pattern could be used within a data center). Future work should look at a broader range of self-adaptive systems to enumerate all the patterns that have been used successfully in practice.

Furthermore, understanding when it is best to use one pattern over another should be an active area of future research. We conceive of at least three dimensions that will affect the choice of pattern:

1. The desired quality attributes and the level of guarantee required for them. For example, it may be easier to prove that global quality attributes such as performance will be achieved in the master/slave pattern, but that scalability would be difficult to achieve.
2. The architecture of the managed system will likely influence which patterns are applicable. For example, a hierarchical pattern will be unlikely to work

if there is no obvious hierarchy of authority in the managed system, or applying the information sharing pattern will likely be influenced by how much information about the managed subsystems can be shared.
3. Domain constraints may affect the choice of a particular pattern. For example, centralizing adaptation decisions may not be possible for confidentiality reasons or because of dynamics in the network topology. In such scenarios, a decentralized solution may be preferable.

We expect that a better understanding of how the drivers relate to the patterns, and how the architecture of the managed system restricts the patterns that can be employed to manage it, will lead to more principled design of self-adaptive systems in the future.

Acknowledgments. This chapter is based on the results of a collaborative effort of a breakout group for and the Dagstuhl Seminar on Software Engineering or Self Adapative Systems, October, 2010. The authors would like to thank the other participants in the breakout session for their contributions to the discussion: Schahram Dustdar, Jeff Kramer and Rick Schlichting, as well as other attendees of the seminar.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling Dimensions of Self-Adaptive Software Systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
2. Arkin, R.: Bahavior-Based Robotics (1998)
3. Babaoglu, O., Canright, G., Deutsch, A., Caro, G.A.D., Ducatelle, F., Gambardella, L.M., Ganguly, N., Jelasity, M., Montemanni, R., Montresor, A., Urnes, T.: Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst. 1, 26–66 (2006)
4. Brittenham, P., Cutlip, R.R., Draper, C., Miller, B.A., Choudhary, S., Perazolo, M.: It service management architecture and autonomic computing. IBM Syst. J. 46, 565–581 (2007), http://dx.doi.org/10.1147/sj.463.0565
5. Brun, Y., Medvidovic, N.: An architectural style for solving computationally intensive problems on large networks. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007), Minneapolis, MN, USA (May 2007)
6. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture, A Pattern Language for Distributed Computing, vol. 4. Wiley, Chichester (2007)
7. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture, A System of Patterns, vol. 1. Wiley, Chichester (1996)
8. Cardellini, V., Casalicchio, E., Grassi, V., Lo Presti, F.: Adaptive Management of Composite Services under Percentile-Based Service Level Agreements. In: Maglio, P.P., Weske, M., Yang, J., Fantinato, M. (eds.) ICSOC 2010. LNCS, vol. 6470, pp. 381–395. Springer, Heidelberg (2010)

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, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
10. Cheng, S.W., Garlan, D., Schmerl, B.R.: Evaluating the effectiveness of the rainbow self-adaptive system. In: SEAMS, pp. 132–141 (2009)
11. Dobson, S., Denazis, S., Fernndez, A., Gati, 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)
12. Dowling, J., Cahill, V.: The K-Component Architecture Meta-model for Self-Adaptive Software. In: Matsuoka, S. (ed.) Reflection 2001. LNCS, vol. 2192, pp. 81–88. Springer, Heidelberg (2001)
13. Esfahani, N., Kouroshfar, E., Malek, S.: Taming uncertainty in self-adaptive software. In: SIGSOFT FSE, pp. 234–244 (2011)
14. Esfahani, N., Malek, S.: On the Role of Architectural Styles in Improving the Adaptation Support of Middleware Platforms. In: Babar, M.A., Gorton, I. (eds.) ECSA 2010. LNCS, vol. 6285, pp. 433–440. Springer, Heidelberg (2010)
15. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of distributed consensus with one faulty process. Journal of the ACM 32(2), 374–382 (1985)
16. Gambi, A., Pezzè, M., Young, M.: SLA protection models for virtualized data centers. In: Proc. of the Int. Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS (2009)
17. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37, 46–54 (2004)
18. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: 1st Workshop on Self-Healing Systems. ACM, New York (2002)
19. Gilbert, S., Lynch, N.: Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. SIGACT News 33, 51–59 (2002), `http://doi.acm.org/10.1145/564585.564601`
20. Goeschka, K.M., Froihofer, L., Dustdar, S.: What soa can do for software dependability. In: Workshop on Architecting Dependable Systems (WADS 2008), Supplemental Proceedings of the 38th IEEE International Conference on Dependable Systems and Networks (DSN 2008), pp. D4–D9. IEEE Computer Society (2008)
21. Gomaa, H., Hussein, M.: Software reconfiguration patterns for dynamic evolution of software architectures. In: Proceedings of Fourth Working IEEE/IFIP Conference on Software Architecture, WICSA 2004, pp. 79–88 (2004)
22. Gomaa, H., Hashimoto, K., Kim, M., Malek, S., Menascé, D.A.: Software adaptation patterns for service-oriented architectures. In: Proceedings of the 2010 ACM Symposium on Applied Computing, SAC 2010, pp. 462–469. ACM, New York (2010)
23. Hebig, R., Giese, H., Becker, B.: Making control loops explicit when architecting self-adaptive systems. In: Proceeding of the Second International Workshop on Self-organizing Architectures, SOAR 2010, pp. 21–28. ACM, New York (2010)
24. Huebscher, M.C., McCann, J.A.: A survey of autonomic computing–degrees, models, and applications. ACM Computing Surveys 40, 7:1–7:28 (2008), `http://doi.acm.org/10.1145/1380584.1380585`

25. IBM: An architectural blueprint for autonomic computing. Tech. rep., IBM (January 2006)
26. Jackson, M.: The meaning of requirements. Ann. Softw. Eng. 3, 5–21 (1997), http://dl.acm.org/citation.cfm?id=590564.590577
27. Jung, G., Hiltunen, M.A., Joshi, K.R., Schlichting, R.D., Pu, C.: Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In: Proceedings of the 2010, IEEE 30th International Conference on Distributed Computing Systems, ICDCS 2010, pp. 62–73 (2010)
28. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
29. Kircher, M., Jain, P.: Pattern-Oriented Software Architecture. Patterns for Resource Management, vol. 3. Wiley, Chichester (2004)
30. Kramer, J., Magee, J.: The evolving philosophers problem: Dynamic change management. IEEE Trans. Softw. Eng. 16, 1293–1306 (1990), http://dl.acm.org/citation.cfm?id=93658.93672
31. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington, DC (2007)
32. Leymann, F.: Combining Web Services and the Grid: Towards Adaptive Enterprise Applications. In: Castro, J., Teniente, E. (eds.) First International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) - CAiSE Workshop, pp. 9–21. FEUP Edi cões (June 2005)
33. Litoiu, M., Woodside, M., Zheng, T.: Hierarchical model-based autonomic control of software systems. In: Proceedings of the 2005 Workshop on Design and Evolution of Autonomic Application Software, DEAS 2005, pp. 1–7. ACM (2005)
34. Maes, P.: Computional reflection. Ph.D. thesis, Vrije Universiteit (1987)
35. Malek, S., Beckman, N., Mikic-Rakic, M., Medvidovíc, N.: A Framework for Ensuring and Improving Dependability in Highly Distributed Systems. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems III. LNCS, vol. 3549, pp. 173–193. Springer, Heidelberg (2005)
36. Malek, S., Mikic-Rakic, M., Medvidovic, N.: A decentralized redeployment algorithm for improving the availability of distributed systems. In: 3rd International Conference on Component Deployment, Grenoble, France (November 2005)
37. Manson, S.M.: Simplifying complexity: a review of complexity theory. Geoforum 32(3), 405–414 (2001)
38. Müller, H., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Proceedings of the 2nd International Workshop on Ultra-large-scale Software-intensive Systems, ULSSIS 2008, pp. 23–26. ACM, New York (2008), http://doi.acm.org/10.1145/1370700.1370707
39. OASIS, http://www.oasis-open.org
40. 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, 54–62 (1999), http://dx.doi.org/10.1109/5254.769885
41. Pereira, J., Oliveira, R.: The mutable consensus protocol. In: Proceedings of the 23rd IEEE International Symposium on Reliable Distributed Systems, pp. 218–227. IEEE Computer Society (October 2004)
42. Ramirez, A.J., Cheng, B.H.C.: Design patterns for developing dynamically adaptive systems. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 49–58. ACM, New York (2010)

43. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
44. Schloss Dagstuhl Seminar 10431, Wadern, Germany: Software Engineering for Self-Adaptive Systems (October 2010), `http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=10431`
45. Schmidt, D.C., Stal, M., Rohnert, H., Buschmann, F.: Pattern-Oriented Software Architecture. Patterns for Concurrent and Networked Objects, vol. 2. Wiley, Chichester (2000)
46. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
47. Shaw, M., Clements, P.: The golden age of software architecture. IEEE Softw. 23, 31–39 (2006), `http://dl.acm.org/citation.cfm?id=1128592.1128707`
48. Sykes, D., Magee, J., Kramer, J.: Flashmob: distributed adaptive self-assembly. In: Proceeding of the 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems, SEAMS 2011, pp. 100–109. ACM, New York (2011), `http://doi.acm.org/10.1145/1988008.1988023`
49. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
50. Toffetti, G., Gambi, A., Pezzè, M., Pautasso, C.: Engineering Autonomic Controllers for Virtualized Web Applications. In: Benatallah, B., Casati, F., Kappel, G., Rossi, G. (eds.) ICWE 2010. LNCS, vol. 6189, pp. 66–80. Springer, Heidelberg (2010)
51. Van Dyke Parunak, H., Brueckner, S.A., Sauter, J.A., Matthews, R.: Global convergence of local agent behaviors. In: Proceedings of the Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, AAMAS 2005, pp. 305–312. ACM, New York (2005)
52. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A framework for evaluating quality-driven self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2011, pp. 80–89. ACM, New York (2011), `http://doi.acm.org/10.1145/1988008.1988020`
53. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), Honolulu, Hawaii (2011)
54. Wegner, P.: Why interaction is more powerful than algorithms. Commun. ACM 40(5), 80–91 (1997)
55. Weyns, D., Iftakhir, M.U., Malek, S., Andersson, J.: Claims and supporting evidence for self-adaptive systems: A literature review. In: Proceedings of the 7th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2012. ACM, New York (2012)
56. Weyns, D., Malek, S., Andersson, J.: On decentralized self-adaptation: lessons from the trenches and challenges for the future. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 84–93. ACM, New York (2010), `http://doi.acm.org/10.1145/1808984.1808994`
57. Weyns, D., Malek, S., Andersson, J.: Forms: Unifying reference model for formal specification of distributed self-adaptive systems. ACM Transactions on Autonomous and Adaptive Systems, Special Issue on Formal Methods for Pervasive, Self-Aware, and Context-Aware Systems 7(1) (2012)