Pervasive and Mobile Computing 17 (2015) 184–206

Contents lists available at ScienceDirect

Pervasive and Mobile Computing

journal homepage: www.elsevier.com/locate/pmc

A survey on engineering approaches for self-adaptive systems

Christian Krupitzer [a,*], Felix Maximilian Roth [a], Sebastian VanSyckel [a], Gregor Schiele [b], Christian Becker [a]

[a] University of Mannheim, Schloss, 68131 Mannheim, Germany

[b] Insight Research Centre, National University of Ireland, Galway IDA Business Park, Galway, Ireland

ARTICLE INFO

Article history:
Available online 8 October 2014

Keywords:
Taxonomy
Self-adaptation
Survey
Self-adaptive systems
Context adaptation

ABSTRACT

The complexity of information systems is increasing in recent years, leading to increased effort for maintenance and configuration. Self-adaptive systems (SASs) address this issue. Due to new computing trends, such as pervasive computing, miniaturization of IT leads to mobile devices with the emerging need for context adaptation. Therefore, it is beneficial that devices are able to adapt context. Hence, we propose to extend the definition of SASs and include context adaptation. This paper presents a taxonomy of self-adaptation and a survey on engineering SASs. Based on the taxonomy and the survey, we motivate a new perspective on SAS including context adaptation.

1. Introduction

The complexity of modern pervasive information systems is increasing. Due to the growing number of powerful mobile and embedded devices as well as the omnipresence of relatively high speed wireless networking, users today expect systems to operate whenever and wherever they want, while traveling, at home, at work, or during vacation. Systems are highly distributed and must integrate all available, highly specialized and heterogeneous devices (ranging from embedded sensor nodes to Cloud servers) and data streams (including web data and real time sensor data) that operate in an ever-changing environment with fluctuating network resources and availability. In additions, systems are no longer restricted to small, tightly controllable areas with single administrative responsibility, like smart rooms or buildings but are interconnected, leading to truly pervasive, global systems like Smart Cities or the Internet of Things.

Developing, configuring, and maintaining such systems is a very difficult, error prone, and time consuming task. One promising way to reduce this effort is self-adaptation. A *self-adaptive system* (SAS) is able to automatically modify itself in response to changes in its operating environment [1,2]. The modification is done by adjusting attributes (parameters) or artifacts of the system in response to changes in the system itself or in its environment. In recent years, SASs have seen an increasing level of interest in different research areas like Pervasive Computing, Autonomic Computing [2], and Nature-Inspired (Organic) Computing [3].

SASs provide so called self-* or self-management properties like self-configuration, self-healing in the presence of failures, self-optimization, and self-protection against threats [2,4]. For achieving adaptive behavior, basic system properties are self-awareness and context-awareness [5]. Self-awareness describes the ability of a system, to be aware of itself, i.e., to

* Corresponding author. Tel.: +49 6211812104.
E-mail addresses: christian.krupitzer@uni-mannheim.de (C. Krupitzer), felix.maximilian.roth@uni-mannheim.de (F.M. Roth), sebastian.vansyckel@uni-mannheim.de (S. VanSyckel), gregor.schiele@deri.org (G. Schiele), christian.becker@uni-mannheim.de (C. Becker).

http://dx.doi.org/10.1016/j.pmcj.2014.09.009
1574-1192/

be able to monitor its resources, state, and behavior [6]. Context-awareness means that the system is aware of its operational environment, the so called context [7]. According to Dey, context is "any information that can be used to [characterize] the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves" [8]. The system uses sensors to collect information about its context and reasons about the information.

In this paper, we provide a structured overview of self-adaptation and approaches for engineering SASs, analyze future research directions, and motivate the need for a new perspective on self-adaptation in pervasive computing systems. Our main contributions are as follows: First, we develop a taxonomy for self-adaptation that integrates existing views on self-adaptation and specifically context-adaptive systems, which are most relevant to pervasive computing. Second, we survey existing approaches for engineering SASs. Third, we discuss a new type of SASs.

These contributions are directly reflected in the structure of the remaining part of the paper. In the next section, our taxonomy for self-adaptation is presented. In Section 3, we present approaches for engineering SASs. Based on the taxonomy and the approaches, in Section 4 we describe a new perspective of a SAS. A conclusion closes the paper.

2. Self-adaptation

In this section, we summarize different aspects and perspectives on self-adaptation in SAS research and adaptation in general, e.g., in pervasive systems, and present them in a comprehensive taxonomy for self-adaptation. Our taxonomy incorporates the results of an extensive literature review and integrates different existing taxonomies and works on (self-)adaptation. Fig. 1 shows an overview of our taxonomy. In the remaining part of this section we discuss the different dimensions of our taxonomy.

Our taxonomy presents important characteristics of self-adaptation. These issues must be addressed by the implementation of the adaptation logic of such adaptive systems. They influence the reasoning about adaptation as well as monitoring. The presentation of techniques for the implementation of the adaptation logic follows in Section 3.2.

Before going into detail about our taxonomy, we first provide an overview of the main sources and influences for it. Different taxonomies for self-adaptation or adaptation have been developed over the years. As one of the first ones, Rohr et al. describe a classification scheme for self-adaptation research in 2006 [9]. In their work, they classify self-adaptation among the dimensions *origin*, *activation*, *system layer*, *controller distribution*, and *operation*. In 2009, Salehie and Tahvildari present an overview over the landscape of self-adaptive software and related research challenges, including their own taxonomy for self-adaptation [5]. A current overview of adaptation can be found in [10], in which Handte et al. classify the adaptation support for pervasive applications into the dimensions of *time*, *level*, *control*, and *technique*. Macías-Escrivá et al. describe in a recent survey current approaches, research challenges, and applications for SASs [11]. All these surveys provide important insights into the field of SAS. However, none of them gives an integrated view, incorporating all different existing views and aspects. In contrast, the goal of our work is to present such a uniform taxonomy for self-adaptation.

Additionally, several works discuss aspects of self-adaptation. McKinley et al. highlight the difference regarding *parameter* vs. *compositional* adaptation [12]. In [1], the authors discuss the spectrum of adaptation from static activities to dynamic ones. In 2008 and 2010, two Dagstuhl seminars focused on research issues regarding the engineering of SASs [13,14]. Furthermore, there are two surveys that focus on Autonomic Computing. In [4], Huebscher and McCann present an overview of Autonomic Computing and its applications, whereas Dobson et al. highlight autonomic communications in [15].

In [5], Salehie and Tahvildari introduce the 5W + 1H questions for eliciting adaptation requirements:

- When to adapt?
- Why do we have to adapt?[1]
- Where do we have to implement change?
- What kind of change is needed?
- Who has to perform the adaptation?
- How is the adaptation performed?

Other authors formulate similar questions (e.g., [12,16–18]). According to Salehie and Tahvildari, the questions must be addressed during implementation of a SAS [5]. Therefore, it seems reasonable to answer these questions when speaking about adaptation. Our taxonomy shows, how we answer the six questions.

As mentioned above, the different aspects of the taxonomy answer the 5W + 1H questions. However, our taxonomy has a different view on the dimension known as type of control. Automatic control is distinguished from manual control [10]. As a SAS should adapt without user involvement, we do not include this aspect in our taxonomy. In other words, the question: "*Who has to perform the adaptation?*" is not answered with our taxonomy, because the nature of a SAS leads to an automatic type of adaptation. Table 1 shows how the taxonomy answers the questions for adaptation. In the rest of the section, we present our taxonomy in more detail.

[1] The why question was changed for this work. In [5], it was understood as motivation for building SASs.

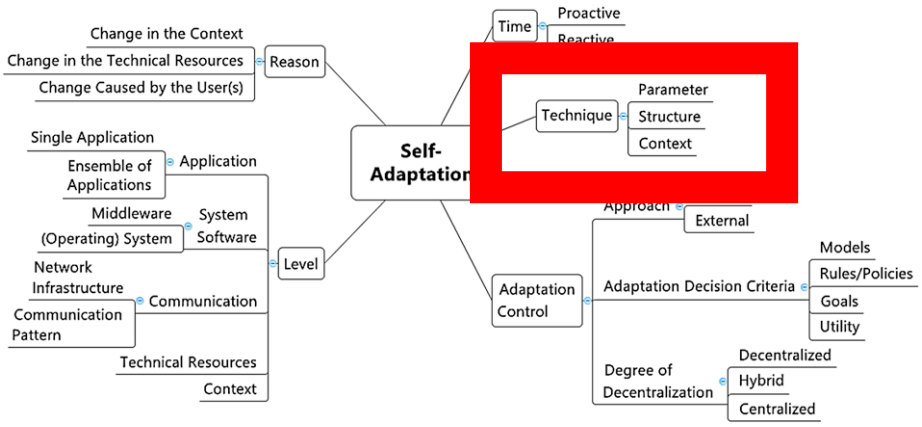


Fig. 1. Taxonomy of self-adaptation.

Table 1
Relation of the taxonomy dimensions and the questions.

Question	Dimension of the taxonomy
When?	Time (reactive vs. proactive)
Why?	Reason (context, technical resources, user)
Where?	Level (application, system software, communication, technical resources, context)
What?	Technique (parameter, structure, context)
Who?	N/A (nature of a SAS leads to an automatic type of adaptation)
How?	Adaptation control (approach, adaptation decision criteria, degree of decentralization)

2.1. Time

The *time* aspect is related to the when-question: "*When should we adapt?*". The "traditional" view is a *reactive* adaptation: After an event that causes the need for adaptation, e.g., a change in the resources or a drop in performance, an adaptation plan is worked out [10]. Rohr et al. identify two further perspectives that plan adaptation before an actual event happens: (i) *predictive* and (ii) *proactive* [9]. Here, the predictive dimension of time describes a situation, in which a system identifies the need of adaptation before a drop in performance occurs, whereas proactive is an adaptation to improve the performance without foregoing or anticipating drop in performance. Handte et al., however, divide the temporal aspects of adaptation in two dimensions: (i) *reactive* and (ii) *proactive*. The reactive dimension is in accordance with Rohr et al.'s description, however, proactive is defined as "modifications of an application performed before an application can no longer be executed" [10]. Therefore, Rohr et al.'s [9] *predictive* dimension is equal to the *proactive* dimension of Handte et al. [10], and the *proactive* dimension of Rohr et al. is not explicitly mentioned by Handte et al. [10]. In this work, we use the temporal definition of Handte et al. The split of time into three dimensions is not necessary. The distinction of adaptation *before* or *after* the need for adaptation is enough for describing the temporal aspect of adaptation. The third perspective introduced by Rohr et al. – continuous adaptation for performance improvements – is part of the self-optimization property [2]. Self-optimization is a subset of self-adaptation and, therefore, is implicitly included [5].

The time of adaptation is a central question. From the user's point of view, proactive adaptation is preferable, since it avoids interruptions in the user's workflow with the system. On the other hand, the prediction algorithms needed for proactive adaptation have several issues. They are complex to develop, their suitability is highly dependent on the specific prediction tasks, and faulty results can cause suboptimal or malicious adaptations. Therefore, many approaches focus on reactive adaptation [10]. However, the choice for proactive or reactive adaptation is not exclusive. Mapping the adaptation process to the Autonomic Computing MAPE cycle, the basic functionality for adaptation is monitoring the environment, analyzing for change, computing adaptation plans, and executing these plans [2]. Reactive and proactive adaptations involve similar activities regarding monitoring, planning, and executing, but strongly differ in the analyzing phase. With reactive adaptation, the monitored data is analyzed for abnormal patterns. With proactive adaptation, the monitored data is used to forecast system behavior or environmental state. It is possible to combine proactive and reactive adaptation such that proactive adaptation is the goal, and reactive adaptation is used as a back-up mechanism, i.e., if a change was not predicted (e.g., failing of a component).

An additional aspect related to time is the type of monitoring [5]. *Continuous monitoring* describes a constant monitoring effort regarding the resources and the environment. *Adaptive monitoring* refers to monitoring of selected features and, in case of anomalies, the monitoring process is intensified. The decision between continuous vs. adaptive monitoring influences the cost of monitoring and, hence, of the self-adaptation process. However, as it is an implementation detail and not related to the adaptation decision itself, it is not included in our taxonomy.

2.2. Reason

In general, adaptation is a reaction to a change. On the one hand, this reaction is costly. Accordingly, the type and impact of a change should be clearly identified in order to decide, if an adaptation is necessary. On the other hand, the parameters of an adequate adaptation are determined by the source of the change. Therefore, it is important to identify the *reason* for an adaptation: "*Why do we have to adapt?*". This is a central question influencing the reaction, because different reasons result in different adaptation activities.

In a SAS, the reason for an adaptation is a change in one or various system elements:

- (i) a change in the technical resources, e.g., a defect of a hardware component, software fault, or the availability of an alternative network connection,
- (ii) a change in the environment, e.g., the state of a context variable changed, or
- (iii) a change regarding the user, e.g., a change in the composition of the user group or the user preferences.

Therefore, in our taxonomy, adaptation can be triggered by changes in the technical resources, the context, or the user(s). The explicit inclusion of the user as a reason for adaptation is not mentioned in other taxonomies.

Other taxonomies for (self-)adaptation do not include the reasons for adaptation. An explanation could be that the reason for an adaptation is not seen as part of the decision, i.e., how the system should recover from a change or the adaptation process itself. In our taxonomy, we explicitly include the reason for an adaptation because it is the trigger of the adaptation process. In our understanding, it is fundamental for the adaptation decision to identify the reason for adaptation and to reason about it. Consequently, the question "*Why do we have to adapt?*" has to be included in the aspects of self-adaptation. Furthermore, the reasons for adaptation should be included in the aspects of self-adaptation, because they determine the elements that have to be monitored. Without an adequate monitoring process, the need for an adaptation cannot be identified and, therefore, the adaptation process cannot be started. Accordingly, it is important to monitor the state of the technical resources (including hardware and software), the environment, and the users interactions with sensors and interfaces for user interaction.

2.3. Level

Adaptation can be implemented on different *levels* in the system. In order to answer the question of "*Where do we have to implement change?*", one must be aware of the different levels of a SAS. In general, a SAS is composed of different elements: the managed element(s) and the adaption logic [19]. While the adaptation logic – as the control unit of the technical resources – often stays stable, the managed elements can be adapted. Furthermore, it is possible to adapt the environment and the user(s). This extends the common view on SASs, where the environment is only monitored, not changed. An adaptation of the adaptation logic itself would enable an improvement of the adaptation performance over time, but is out of scope of this work.

The managed elements are composed of various levels. The technical resources are hardware and other managed resources, such as computers, smartphones, robots, traffic signs, or production facilities. The hardware is controlled by system software, i.e., an operating system and – in case of distributed systems – middleware. On top of the system software, the application is set up. The application can be an application running on a single device, or a distributed application split in application parts running on various devices simultaneously. Both are seen as single application in the following. Furthermore, various different applications can run simultaneously, as well as interact and form ensembles of applications. This can lead to interferences, i.e., undesired interceptions and dependencies regarding the use of resources [20]. For the interaction between the managed resources as well as the adaptation logic elements, communication is needed. Communication is seen in two perspectives. The *network infrastructure* is the physical network connection, consisting of network cards, routers, WLAN, etc. *Communication patterns* are the logical communication, i.e., the style of interaction between the elements. Possible implementations can be event-based communication or pub/sub communication. An additional level besides the technical resources and software is the system's context. Context-altering systems are able to adapt their context [7]. This extends the former view on SASs, where the environment is monitored but not explicitly altered. Implicitly, this may be done by actuators of the technical resources, but so far is not explicitly controlled by the adaptation logic. In current SAS literature, context adaptation is not included in the level of adaptation.

Adaptation can happen on all levels [9,10]: Smartphone apps that switch to silent mode when the user is in a meeting, e.g., detected by using calendar information, offer adaptation on the application level. Adaptive middleware offers the possibility to exchange components at runtime [21,22]. Autonomic communication techniques enable adaptation on the network level [15]. An example adapting the communication is switching the network connection, e.g., from 3G to WLAN as far as a WLAN connection is available. Self-healing capabilities enable the automatic start of back-up systems, e.g., in a data center, which alters the technical resources. Context-adaptive applications can adapt their behavior to the surrounding context, or adapt the context through actuators [7]. An example for context adaptation is a smart meeting room that automatically dims the light when a presentation starts.

The adaptation logic of a SAS must be aware of these different levels, possible adaptation alternatives, and define adaptation plans for the appropriate levels. Therefore, the question: "*Where to adapt?*" must be answered on the correct level(s) in order to achieve the system's goals. In case the adaptation alternatives can target different levels, the adaptation

logic must decide which plan to execute based on criteria like cost, e.g., in terms of time, or achievable utility for the system. Adaptation of the user is possible in theory, but not desirable in practice, as the application is for the user and not the other way round. Hence, we do not include adaptation of the user in our taxonomy.

2.4. Technique

It is not sufficient to only identify the levels, where the adaptation should take place. Additionally, the specific adaptation actions that should be carried out on those levels need to be identified, i.e., "*What kind of change is needed?*". In literature, different *techniques* for adaptation can be found.

McKinley et al. distinguish between two approaches for adaptive software: (i) *parameter adaptation* and (ii) *compositional adaptation* [12]. Parameter adaptation achieves a modified system behavior by adjusting system parameters. This can be achieved quite easily, as the adaptation logic must only control and change parameters. On the other hand, changing parameters can involve high complexity, if the parameters are depended on each other. Furthermore, in case different algorithms for one component exist, it is possible to switch between them. However, the dynamic integration of new algorithms at runtime is not possible. The dynamic integration of new components is not possible as well. An example for parameter adaptation is a rule-based system, in which rules specify the necessary amount of servers running, depending on the current workload of the servers. In this setting, it is only possible to add new servers as long as configured servers are available. However, it is not possible to dynamically transfer the responsibilities of a faulty server instance to a new server instance, as such an exchange affects the system's structure. Compositional adaptation enables the exchange of algorithms or system components dynamically at runtime. Therefore, it is possible to exchange defect components in order to prevent performance loses, improve the performance by adding new components, or adjust the system to new circumstances. To use the server example above, this means that during the execution of the application, additional servers can be integrated into the system, or that a new server instance can overtake the responsibilities and tasks of another server instance.

A third technique can be derived from Pervasive Computing. Here, systems are context-adaptive and can alter the context in which the systems are running. Therefore, we extend the range of techniques by context adaptation. Context adaptation is not addressed by McKinley et al. [12]. Whereas the monitoring of the context and the detection of contextual changes is supported by many SAS, context alteration is often not integrated in SASs.

Handte et al. categorize adaptation techniques in *behavior*, *composition* and *context* [10]. Here, parameter adaptation is a form of behavior adaptation. As a change in composition can also lead to a modified behavior, this view overlaps. In this work, the techniques for adaptation are categorized in *parameter*, *structure*, and *context*. This is a combination of the approaches mentioned by McKinley et al. and Handte et al. *Parameter* refers to adaptation through the change of parameters. *Structure* subsumes change in the structure of the technical system, such as the exchange of components, a new composition of components, or the removal/addition of components. Further, changes in the relation between elements, technical resources or the environment/user(s), are structural adaptations as well. *Context* refers to any changes in the context, e.g., modifying the state of context variables via actuators. Combinations of techniques in one adaptation plan is possible, e.g., changing parameters of one component and adding further ones.

2.5. Adaptation control

A SAS is composed of the adaptation logic and the managed resources. Responsible for controlling the adaptation is the adaptation logic. This involves monitoring the managed system as well as the environment and the user(s), analyzing the monitored data for anomalies, planning the adaptation, and executing the adaptation plans. Therefore, the adaptation logic is responsible for answering the question: "*How to adapt?*". By answering this question the adaptation logic determines how to perform adaptation.

Two approaches for implementing the adaptation logic can be found in literature. SASs following the *internal approach* intertwine the adaptation logic with the system resources. The *external approach* splits the system into adaptation logic and managed resources, which increases maintainability through modularization [5,23].

The control unit needs a metric in order to decide how to adapt. Different metrics are present in the literature: models, rules and policies, goals, or utility functions [24]. The different adaptation possibilities must be analyzed with the help of the criteria and the best one must be chosen. Different criteria can be combined, e.g., goal model based planning where goals have additional utility values for solving conflicts between goals.

Another aspect of the adaptation logic is the degree of decentralization. A centralized adaptation logic can be a solution for systems with a small amount of resources to manage. When it comes to large systems with many components to manage, a decentralized approach for a split of the responsibilities can improve the system performance for adaptation [19]. Various degrees of decentralization are possible. In fully decentralized approaches, each sub-system has a complete adaptation logic and different patterns of communication are possible [19]. Hybrid approaches add central components to decentralized approaches or distribute the adaptation logic functionality to sub-systems.

In this section, we presented a taxonomy of self-adaptation. We combined different works on self-adaptation and adaptation in general. Self-adaptation can be described with the dimensions: Time, Reason, Level, Technique, and Adaptation Control. Developers of SASs must be aware of these dimensions and their systems must integrate these dimensions for

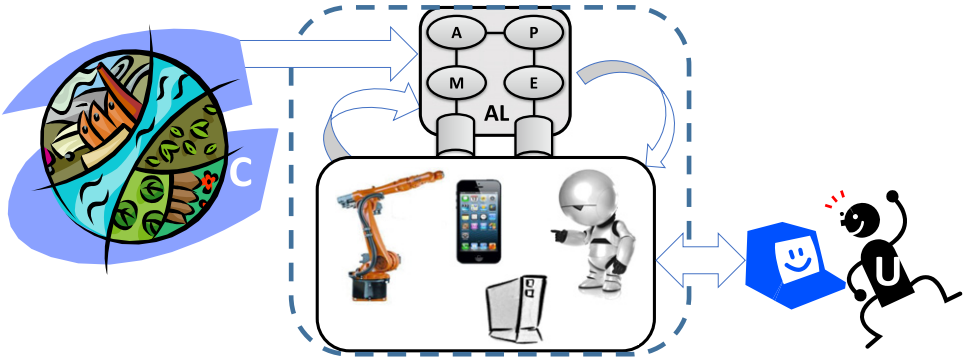


Fig. 2. A SAS (AL = Adaptation Logic, MR = Managed Resources, U = User(s), C = Context, M,A,P,E = MAPE functionality).

monitoring and reasoning about adaptation. In Section 3, we give a detailed introduction to the adaptation logic. Further, we discuss different approaches for developing a SAS's adaptation logic.

3. Engineering self-adaptive systems

In the previous section, we presented our understanding of self-adaptation by creating a taxonomy for self-adaptation. The dimensions of the taxonomy are important aspects for a SAS's adaptation logic to reason about adaptation. In this section, we want to highlight the "*How?*" aspect – the construction of a SAS's "brain" – the adaptation logic. The other dimensions of the taxonomy influence monitoring, reasoning, and executing, controlled by the adaptation logic. Therefore, we present the general structure of the adaptation logic of a SAS as control unit of the adaptation process and discuss general implementation issues. Subsequently, we discuss different approaches for constructing SASs and show which aspects of our taxonomy they address.

3.1. Adaptation logic issues

A SAS is composed of managed resources and the adaptation logic. This can be represented as a tuple $SAS = (AL, MR)$ with the adaptation logic AL and the managed resources MR [19]. Both, adaptation logic and managed resources, can be divided into various elements. The adaptation logic – as control unit for the adaptation [25] – is modeled as group of adaptation logic elements $AL = al_1, \ldots, al_n$ and monitors the environment (M), analyzes the data for change (A), plans adaptation (P), and controls the execution of the adaptation (E). These activities are known from Autonomic Computing as *MAPE cycle* or *MAPE functionality* [2]. Other authors propose similar feedback loops for SASs (e.g., [15,26,27]). Therefore, in this work, the MAPE cycle is used as basic feedback structure for the adaptation logic. The adaptation logic can be complemented by additional elements, such as a knowledge component responsible for managing content (e.g., monitoring values, rules, or policies) or a learning component. The managed resources are a group of resources $MR = mr_1, \ldots, mr_n$, such as hardware with software, smart phones, robotics, or unmanned vehicles. Fig. 2 presents the basic layout of a SAS. The dashed line shows the system border.

The dimensions Time, Reason, Level, and Technique of our taxonomy can be mapped to the MAPE functionality. The *Time* dimension influences the decision of analyzing algorithms, as proactive recognition of the need for adaptation has other requirements – especially the need for predictions – as reactive detection of changes. Monitoring should be continuous no matter whether the adaptation is proactive or reactive. The *Reason* dimension influences monitoring, analyzing, and planning, as it describes the reasons for adaptation and, therefore, the aspects that should be monitored, where analyzing has to determine changes as well as the metrics, that must be addressed with the adaptation plans. The *Level* for adaptation is obviously important for planning and executing as these activities must be aware of the elements that should be adapted. Monitoring has to determine the elements for the levels that are present in the managed resources. The *Technique* dimension influences the planning and executing, as planning describes which adaptation techniques to use on which elements and executing controls the execution of the techniques. The fifth dimension *Adaptation Control* describes the structure of the adaptation logic and is not related to any specific MAPE functionality. Table 2 presents the mapping of the MAPE functionality to the dimensions of our taxonomy.

Abstracted from the concrete implementation of the adaptation logic, general issues can be identified. The adaptation logic can be intertwined with the rest of the application or separated. For analyzing and planning of the adaptation, the adaptation logic can use different metrics. Another issue is the degree of decentralization of the logic and the distribution of the MAPE functionality on different sub-system parts. Following, these issues are presented in detail.

3.1.1. Implementation approaches

In literature, two approaches regarding the interplay of adaptation logic and managed resources can be found [23]. The *internal* approach intertwines the application logic and the managed resources. Sensors, effectors, and adaptation

Table 2
Relation of the MAPE activities and the dimensions of the taxonomy.

	Time	Reason	Level	Technique
Monitoring	Continuous	What to monitor	Identification of the levels	–
Analyzing	Algorithms depend on reactive or proactive dimension	Where to analyze	–	–
Planning	–	What should be influenced by planning	Adaptation plans address these levels	Plans for performing the techniques
Executing	–	–	Execution of the change on the levels	Execution of the techniques on different elements

logic elements are mixed with the managed resources. This approach has several drawbacks: issues in scalability and maintainability can arise, global information of the system is not guaranteed, and testing of the adaptation logic is complicated [23]. Nevertheless, the internal approach can be suitable for local adaptations, e.g., exception handling.

The *external* approach separates the adaptation logic and managed resources [23] and connects them via sensors and effectors [2]. Sensors are interfaces used by the adaptation logic for getting information from the managed resources, e.g., the system state or performance measurements. Effectors are used by the adaptation logic for enabling adaptation, e.g., through changing parameters or starting components. The external approach addresses the drawbacks of the internal one and offers scalability as one adaptation logic can manage various resources, maintainability as the responsibilities are divided and can be maintained separately, and eases the achievement of a global view. Furthermore, it offers reusability of the adaptation logic or at least of the processes and algorithms used in the adaptation logic [5,23]. The external approach is superior in most cases and can be found more often in literature [23].

3.1.2. Adaptation decision criteria

Different approaches for analyzing and planning the adaptation are present. These approaches need a metric for identifying the need for adaptation and for choosing suitable adaptation plans, respectively. These metrics are based on: *models*, *rules*/*policies*, *goals*, or *utility* [24].

In model-based approaches, models represent the actual and the desired situations. They include goals, the system architecture, the environment, or other circumstances. Through analysis of the models, suitable adaptation plans are worked out. For rule-based or policy-based approaches, rules or policies determine, how the system should react in different situations and how to adapt. Often, rules/policies are defined at design time, which leads to non-dynamic approaches. Goal-based approaches aims at fulfilling specific system goals. These goals influence, how the system should perform. During the planning process, the adaptation logic must define adaptation plans for achieving these goals. One has to mention, that the goals can be contradicting, which must be solved by the adaptation logic. In utility-based approaches, utility is a function of the system value for the user and involved costs. The goal is to maximize the overall system utility. The adaptation logic evaluates the utility values of adaption strategies and selects the one with the highest utility. Disadvantageous are the difficulty of defining utility functions, as well as the complexity and the uncertainty in calculating adaptation costs and utility values.

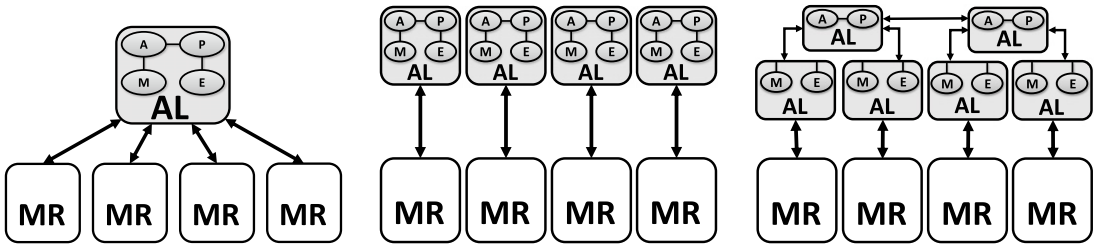
So far, most SASs monitor the context for detecting changes in the context. The explicit inclusion of context adaptation through actuators is not included in the decision criteria for the analyzing and planning. This could be done for reducing problems with plans that lead to unanticipated changes resulting of context adaptation.

3.1.3. Degree of decentralization

Usually, SASs are systems-of-systems. SASs can be found in cloud computing, traffic control systems, production facility control, or pervasive systems. Central questions are, whether the adaptation logic should be decentralized or centralized and how the MAPE functionality should be distributed. In [28], the authors define self-adaptation as a top-down approach for system control. This means that a central unit has to control the system. Self-organization is seen as the opposite, a bottom-up approach. Dedicated units organize and coordinate themselves without a central instance. In this work, self-adaptation subsumes both. *Centralized self-adaptation* with a central instance for control is defined as self-adaptation in [28]. Self-organization with decentralized system control is in this work *decentralized self-adaptation*. In case the adaptation logic has to be distributed, developers have to define an interaction pattern.

Within a *centralized* adaptation logic (see Fig. 3(a)), one sub-system implements the adaptation logic, is responsible for monitoring the context and all resources, and controls adaptation. Whereas a global maximum can be achieved because the central instance is aware of all information, this approach is not suitable for large systems, with a high amount of information, or resource-poor devices because of the calculation power needed [19,29]. For large systems, a central approach is hard to achieve because of the size of the system and real time constraints.

One alternative is a *fully decentralized* approach (see Fig. 3(b)), in which each sub-system has its own application logic with full MAPE functionality and adapts itself. Furthermore, the adaptation logic elements can communicate for achieving global goals. In the IBM reference model for Autonomic Computing [2,30], additional orchestrating autonomic managers



(a) Centralized adaptation logic. (b) Decentralized adaptation logic. (c) Hybrid adaptation logic.

Fig. 3. Comparison of adaptation logic structures (AL = Adaptation Logic, MR = Managed Resources, M,A,P,E = MAPE functionality).

improve the coordination, but introduces a *hybrid approach*. Weyns et al. describe patterns for how to distribute the MAPE functionality on various sub-systems with different levels of interaction and decentralization [19]: (i) *Coordinated Control Pattern*, (ii) *Information Sharing Pattern*, (iii) *Master/Slave Pattern*, (iv) *Regional Planning Pattern*, and (v) *Hierarchical Control Pattern*. The patterns (i) and (ii) are fully decentralized with interaction, whereas the patterns (iii)–(v) have centralized elements and, therefore, are hybrid approaches. Fig. 3(c) shows one concrete implementation of the Regional Planning Pattern with 2 groups as an example for a hybrid approach.

Designers of SASs must be aware of the implications resulting from the distribution of their applications. It is important, to decide for a suitable pattern depending on the number of sub-systems, coordination needed between them for adaptation decision making, or relevance of achieving global goals.

These three issues – *approach* for the adaptation logic, *adaptation decision metrics*, and the *degree of decentralization* – influence the design of an adaptation logic. Furthermore, monitoring and reasoning (analyzing and planning), must include the dimensions presented in our taxonomy. In the following, we present different approaches for SASs.

3.2. Survey on engineering self-adaptive systems

So far, we presented different aspects of the adaptation logic's implementation like the integration of adaptation logic, adaptation metrics, and decentralization of the logic. These are general issues. We have not answered the question, how a concrete adaptation logic is implemented. In literature, different approaches for SAS construction can be found. In this section, we categorize and present these approaches. Furthermore, we link the approaches to the dimensions of our taxonomy and the MAPE activities.

Our objective is to present the diversity of approaches that are present in literature and their appropriateness for SAS development. We do not want to discuss every category and approach in full detail nor do we claim to present all available literature for each category as this would go beyond the scope of this paper. In literature, many works detail the use of the approaches, discuss their strengths and weaknesses, and point to similar works. The interested reader is referred to the cited works for more detailed information about the approaches.

3.2.1. Model-based approaches

In *Model-Driven Engineering* (MDE), the focus is on models as first class entities for describing software and its environment [31]. Incomplete information at design time and changing conditions at runtime leads to a shift in the use of models from using models for design to the use of models at runtime [32]. Therefore, *runtime models* or *models@run.time* enable monitoring, e.g., by representing the system and its environment, and reasoning, e.g., comparing a model capturing the actual system state with a desired system state, at runtime and provide abstractions from code. In this section, we present different model-based approaches for SASs that use models for monitoring and reasoning. Therefore, we focus on runtime models and neglect the use of models at design time.

According to [4] and [33], three types of models are used for monitoring and reasoning in a SAS:

- System models represent the system state.
- Goal models, policies and rules, or utility functions model the adaptation decision criteria.
- Environment models capture the context.

In the following, we present the first two in more detail. More information on context modeling can be found in literature (e.g., in [34]). Three types of *system models* can be found in literature: *architectural models*, *feature models*, and *behavioral models*. Architectural models represent the system's architecture (e.g., [23,35,36]). These models capture the architecture in different representations, e.g., as components [37] or layers [38]. In Section 3.2.2, we enlarge on the use of architectural models. Another type of system models are feature models that capture different features of software [39]. Defining all possible features and combinations of them with feature models offers a way for reasoning about adaptation by representing all possible configurations of a SAS and choosing the most suitable configuration [40]. Feature models for SASs are used by different authors (e.g., [21,40–43]). Often, feature models are used as *variability models* which capture commonalities and variability in a system family [40,44]. A third type of system models, behavioral models, describe the behavior of the

system and the possible transitions between the different system behaviors [45]. Therefore, they are applicable for describing behavioral adaptation. Often, state automata [46] or UML state machines [47] are used as notation for behavioral models.

Goal models describe the goals, that a system should fulfill [38], or its requirements [48], respectively. If the system does not fulfill its goals, adaptation is required. Goal models are used for reasoning as well as creation of adaptation plans (e.g., [38,48–50]).

In literature, different approaches using model-driven techniques for SASs can be found. The Software Engineering Institute defines a *Software Product Line* (SPL) as "a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and that are developed from a common set of core assets in a prescribed way" [51]. Besides the first authors, Gomaa and Hussein presented the *Reconfigurable Evolutionary Product Line Life Cycle* based on SPLs [52]. Hallsteinsen, Hinchey, and Schmid propose a dynamic SPL approach [53], which is used in [21] in combination with models@run.time. Further approaches that use dynamic product lines can be found (e.g., [40,43,52,54–58]). All of these approaches use the SPL principle for modeling different system configurations, often with feature models [43] or variability models [40]. The most suitable configuration is chosen and the system is adapted accordingly, e.g., with the help of component-based [52] or aspect-oriented programming techniques [21]. Therefore, the SPL approaches address mostly analyzing and planning. The *MUSIC* framework offers model-based development for SASs by integrating service-orientation and component-based development [23,59,60]. Design models are used in the design phase for specifying the system components and the initial system configuration. Component models represent the structure of the components. Runtime adaptation models are used for modeling system adaptation at runtime [60]. The MUSIC framework addresses all MAPE activities. Meta-modeling is essential for model-driven development [61]. Therefore, runtime models must be constructed on base of meta-models. Vogel, Seibel, and Giese propose to use *megamodels* for subsuming different runtime models and reasoning about adaptation on higher levels of abstraction [62]. They integrated their megamodels in a model-driven architecture with a model transformation engine between the adaptation logic and the managed system [63,64]. Lehmann et al. identify the need of additional runtime capabilities for meta-models and propose an approach for meta-modeling of runtime models [65].

Models are often used to support monitoring and reasoning in SASs. Different authors propose the use of models@run.time for SAS construction (e.g., [32,66–70]). Nevertheless, models are only a supporting tool for monitoring and reasoning that need to be accompanied by additional techniques for the change of the managed resources, such as aspect-oriented programming (e.g., [21,41,71]), component-based programming (e.g., [52,72]), architecture-based approaches (e.g., [1,35,69]), service-oriented approaches (e.g., [60,73]), or integration in frameworks, e.g., for context-aware systems [70]. In the further sections, we present these categories in detail. Furthermore, models can be used for checking and verifying system states [31,71,74].

3.2.2. Architecture-based approaches

A software architecture is a set of software elements (or components), the relations between them, as well as the properties of both [75] and denotes the high-level structure of software [76]. Regarding adaptation purposes, software architectures are used for different activities. A SAS must be aware of its structure, which is called self-awareness [5]. Software architectures can be used for representing the system structure and reasoning about adaptation levels [15]. On the other hand, software architectures are used for the construction of SASs and defining responsibilities [38]. In this section, we present architecture-based approaches.

The *Rainbow framework* [35] is one of the most well-known frameworks for SASs. Driven by external control for self-adaptation, the SAS is divided into an architecture layer and a system layer with the managed resources. The architecture layer has different components, which define adaptation plans. A translation infrastructure translates and controls the deployment of these plans in the system layer. For supporting runtime adaptation, an architectural style notion of the system is used and extended with adaptation operators and strategies. This architectural notion offers the possibility to monitor the resources as a system model and reason about change. Rainbow supports structural and parameter adaptation on different system levels. The *3L Approach* of Kramer and Magee [38] offers a three layer architecture model for self-management based on Gat's three layer sense–plan–act architecture [77]. The Component Control layer controls the managed resources. On top of the Component Control layer, the Change Management layer monitors the system, signals changes to the uppermost layer, and implements adaptation plans on the components, e.g., compositional adaptation through switching components. The Goal Management layer on top of the Change Management layer, works out adaptation plans based on goals the SAS should fulfill. As the planning process is a reaction on changes, only reactive adaptation is supported.

Further approaches use agent-related techniques for implementing architectural change (e.g., [78,79]; see Section 3.2.7) or dedicated architectural components for controlling the adaptation as Rainbow's architecture layer [35], the *Architectural Run-time Configuration Manager* [80], or the Architecture evolution manager in *Archstudio* [1,36]. Architecture-based approaches are accompanied by factors for controlling the adaptation as strategies (e.g., [35]), policies (e.g., [2,30,78, 80,81]), goals (e.g., [38,82,83]), task characteristics (e.g., [84]), or constraints (e.g., [35]). Architecture-based adaptation research is complemented with research on improvements in the use of architectures and architecture adaptation in a SAS, e.g., architectural patterns [19,85–88], resource prediction for improving self-adaptation [89], or dynamic architectural styles [90].

Many architecture-based approaches represent the architecture in the form of models. Therefore, they use architectural models for monitoring the resources and/or reasoning about adaptation (e.g., [1,23,35–37,69,72,91–96]). Architecture

Description Languages (ADL) describe architectural models. Different authors have developed ADLs or extended existing ones for SASs, e.g., *DARWIN* ([97] used, e.g., in [98]), *C2/xADL* [36], *Dynamic Wright* [99], *Gerel* [100], *CHAM* [101], *COMMUNITY* [102], *Rapide* [103], *LEDA* [96], *ACME* [91,92], or *STITCH* [104]. A comparison of ADLs can be found in [105].

All architecture-based approaches have similarities: All implement – mostly implicitly, i.e., without dedicated components – the MAPE functionality. For reasoning about the system's structure, architectural models are used in combination with metrics as policies, goals, strategies, or constraints. Furthermore, all approaches are external, i.e., they have a dedicated adaptation logic, e.g., in the form of components (e.g., [1,36,80]) or layers (e.g., [35] or [38]).

3.2.3. Reflection approaches

Reflection is the ability of software to examine and possibly modify its structure (*structural reflection*) or behavior (*behavioral reflection*) at runtime [106,107]. The notion of *computational reflection* in programming languages was introduced by Brian Cantwell Smith [108] and is equivalent to the term reflection [109]. Reflection is divided into two activities: *introspection* refers to the observation of an application's own behavior, *intercession* is the reaction on introspection's results [12], i.e., structural, parameter, or context adaptation.

The ability of reflection is an underlying principle for SASs [12,110,111] that enables self-awareness. In [110], the authors describe a reflection reference model which explicitly includes a meta-level sub-system for reflection and a reflection prism describing the properties of reflection. Weyns, Malek, and Andersson describe a formal approach for self-adaptation based on computational reflection [33], which is presented in detail in Section 3.2.9.

Reflection can be used on different levels. *Architectural reflection* (or structural reflection) leads to reflection regarding the software architecture of an application, i.e., its components, interconnections, or data types [12,112]. Approaches for architectural reflection are proposed by various authors (e.g., [112–118]). *Behavioral reflection* refers to reflection regarding the behavior of the software, e.g., algorithms for computation or communication mechanisms [12]. *Reflex* is an approach for integrating partial behavioral reflection in Java [111]. Additionally, approaches that use behavioral models for analyzing changes in the system fall into this category.

Reflective middleware, such as *DynamicTAO* [119] and *Open ORB* [120], can support runtime reconfiguration of a component-based system [110,121]. According to [122], reflective middleware can deal with highly dynamic environments and supports development of flexible, adaptive systems. *CARISMA* integrates the reflective middleware concept and context-awareness [123], which enables the construction of SASs.

In the approaches presented so far, reflection focuses on the software itself, e.g., on components, architecture, or communication behavior. Sawyer et al. propose *requirements-awareness* for SASs: the use of reflection techniques for introspection of requirements at runtime [124]. Introspection and intercession are basic functionalities for adaptation. With introspection, the reason for adaptation can be detected, i.e. introspection includes monitoring and analyzing. Since introspection classically only detects the cause for adaptation after it happened, only reactive adaptation is possible. Intercession determines and controls the adaptation, i.e., controls planning and execution.

3.2.4. Programming paradigms

For parameter and structural adaptation, different programming paradigms can be used. This section presents different programming approaches. These are known from Software Engineering and are not specific to SASs.

In *Component-Based Development* (CBD), also known as *Component-Based Software Engineering* (CBSE) and *Component-Based Programming* (CBP), software components are encapsulated parts of software, that can independently from each other be developed, deployed, and composed [125]. Known examples are *Enterprise JavaBeans*, *COM/DCOM*, or the *CORBA Component Model* [12]. In [12], the authors propose the use of components integrated in a framework that supports late binding for enabling structural adaptation. Therefore, developers must clearly define interfaces, and the adaptation logic must be able to handle the coexistence of components and control the exchange of components. For determining which components must be exchanged, some kind of metric is needed. In literature different metrics can be found, e.g., goal-based [38,82] or model-based [37,72]. Architectural models [37,72] or component models – such as *Fractal* [126] (e.g., used in [127] or [128]), *K-components* [116], or *OpenCom* [121] – define self-contained components. Component-based approaches for SASs are used in many approaches (e.g., [12,37,38,72,82,95,116,128–135]). The use of CBP techniques at runtime for achieving structural adaptation of applications differs from using CBD at design time for developing adaptive software as in [47], where CBP is used for constructing autonomous component units.

In *Aspect-Oriented Programming* (AOP) the program is divided into distinct parts, called *concerns*. The goal is to use generic functionality in different classes (*cross-cutting concerns*). Therefore, logical concerns are separated from the concrete implementation [136]. One example of an AOP language is *AspectJ*, an extension to Java [137]. Whereas dynamic recomposition is often related to cross-cutting concerns as QoS, McKinley et al. propose AOP for enabling separation of concerns, which leads to simplified compositional adaptation [12]. Haesevoets et al. show different possibilities to use AOP techniques for self-adaptation [138]. Other authors highlight single aspects (e.g., [127,139–141]). For SAS development, AOP is often used as an adaptation mechanism, e.g., for structural adaptation, that is controlled by the adaptation logic. Morin et al. present such an approach by combining MDE and aspect-oriented modeling to manage dynamic variability [21,41].

Other authors propose the use of *generative programming* for SASs [12,142]. In generative programming, software is built with the help of high-level descriptions, that are mapped to generic classes, templates, aspects, and components [143].

Whereas the transformation from description to code is unproblematic, vice versa is not supported, what complicates the use of generative programming for SAS implementation [142]. *Adaptive programming* techniques can support the development of SASs [144,145]. In [146], the authors propose an adaptive programming approach which integrates reinforcement learning for learning behavior at run-time, which can only roughly be determined by the developer at design-time. In *Context-oriented Programming* (COP) – known from Pervasive Computing and Ubiquitous Computing – the context is incorporated as a first-class construct in programming languages [147]. This way, the software is able to reason about changes in the context and to appropriately adapt. For a SAS, incorporating only the context is not sufficient but the managed resources must be integrated as well. COP approaches can be found in [148], [147], and [149]. Often, COP is combined with AOP [149].

The presented programming techniques, are mainly used for compositional, reactive adaptation. However, they could be used for all kinds of software adaptation because it only offers an adaptation mechanism that is controlled by the adaptation logic. They can be used for planning and executing of adaptation. Further procedures for monitoring and analyzing have to be included. Often, the techniques are supported by middleware and reasoning is model-based.

3.2.5. Control theory

Control structures are an important part of the adaptation logic of SASs [27]. Control loop engineering poses different challenges: developing reference architectures for control loops, creating a catalog of control loop structures, middleware support for control loop integration, verification and validation techniques to test and evaluate control loops' behavior, and integration of the user [150]. To address these issues, knowledge from control theory is used for the development of SASs. Shaw states that control engineering methodologies should be considered for the software architecture "when the execution of a software system is affected by external disturbances" [151]. Furthermore, she separates the control system from the main process. There are two types of control systems: *open loop* and *closed loop* [151]. The difference is that a closed loop system uses information about monitored output variables to adjust process variables whereas an open loop system adjusts process variables without considering output variables. Closed loop systems can be further divided into feedback control systems and feedforward control systems. Whereas the feedback loop reacts on changes of the output variable, the feedforward loop adjusts process variables by anticipating the effects on the output variable. According to Abdelzaher et al., "control theory provides a formal approach to designing closed loop systems" [152].

Feedback control loops can be adaptive [27]. An adaptive control loop can adjust the controller to respond to changes of the controlled process. For this purpose, there is a second control loop on top of the main control loop. Two standard schemes of adaptive feedback control loops are *Model Identification Adaptive Control* (MIAC) [153] and *Model Reference Adaptive Control* (MRAC) [154]. However, adaptive control comes to its limitations when the control law, e.g., decision functions, stay fixed for their lifetime [155]. Then, the control law can only be updated in a limited way. To overcome this problem, reconfigurable control makes use of learning approaches in order to dynamically adapt the control law [155]. In [156], feedback control loops are represented using actor objects [157]. An actor encapsulates its state and behavior and can only receive, process, and send messages. Each actor is assigned one specific role (sensor, filter, controller, or effector). Adaptive behavior is achieved by connecting corresponding actors. Sensors are responsible for collecting data which is then analyzed by filters. Controllers decide on appropriate actions which are executed by effectors.

Kephart and Chess introduce in [2] the *MAPE-K loop*. There, the MAPE functionality is extended with a shared knowledge repository. Meanwhile, the MAPE-K loop concept is a widely used reference model for SASs (e.g., [19,29,38,60,73,156,158]). As the control loop is separated from the managed element, adaptation control is external. Adaptation time is reactive since the MAPE-K loop is a feedback loop. Similar to the MAPE-K loop, Dobson et al. presented the autonomic control loop [15]. Here, the MAPE functionality is called differently – *collect*, *analyze*, *decide*, and *act* – but has the same responsibilities [15].

In [159], Müller et al. propose to model feedback loops as first class design elements. Hebig et al. take up this issue and present a UML profile that allows for explicitly architecting control loops with component diagrams [160]. There, a control loop is built of four components: controller, process component, sensor, and actuator. Sensor and actuator are used to observe and adjust the managed element. Thus, they use an external approach. In [161], Diao et al. implement a deployable testbed for Autonomic Computing based on different control approaches.

Control engineering for SASs focuses on feedback loops which results in reactive adaptation. Proactive approaches might be feasible using feedforward loops. Adaptation control is mainly external and can take place decentralized. Furthermore, the approaches mentioned above address the whole MAPE functionality. The reader is referred to [162] for further information on designing SASs using control engineering.

3.2.6. Service-oriented approaches

Services are small, encapsulated, and autonomous units of software that fulfill a specific task. In *Service-Oriented Computing* (SOC) such services are used to "support the development of rapid, low-cost, interoperable, evolvable, and massively distributed applications" [163]. The key for SOC is a service-oriented architecture (SOA), which enables finding, use, and connection of services. The service approach can be transferred to SASs.

The naive approach for building a service-oriented SAS is to model the managed resources' functionality as services and use the SOA for communication. The adaptation logic decides, which services should run. Therefore, dynamic exchange of services offers structural adaptation capabilities [164]. The problem is, how to build the adaptation logic and how to enable the exchange of services.

Different frameworks for SASs use service-oriented techniques. In [165], the authors outline the principle of autonomic SOAs. The *MUSIC* framework offers model-driven development of service-based SASs [23,59,60]. The *SASSY* framework is a model-driven, self-architecting framework for SASs [73,166]. Services in a directory, QoS-patterns, and adaptation patterns form the base for SAS construction. In *MetaSelf*, a SOA is controlled by dynamically modifying metadata, policies, and components [167]. Other approaches combine AOP and service-orientation for building SASs [139,141], integrate multi-agent system approaches and SOA [168], use component models which are filled with service implementations at runtime [134,169], apply requirements engineering techniques for building service-based SASs (*CARE* method) [170], or focus on QoS aspects within service-based self-adaptation, such as the *MOSES* framework [171].

Service-oriented approaches focus on structural adaptation through exchange of services or change in the composition of services. Therefore, the managed resources are represented as services and the adaptation logic, either implemented as services or as a layer above, controls the composition of services. Often, models are used for reasoning, which services need to be adjusted (e.g., [60,73,134]). These approaches are reactive, whereas proactive approaches would be feasible, too. The level of adaptation is mostly the application level, the services of the application. The re-composition or exchange of services concerns the planning and executing activities and needs to be accompanied by monitoring and analyzing procedures for building a SAS.

3.2.7. Agent-based approaches

A software agent is a piece of software that fulfills a specific task autonomously and cooperate for common tasks. A *multi-agent system* (MAS) is a system of agents that share common goals and, therefore, communicate and cooperate. For MASs it is claimed, that "they are especially suited to develop software systems that are decentralized, can deal flexibly with dynamic conditions, and are open to system components that come and go" [79]. These properties make agent-based techniques valuable for the development of SASs.

De Wolf and Holvoet discussed several aspects in the construction of self-organizing MAS, such as dynamics and decentralized control in a MAS [172], emergence and self-organization for MASs [173], a proposition for a methodology for engineering self-organizing MASs [174], and design patterns for adaptable MAS [86]. Tesauro et al. present *Unity* [175], a decentralized architecture for Autonomic Computing with autonomic elements. With Unity, autonomic system behavior – such as goal-driven self-assembly, self-healing, and real time self-optimization – can be achieved. Bernon et al. propose the *Adaptive Multi-Agent Systems* (AMAS) theory, in which the system is based on the MAS principle and the agents cooperate for achieving self-organization and adjustments to changes in the environment [176]. Furthermore, they offer tools for the development of SASs based on the AMAS theory. The TOTA middleware can support self-organization in MAS by integrating adaptivity and context-awareness [177].

Other authors integrate a MAS and SOA for building SASs [168], use CBD for designing autonomous agents as components [47], use adaptive object models for designing a MAS [117], or present design patterns for SASs based on agent structures [86,87,178]. Further approaches use agent-related techniques for implementing architectural change [78,79]. The variety of agent-based approaches offers (in theory) both temporal aspects of self-adaptation. Adaptation happens mainly on the application level, but other levels are feasible, too. Most approaches rely on a decentralized adaptation logic, as the MAS should not be controlled by one central unit. The main focus in this category of the MAPE cycle is the planning component.

3.2.8. Nature-inspired approaches

Natural systems are composed of a large number of decentrally organized interacting components [179]. Each component has only limited information based on which they adapt. The resulting overall behavior of the system is then different from the behavior of the individual components, called *emergence* [27,173]. *Nature-inspired systems* may bring certain benefits such as spatiality, self-adaptability, and openness [180]. Furthermore, they can be categorized into four key metaphors: *biological*, *physical*, *chemical*, and *social*. In this section, we present some examples for using nature-inspired mechanisms in SAS. Further overviews can be found in literature (e.g., [11,180]).

Biological approaches in computer science have emerged with the study of *collective behavior* in natural MAS by Parunak [181]. Biological mechanisms, such as *flocking* [182–184], *foraging* [182,185,186], *nest building* [182], *molding* [182], *local inhibition* [179,187,188], *lateral inhibition* [189], *chemotaxis* [179,190], *embryogenesis* [182], *morphogen gradient* [179], *local monitoring* [179], *quorum sensing* [179,182,191], *consensus* [179], *firefly synchronization* [192], *stigmergy* [181,182,190], *web weaving* [182,193], *brood sorting* [182], and the *human immune system* [194] or *human autonomous nervous system* [2, 182], respectively, have been applied in self-organizing systems and can be transferred to SASs.

Physical approaches, so far, mainly focus on the metaphor of *potential fields* pioneered by Kathib who employed them for obstacle avoidance in path planning in [195]. Reif and Wang propose a distributed method to control autonomous robots based on potential fields [196]. In [197], the authors apply potential fields for exploration and foraging tasks of autonomous robots. Weyns et al. employ potential fields for adaptive task assignment in MAS [198]. In [177], Mamei and Zambonelli present the *TOTA* approach, a framework for pervasive and mobile computing approaches. Here, communication takes place with the help of potential fields.

Chemical approaches focus on chemical reactions. The *Higher Order Chemical Language* (HOCL) serves as a mean for programming service composition based on chemical reactions and enables late binding of components at runtime [199]. Viroli and Casadei propose *biochemical tuple spaces*, a coordination model for self-organizing systems based on chemical reactions [200].

Social approaches concentrate on *market* and *auction* mechanisms, as well as *social norms* (e.g., [201–203]). As an example, in [203], coordination in MAS is based on social conventions.

Most investigated approaches focus on the planning component. Thus, adequate procedures for monitoring, analyzing, and executing need to be included.

3.2.9. Formal modeling and verification approaches

Self-* properties and dynamics make it hard to prove the correctness of SASs [204,205]. However, a SAS requires behavioral and structural guarantees [26], especially in safety-critical domains, e.g., traffic light systems [206].

As an approach to reduce the complexity of the verification task, Güdemann, Ortmeier, and Reif introduced the *Restore Invariant Approach* (RIA) [204,205,207] based on transition automata for representing system components and temporal logic to express functional properties. In this context, an invariant is a formula that divides all possible allocations of functional properties into two sets: the functionality can be provided, i.e., the invariant holds, and the functionality cannot be provided, i.e., the invariant is violated [207]. The former set is also called corridor [205]. As long as the invariant holds, the system finds itself in the production phase. As soon as it gets violated, the system changes to the reconfiguration phase, in which it restores the invariant and then switches back to production. So far, the adaptation control of the RIA has been centralized and external, but a decentralized control unit is possible, too [205]. RIA is reactive and adapts only to changes in the (technical) resources by altering the structure.

Weyns, Malek and Andersson suggest the *formal reference model for self-adaptation* (FORMS) [33]. They use the *Z* notation in order to define a SAS, its relationship with the environment, and self-adaptation. Adaptation control takes place centralized and external. King et al. propose a self-testing framework for Autonomic Computing systems that validates structural and behavioral change requests at runtime in order to avoid costly system failures [208]. When a change request comes in, it is forwarded to a special testing autonomic manager which validates the request. This approach uses an external adaptation control. Smith and Sanders [209] present an incremental top-down approach to formally develop self-organizing systems, which is built on the three "scales of observation" introduced in [210]. The *Autonomic System Specification Language* is a framework for formally specifying and generating autonomic systems [211]. Cordy et al. use a model checking technique based on transition systems in order to verify dynamic software product lines [58]. Filieri and Tamburrelli focus on probabilistic runtime model checking using Discrete Time Markov Chains [212]. Priesterjahn et al. present a formal model for the timed hazard analysis of self-healing systems [213].

This category mainly addresses the analyzing part of the MAPE-loop. Further information on formal methods in SASs can be found in [214].

3.2.10. Learning approaches

Learning in a SAS is tightly coupled to self-optimization. A SAS continually optimizes its structure, parameters, or algorithms in order to become more efficient with regard to performance or cost [2]. Therefore, learning in a SAS focuses on *structural optimization* (e.g., [42,175,215–218]). In *Unity* [175], self-optimization works with the help of a central resource arbiter that computes optimal allocations of resources based on a utility function, which is refined by learning. Elkhodary et al. present a feature-oriented architecture for SASs [42,215]. Each goal has a utility function that is used for optimization. Fisch et al. propose a collaborative learning approach by exchanging knowledge [216]. Rules represent knowledge. When an agent discovers a novel useful rule, it broadcasts the rule, so that other agents can make use of it. Dowling et al. present *collaborative reinforcement learning* for decentralized coordinated self-adaptive components [217]. Collaborative reinforcement learning makes collective adaptation possible by collaborative feedback [219]. Parra et al. employ a constraint-based approach to optimize context-aware adaptations [218].

Prothmann et al. use *evolutionary programming* for learning [206]. In an organic traffic control scenario, they introduce an *observer/controller* architecture with on- and offline learning. A learning classifier system, responsible for online learning, is combined with an offline learning evolutionary algorithm. Tomforde et al. extend the observer/controller architecture with a decentralized collaboration mechanism [220] and discuss several distribution possibilities of the architecture [221]. Cakar et al. use a generic observer/controller architecture, proposed in [158], as basis for their learning approach [222]. Brockmann et al. developed a framework for controlled self-optimization in modular system architectures based on goals with an online learning approach based on machine learning [223].

Other works address *parameter optimization*. [224] gives an overview on evolutionary algorithms with regard to parameter optimization. Abdelwahed, Kandasamy, and Neema present a control framework for self-managing distributed systems [225] and apply model learning procedures online for managing varying environmental and operating conditions.

Algorithmic optimization aims at optimizing how a goal is achieved, hence altering the algorithms. Oreizy et al. mention evolutionary programming and AI-based learning techniques in order to generate new algorithms to encounter uncertain changes [1].

Learning approaches mainly focus on the planning component. Adequate procedures for monitoring, analyzing and executing have to be added. Further, learning can be used for adaptation of the adaptation logic itself.

3.2.11. Requirements engineering for self-adaptive systems

For SASs, a specialized form of requirements engineering (RE) is needed. Berry, Cheng, and Zhang defined four levels of RE for dynamic adaptive systems: (i) Level 1: general definition of the system and its reaction by developers, (ii) Level 2:

the system does RE at runtime for achieving adaptation, (iii) Level 3: decision of developers about adaptation mechanisms, and (iv) Level 4: research regarding adaptation mechanisms [226]. In this work, we focus on the dynamic aspects of RE at runtime (level 2), as it enables adaptation of the SAS.

The dynamic nature of a SAS potentially results in uncertainty at runtime [124], due to a lack of information at design time [13]. According to Whittle et al., changes in a system's environment, such as "sensor failures, noisy networks, malicious threats, and unexpected (human) input" [227] are the main reasons for uncertainty. They further propose to relax non-critical system goals in emergency situations in order to achieve high-level goals. Different authors propose to include runtime capabilities for RE. Bencomo, Whittle, and Sawyer propose to model requirements as runtime entities and to integrate reflection capabilities for requirements [228]. *FLAGS* is a goal model based approach for modeling requirements at runtime [229]. In [230], the authors present a runtime infrastructure that is able to manage requirements at runtime based on modeling the requirements with FLAGS. In his doctoral thesis, Souza proposes two new classes of requirements for SASs: *awareness requirements* prescribe indicators of convergence in requirements and *evolution requirements* represent strategies for adaptation addressing changes in the requirements models themselves [231]. Furthermore, he describes an approach for requirements-based system adaptation.

Different requirement languages for SASs can be found in literature. In [232], the authors present RELAX, which explicitly allows to specify and deal with uncertainty. As *RELAX*, still most notations for specifying requirements are based on natural language prose [13]. However, there exist also other approaches based on goal models, agents, or scenarios. Goal-based approaches are for example *KAOS* [233], *i** [234], *LoREM* [48], *FLAGS* [229], and *CARE* [235]. However, only CARE explicitly supports uncertainty. *Tropos4AS* [236] is an agent-based design framework for modeling SAS requirements based on Tropos [237]. No explicit support for uncertainty is given. Scenario-based notations, such as *live sequence charts* [238], also do not explicitly support uncertainty.

Since requirements can influence all dimensions of our taxonomy, runtime capabilities for managing requirements are important for reasoning about adaptation. Requirements Engineering approaches usually address the whole MAPE cycle. As requirements are a kind of goal the SAS should fulfill, RE approaches are mainly goal-based.

3.2.12. Further approaches

There are further approaches, which cannot be categorized into one of the categories presented above.

In *task-based adaptation*, the system determines suitable adaptation policies based on the users' task characteristics [84]. The adaptation mechanism is not defined, it only affects reasoning. Therefore, the approach cannot be classified into one of the mentioned categories. It can be seen as utility-based adaptation with the objective to maximize the system utility for the users in supporting their tasks. Task-based self-adaptation is used within the *Aura* project [239].

Some approaches use middleware-centric adaptation for achieving structural adaptation. Sadjadi and McKinley present an overview on *adaptive middleware* [22]. In [21], the authors build their approach on top of adaptive middleware. Hallsteinsen, Floch, and Stav proposed an approach for building SASs by using generic middleware for handling the adaptation [115].

Additionally, some authors propose processes and modeling dimensions for simplifying the engineering of SASs. In the *FESAS* project, we present a framework for using reusable components of a component library for adaptation logic development [240]. As outcomes of Dagstuhl Seminars about software engineering (SE) for SASs [13,14], *modeling dimensions* (*goals*, *change*, *mechanisms*, and *effects*) [241], a *design space* for SASs with five principal clusters of design decisions (*observation*, *representation*, *control*, *identification*, and *enacting adaptation*) [25], and *SE processes* for moving activities from design to runtime when engineering SASs [242] are proposed. Zhang and Cheng proposed a model-based process for the construction of adaptation models, automatic code generation from the models for building adaptive programs, and verification and validation of the models [74]. In [243], the authors propose a guideline with 14 tasks for the construction of self-organizing systems. Lightstone presents a guideline for Autonomic Computing application development focusing on integration of the user [244]. Different authors propose the use of design patterns for developing SASs (e.g., [19,85–88,178,240,245]).

3.3. Conclusion

In this section, we highlighted implementation issues for the adaptation logic of a SAS. The adaptation logic can be integrated into the resources or dedicated. Different metrics such as models, goals, rules/policies, or utility influence the adaptation decision. Furthermore, developers must specify the degree of decentralization, distribution of the adaptation logic elements, and interaction patterns.

Additionally, we discussed different approaches that can be used for building a SAS's adaptation logic and we classified the approaches into model-based, architecture-based, reflection, programming paradigms, control theory, service-oriented, agent-based, nature-inspired, formal modeling and verification, learning, and requirements-oriented. In contrast, Macías-Escrivá et al. differentiate approaches for building SASs into *main approaches* (external control mechanisms, CBSE, model-driven, nature-inspired computing, MAS, feedback systems), *global tools and methods* (models, simulation, architecture, frameworks), and *specific tools and methods* (feedback control loops, decision-making, RE) [11]. Whereas the categories are similar – simulation is part of the learning approaches we presented – we did not divide the approaches in their application for SAS development. Nevertheless, we acknowledge that our classification is not interception-free, e.g., in some architecture-based approaches, models are used for representing the architecture and CBSE for the structural adaptation

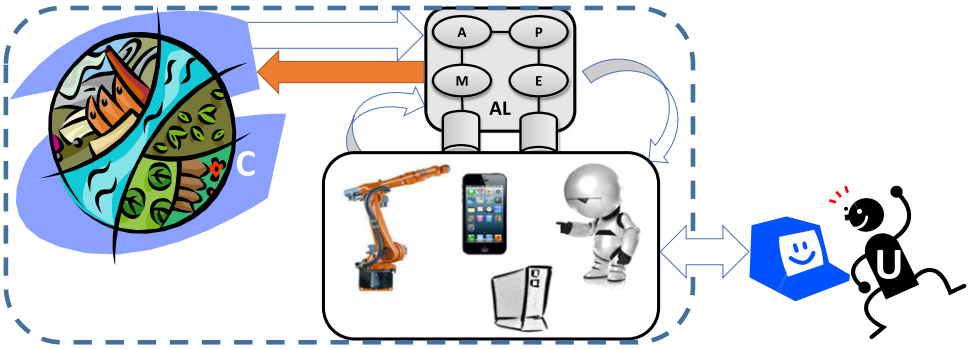


Fig. 4. A context-altering self-adaptive system (AL = Adaptation Logic, MR = Managed Resources, U = User(s), C = Context).

of the architecture. Furthermore, the categories can be combined for developing a SAS, e.g., model-based analyzing with component-based adaptation.

We mapped the approaches to our taxonomy of self-adaptation as well as the MAPE activities. By doing this, we identified that most approaches for development of SAS are reactive and do not explicitly include the influence of actuators of the managed resources on the context, i.e., context adaptation. This can lead to serious problems and, in the worst case, end up in a cycle where the adaptation logic decides to adapt, the adaptation leads to an unforeseen change in the context, which results in further adaptation.

For that reason, in the next section, we extend the definition of a SAS to context-altering SAS based on the mapping of our taxonomy for self-adaptation to the survey of SAS approaches.

4. A new perspective on self-adaptive systems and research challenges

The survey of approaches showed that the inclusion of context in most approaches is not sufficient. Whereas most approaches monitor the context, explicit alteration of context is not included in many approaches and the environment remains uncontrollable for the adaptation logic [242]. This can lead to undesired adaptation results. Therefore, we propose explicit integration of context alteration into the reasoning process. In this section, we include context adaptation for SASs and extend the definition of a SAS to a context-altering SAS. Furthermore, we will present different research issues related to the implementation of SASs.

A SAS is composed of managed resources and the adaptation logic [19]. The surrounding context is used as input for the analyzing process. For a SAS, the context is not controlled by the adaptation logic [242], i.e., it is not included in the planning activity. So, context alteration is not explicitly included in the reasoning process. Furthermore, the missing control of the context can lead to undesired change in the context through the managed resources' actuators and result in interferences [20]. In our understanding of SASs, the possibility to change the context is explicitly integrated. Derived from the taxonomy presented in Section 2 – there, context is a level where adaptation can happen – we propose the explicit integration of context for planning. Modeling context and context-altering capabilities as a construct enables reasoning about it and enables the control of context adaptation by the adaptation logic. So, the adaptation logic explicitly manages the context. Therefore, we extend the definition of a SAS and integrate context-adapting and context-altering, respectively:

> "*A Self-Adaptive System is able to modify its behavior* ***or its environment*** *in response to changes in its operating environment. The operating environment includes anything observable, such as end-user input, hardware devices, surrounding context, or program instrumentation.*"

This new perspective on SASs leads to a new definition of the system border of a SAS (compared to Section 3.1). For a context-altering SAS, the system is able to change its behavior – through parameter or structural adaptation of the managed resources – and the context. This can result in additional dynamics for the adaptation compared to a common SAS. Therefore, we include control of the context into the SAS and extend the SAS to a triple $SAS = (AL, MR, C)$ with the technical system – composed of the adaptation logic AL and the managed resources MR – and the context C. Different context variables, e.g., temperature, noise, or light level, define the context. They are influenced via actuators of the managed resources, controlled by the adaptation logic. Therefore, the context is modeled as $C = \{c_1, \ldots, c_n\}$ where each c_i symbolizes a context variable, e.g., the temperature. The user is not integrated because the user should not be adapted. Fig. 4 shows the principle layout of our view on SASs. The dashed line shows the system border. This new view on SASs leads to different issues.

Context adaptation. The inclusion of context adaptation controlled by the adaptation logic leads to research issues that are not relevant in the traditional view of SAS, where the environment is not controlled by the adaptation logic [242]. Different issues for context reasoning, e.g., unreliability of sensor information, are present [34]. These issues are valid for context-altering SASs, as well as for analyzing activities of common SASs. So far, context is mainly included for analysis purposes in SASs. For a context-altering SAS, the context must be integrated into planning, i.e., information about the context-altering capabilities of the technical resources is necessary for planning. The reasoning part of the adaptation

logic must know all adaptation possibilities [246]. This implies that the actuators of managed resources for changing context variables must be integrated in the reasoning process and the context variables must be modeled accordingly [246]. Therefore, existing context modeling and reasoning techniques (e.g., see [34]) must be integrated in approaches for SAS for building context-altering SAS.

Decentralization of the adaptation logic. As already mentioned in Section 3.1.3 and in the taxonomy in Section 2.5, one issue of implementing SASs is the decentralization of the adaptation logic [26]. Different authors already addressed issues of a decentralized adaptation logic (e.g., [19,29,86,160,172]). Nevertheless, there are open challenges: partial shared knowledge, coordination in the MAPE functions, uncertainty, conflicting goals, system-wide assurance and verification, overhead for coordination, and missing standardization [19,29]. Research needs to tackle these challenges. Furthermore, we propose systematic research on factors that influence the degree of decentralization of the adaptation logic, as well as which interaction pattern should be used [240].

The inclusion of context alteration introduces further challenges. In a distributed system, different elements share the context. Having a decentralized adaptation logic, the problem of uncontrollable context adaptation can still happen, even if context adaptation is included in the reasoning process, e.g., if two subsystems A and B both have an adaptation logic that do not interact, but an adaptation of subsystem A influences the context in a negative way for subsystem B. Therefore, decentralization of the adaptation logic can lead to a fragmentation of context(-altering) information. This must be addressed, e.g., by exchange of information about context-altering capabilities between the different adaptation logic elements or context needs, respectively or with frameworks, such as the COMITY framework [20]. Furthermore, distributed monitoring of the context can be an issue as different parts of a distributed system can have different types of sensors and gain different information. An approach can be the implementation of a context broker, that distributes the context information within the adaptation logic, e.g., implemented similar to the information sharing pattern [19].

Proactive adaptation. Proactive adaptation is one of the temporal dimensions for adaptation (see Section 2.1). The aim is to adapt before it becomes necessary based on prediction. From the user's point of view, this is preferable, as it reduces interruptions and adaptations can be optimized for a sequence of events [10,246]. Further, proactive adaptation includes context adaptation via actuators in order to avoid unwanted situations. However, proactive adaptation has several remaining challenges, especially in scenarios of multiple applications or systems that share context. First, it is very complex to develop without suitable frameworks, which do not exist at this point. Second, it is highly dependent on the correctness of the predictions, as faulty predictions can cause suboptimal adaptations. The major challenges here are predicting the time of an event with high enough accuracy, as well as predicting user behavior and rare events. Third, as mentioned before, adaptations from independent sources can cause oscillating adaptations. This problem is especially prevalent for proactive adaptation, as additional adaptations would cancel its benefits. Hence, coordination is very important for proactive adaptation. As the survey has shown, so far most approaches for adaptive systems focus on reactive adaptation. Integration of proactive adaptation into SASs is a challenging task that needs further research.

Further challenges for SAS development. Further challenges can be found in literature. Integration of the user in the adaptation process, i.e. human-in-the-loop integration, gets increasing importance within the community. Central issues for runtime models are the integration of different model representations, how to use known techniques from design models at runtime, reversible model transformations, or modeling the relation between runtime models and the represented architecture [69,247]. Due to uncertainty, aspects of requirements engineering must be performed at run-time [124,248]. Verification and validation (V and V) techniques need to handle the dynamics of requirements as well as uncertainty [26, 150]. SE processes have to be adapted for SAS development. Especially the shift of activities from development time to runtime is challenging [242]. This has been addressed in the second Dagstuhl seminar on SE for SASs [13]. Reusability of processes as well as components is addressed in the FESAS project [240]. A detailed presentation of these challenges would go beyond the scope of this paper.

5. Conclusion

In this paper, we presented a taxonomy for self-adaptation, a survey on engineering approaches for SASs, and a new perspective on SASs, the *context-altering SAS*. Based on literature research and combination of existing surveys, the taxonomy describes self-adaptation in the dimensions *time*, *reason*, *technique*, *level*, and *adaptation control*. The adaptation logic, which controls the adaptation, must be appropriately designed. Developers need to define the approach, adaptation decision criteria, and degree of decentralization, as well as the integration of the self-adaptation dimensions for monitoring and reasoning. For building SASs, different approaches can be found in literature. We classified the approaches in different categories: *model-based*, *architecture-based*, *reflection*, *programming paradigms*, *control theory*, *service-oriented*, *agent-based*, *nature-inspired*, *formal modeling and verification*, *learning*, and *requirements-oriented*. These categories can overlap, e.g., some architecture-based approaches use architectural models for reasoning and CBD for structural adaptation. The taxonomy showed that most approaches focus on reactive adaptation without the integration of context adaptation. This motivates our new view on SASs. Whereas in a common SAS the context is only monitored, for a *context-altering SAS* we assume that the adaptation logic is aware of actuators and uses them for adapting the context. The new context-altering capabilities introduce additional challenges, such as the integration of context alteration into the reasoning process. These challenges, such as integration of the context for analyzing and planning, decentralization of context monitoring and reasoning on

Table A.3
Overview of the approaches of the survey.

	Approach	MAPE	Time	Reason	Level	Tec	Adaptation control		
							Appr	DC	DDec
Model-based	(Dynamic) software product lines [21,40,43,52,52–58,218]	A/P	React	Ctx/TR/U	App/Comm/TR	Par/Str	Ext	Goals, Policies, Utility	–
	MUSIC [23,59,60]	All	React	Ctx/TR	App/Comm	Par/Str	Ext	Models, Utility	Hyb/Cen
	Meta-models/megamodels [62–64]	All	–	–	–	–	Ext	Models	–
Architecture-based	Rainbow framework [35]	All	React	TR	App/TR	Par/Str	Ext	Models, Policies	All
	3L approach [38]	All	React	Ctx/TR	App/TR	Par/Str	Ext	Goals	All
	Architectural run-time configuration manager [80]	All	React	TR	App/TR	Par/Str	Ext	Policies, Models	–
	Archstudio [1,36]	All	React	Ctx/TR	App/TR	Par/Str	Ext	Models	All
Reflection-based	Introspection [12]	M/A	–	Ctx/TR/U	–	–	Both	–	–
	Intercession [12]	P/E	–	–	All	All	Both	–	–
	Reflection reference model [110]	All	React	Ctx/TR	App/TR	Par/Str	Ext	(Meta) Models	–
	FORMS	See category "Formal Modelling and Verification Approaches"							
	Reflex [111]	All	React	TR	App/TR	Par/Str	Ext	(Meta) Models	–
	Reflective middleware [119,120,123]	P/E	–	–	Sys/Comm	Par/Str	–	–	–
	CARISMA [123]	All	React	Ctx/TR	App/TR	Par	Ext	Policies	Dec/Hyb
Programming paradigms	Component-based SE [12,37,38,72,82,95,116,128–135]	P/E	–	–	App/TR	Str	Ext	All	All
	Aspect-oriented programming [21,41,137,138]	P/E	–	–	App/TR	Str	Ext	All	All
	Generative programming [12,142,142]	P/E	–	–	App/TR	Par/Str	Ext	All	All
	Adaptive programming [144–146]	Addresses various aspects in the development of SAS							
	Context-oriented programming [147–149]	A/P/E	React	Ctx/TR	App/TR/Ctx	All	–	Models, Goals	–
Control theory	Autonomic computing [2]	All	React	TR/Ctx	Sys	Par	Ext	Policies, Goals	Dec
	Autonomic communication [15]	All	React	TR/Ctx	Comm	Par/Ctx	–	–	–
	Control loop patterns [159]	All	React	Ctx	Sys/TR	Par	–	Models	–
	Control loop UML profile [160]	All	React	TR	Comm	Par/Str	Ext	–	Dec
	Control theory foundation [161]	All	React	TR/Ctx	Sys	Par	Ext	Models	–
Service-oriented	MUSIC	See category "Model-based Approaches"							
	SASSY framework [73,166]	All	React	Ctx/TR	App	Str	Ext	Goals, Utility	Dec/Hyb
	MetaSelf [167]	All	React	Ctx/TR	App	Str	Ext	Policies, Rules	All
	Aspect-oriented and service-oriented computing [139,141]	P/E	–	–	App	Str	Ext	–	–
	Agent systems and SOA	See category "Agent-based Approaches"							
	Component models and services [134,169]	All	React	TR	App	Str	Ext	–	–
	RE and service-orientation [170]	A/P/E	React	Ctx/TR/U	App	Str	Ext	Goals, Utility	Cen
	MOSES framework [171]	All	React	TR	App	Str	Ext	Goals, Utility	Cen
Agent-based	MOCAS [47]	All	React	Ctx	Sys	Par	–	Policies	Dec
	Design patterns [86]	P	React	Ctx	App	Par	–	Models	Dec
	Agent-based modeling, dynamical systems analysis, and decentralized control [172]	All	React	Ctx	App	Par	–	Models	Dec/Hyb
	Unity	See category "Learning Approaches"							

(continued on next page)

context, and proactivity of adaptation should be addressed within the SEAMS community for easier development of SAS that uses the potential of context adaptation.

Acknowledgments

This work was funded in part by the European Community in the framework of the VITAL FP7 project (Virtualized programmable InTerfAces for smart, secure and cost-effective IoT depLoyments in smart cities) under contract number FP7-ICT-608662. This work is supported by UBICITEC e.V. (European Center for Ubiquitous Technologies and Smart Cities) and

Table A.3 (*continued*)

	Approach	MAPE	Time	Reason	Level	Tec	Adaptation control		
							Appr	DC	DDec
Nature-inspired	Optimization [183,184,186]	P	React	TR/Ctx	Sys	Par/Str	–	Utility	Dec
	Autonomic computing for pervasive ICT [187]	P	React	TR	App/Sys	Par/Str	–	–	Dec
	Frequency planning [188,189]	P	React	TR	App	Par	–	–	Dec
	Data harvesting [190]	P	React	Ctx	App	Par	–	Utility	Dec
	Network synchronicity [192]	P	React	TR	Sys	Par	–	–	Dec
	Region detection [193]	P	React	Ctx	App	Par	–	–	Dec
	Immune system [194]	P	React	Ctx	Sys	Par	–	–	Dec/Hyb
	Obstacle avoidance [195]	P	React	Ctx	App	Par	Int	Utility	–
	Potential fields [196,197]	P	React	Ctx	App	Par	–	–	Dec/Hyb
	Task assignment [198]	P	React	Ctx	App	Par	–	Utility	Dec
	Ecosystem framework [201]	–	React	Ctx	App	Par	–	Utility	Dec
	RAPPID [202]	P	React	Ctx	App	Par	–	–	Dec
	Social conventions [203]	P	React	Ctx	App	Par	–	–	Dec
Formal methods	FORMS [33]	All	React	TR/Ctx	Sys/TR	Par/Str	Ext	Models	–
	Restore-invariant approach [204,205,207]	A	React	TR	TR	Str	–	–	–
	Self-testing framework [208]	A	React	TR/Ctx	Sys	Par	Ext	–	Dec
	Timed hazard analysis [213]	A/P	React	TR	TR	Str	Ext	Models	–
Learning	FUSION [42,215]	P	React	TR/Ctx	App/Sys	Par/Str	–	Goals, Utility	–
	Unity [175]	P	React	TR	Sys	Par	–	Goals, Utility	Dec
	Evolutionary algorithms [206,220–222,224]	All	React	Ctx	App	Par	Ext	Utility	All
	Control-based framework [225]	P	React	Ctx	Sys	Par	Ext	Models	Cen
RE-based	LoREM [48]	All	React	–	–	Par	–	Models	–
	Requirements@Runtime [230]	A/P	React	U	App	Par/Str	–	Goals	–
	Zanshin [231]	All	React	Ctx	App	Par	–	Goals	–
Further	Task-based adaptation [84,239]	A/P	–	–	App/TR	–	–	Goals, Utility	–
	Middleware-centric adaptation [21,22,115]	All	React	Ctx/TR	App/TR	Par/Str	Ext	All	Dec/Hyb

GAMBAS (Generic Adaptive Middleware for Behavior-driven Autonomous Services) funded by the European Commission under FP7 with contract FP7-2011-7-287661.

Appendix. Overview of the approaches presented in the survey

Table A.3 shows an overview of the approaches that we presented in Section 3.2 and shows their relation to MAPE activities, the taxonomy of Section 2, and the issues for implementing an adaptation logic of Section 3.1. Within the table, we used the following abbreviations:

React = Reactive; **Ctx** = Context; **TR** = Technical resources; **U** = User(s); **App** = Application; **Comm** = Communication; **Sys** = System Software; **Tec** = Technique; **Par** = Parameter; **Str** = Structure; **Appr** = Approach; **Ext** = External; **Int** = Internal; **DC** = Decision Criteria; **DDec** = Degree of Decentralization; **Hyb** = Hybrid; **Dec** = Decentralized; **Cen** = Centralized.

If a cell is marked with "–", this means that the approach does not have a specific requirement or that no further information is given.

References

[1] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D.S. Rosenblum, A.L. Wolf, An architecture-based approach to self-adaptive software, IEEE Intell. Syst. 14 (3) (1999) 54–62.
[2] J.O. Kephart, D.M. Chess, The vision of autonomic computing, IEEE Comput. 36 (1) (2003) 41–50.
[3] C. Müller-Schloer, H. Schmeck, T. Ungerer (Eds.), Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011.
[4] M.C. Huebscher, J.A. McCann, A survey of autonomic computing—degrees, models, and applications, ACM Comput. Surv. 40 (3) (2008) 1–28.
[5] M. Salehie, L. Tahvildari, Self-adaptive software: landscape & research challenges, ACM Trans. Auton. Adapt. Syst. 4 (2) (2009) Art. 14.
[6] M. Hinchey, R. Sterritt, Self-managing software, IEEE Comput. 39 (2) (2006) 107–109.
[7] B. Schilit, N. Adams, R. Want, Context-aware computing applications, in: Proc. WMCSA, IEEE, 1994, pp. 85–90.
[8] A.K. Dey, Understanding and using context, Pers. Ubiquitous Comput. 5 (1) (2001) 4–7.
[9] M. Rohr, S. Giesecke, W. Hasselbring, M. Hiel, W.-J. van den Heuvel, H. Weigand, A classification scheme for self-adaptation research, in: Proc. SOAS, 2006, p. 5.
[10] M. Handte, G. Schiele, V. Matjuntke, C. Becker, P.J. Marrón, 3PC: system support for adaptive peer-to-peer pervasive computing, ACM Trans. Auton. Adapt. Syst. 7 (1) (2012) Art. 10.
[11] F.D. Macías-Escrivá, R. Haber, R. del Toro, V. Hernandez, Self-adaptive systems: a survey of current approaches, research challenges and applications, Expert Syst. Appl. 40 (2013) 7267–7279.
[12] P. McKinley, S. Sadjadi, E. Kasten, B.H.C. Cheng, Composing adaptive software, IEEE Comput. 37 (7) (2004) 56–64.

[13] B.H.C. Cheng, R. de Lemos, P. Inverardi, J. Magee (Eds.), Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009.
[14] R. de Lemos, H. Giese, H.A. Müller, M. Shaw (Eds.), Software Engineering for Self-Adaptive Systems II, in: LNCS, vol. 7475, Springer, 2013.
[15] S. Dobson, F. Zambonelli, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, A survey of autonomic communications, ACM Trans. Auton. Adapt. Syst. 1 (2) (2006) 223–259.
[16] R. Laddaga, Active software, in: Self-Adaptive Software, in: LNCS, vol. 1936, Springer, 2001, pp. 11–26.
[17] J. Buckley, T. Mens, M. Zenger, A. Rashid, G. Kniesel, Towards a taxonomy of software change, J. Softw. Maint. Evol.: Res. Pract. 17 (5) (2005) 309–332.
[18] A. Schmidt, K. van Laerhoven, How to build smart appliances? IEEE Pers. Commun. 8 (4) (2001) 66–71.
[19] D. Weyns, B.R. Schmerl, V. Grassi, S. Malek, R. Mirandola, C. Prehofer, J. Wuttke, J. Andersson, H. Giese, K.M. Göschka, On patterns for decentralized control in self-adaptive systems, in: Software Engineering for Self-Adaptive Systems II, in: LNCS, vol. 7475, Springer, 2013, pp. 76–107.
[20] V. Majuntke, S. VanSyckel, D. Schäfer, C. Krupitzer, G. Schiele, C. Becker, COMITY: coordinated application adaptation in multi-platform pervasive systems, in: Proc. PerCom, IEEE, 2013, pp. 11–19.
[21] B. Morin, O. Barais, G. Nain, J.-M. Jézéquel, Taming dynamically adaptive systems using models and aspects, in: Proc. ICSE, IEEE, 2009, pp. 122–132.
[22] S.M. Sadjadi, P. McKinley, A survey of adaptive middleware, Tech. Rep., Michgan State University, East Lansing, Michigan, USA, 2003.
[23] J. Floch, S. Hallsteinsen, E. Stav, F. Eliassen, K. Lund, E. Gjorven, Using architecture models for runtime adaptability, IEEE Softw. 23 (2) (2006) 62–70.
[24] P. Lalanda, J.A. McCann, A. Diaconescu, Autonomic Computing, Springer, 2013.
[25] Y. Brun, R. Desmarais, K. Geihs, M. Litoiu, A. Lopes, M. Shaw, M. Smit, A design space for self-adaptive systems, in: Software Engineering for Self-Adaptive Systems II, in: LNCS, vol. 7475, Springer, 2013, pp. 33–50.
[26] R. de Lemos, H. Giese, H.A. Müller, M. Shaw, J. Andersson, et al., Software engineering for self-adaptive systems: a second research roadmap, in: Software Engineering for Self-Adaptive Systems II, in: LNCS, vol. 7475, Springer, 2013, pp. 1–32.
[27] Y. Brun, G. Di Marzo Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H.A. Müller, M. Pezzè, M. Shaw, Engineering self-adaptive systems through feedback loops, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 48–70.
[28] O. Babaoglu, H.E. Shrobe, Foreword from the general co-chairs, in: Proc. SASO, IEEE, 2007, pp. ix–x.
[29] D. Weyns, S. Malek, J. Andersson, On decentralized self-adaptation: lessons from the trenches and challenges for the future, in: Proc. SEAMS, ACM, 2010, pp. 84–93.
[30] IBM Corp., An architectural blueprint for autonomic computing, Tech. Rep., IBM, Hawthorne, NY, USA, 2005.
[31] D.C. Schmidt, Guest editor's introduction: model-driven engineering, IEEE Comput. 39 (2) (2006) 25–31.
[32] R. France, B. Rumpe, Model-driven development of complex software: a research roadmap, in: Proc FOSE, IEEE, 2007, pp. 37–54.
[33] D. Weyns, S. Malek, J. Andersson, FORMS: a formal reference model for self-adaptation, in: Proc. ICAC, ACM, 2010, pp. 205–214.
[34] C. Bettini, O. Brdiczka, K. Henricksen, J. Indulska, D. Nicklas, A. Ranganathan, D. Riboni, A survey of context modelling and reasoning techniques, Pervasive Mob. Comput. 6 (2) (2010) 161–180.
[35] D. Garlan, S.-W. Cheng, A.-C. Huang, B.R. Schmerl, P. Steenkiste, Rainbow: architecture-based self-adaptation with reusable infrastructure, IEEE Comput. 37 (10) (2004) 46–54.
[36] P. Oreizy, N. Medvidovic, R.N. Taylor, Architecture-based runtime software evolution, in: Proc. ICSE, IEEE, 1998, pp. 177–186.
[37] N. Bencomo, P. Grace, C. Flores, D. Hughes, G. Blair, Genie: supporting the model driven development of reflective, component-based adaptive systems, in: Proc. ICSE, ACM, 2008, pp. 811–814.
[38] J. Kramer, J. Magee, Self-managed systems: an architectural challenge, in: Proc. FOSE, 2007, pp. 259–268.
[39] D. Batory, Feature models, grammars, and propositional formulas, in: Software Product Lines, in: LNCS, vol. 3714, Springer, 2005, pp. 7–20.
[40] C. Cetina, P. Giner, J. Fons, V. Pelechano, Autonomic computing through reuse of variability models at runtime: the case of smart homes, IEEE Comput. 42 (10) (2009) 37–43.
[41] B. Morin, F. Fleurey, N. Bencomo, J.-M. Jézéquel, A. Solberg, V. Dehlen, G. Blair, An aspect-oriented and model-driven approach for managing dynamic variability, in: Model Driven Engineering Languages and Systems, in: LNCS, vol. 5301, Springer, 2008, pp. 782–796.
[42] A. Elkhodary, N. Esfahani, S. Malek, FUSION: a framework for engineering self-tuning self-adaptive software systems, in: Proc. FSE, ACM, 2010, pp. 7–16.
[43] M. Acher, P. Collet, F. Fleurey, P. Lahire, S. Moisan, J.-P. Rigault, Modeling context and dynamic adaptations with feature models, in: Proc. MRT, vol. 509, CEUR-WS.org, 2009, pp. 89–98.
[44] K.C. Kang, H. Lee, Variability modeling, in: Systems and Software Variability Management, Springer, 2013, pp. 25–42.
[45] C. Ghezzi, A. Mocci, M. Sangiorgio, Runtime monitoring of functional component changes with behavior models, in: Models in Software Engineering, in: LNCS, vol. 7167, Springer, 2012, pp. 152–166.
[46] C. Ghezzi, A. Mocci, M. Monga, Synthesizing intensional behavior models by graph transformation, in: Proc. ICSE, IEEE, 2009, pp. 430–440.
[47] C. Ballagny, N. Hameurlain, F. Barbier, MOCAS: a state-based component model for self-adaptation, in: Proc. SASO, IEEE, 2009, pp. 206–215.
[48] H. Goldsby, P. Sawyer, N. Bencomo, B.H.C. Cheng, D. Hughes, Goal-based modeling of dynamically adaptive system requirements, in: Proc. ECBS, IEEE, 2008, pp. 36–45.
[49] X. Peng, B. Chen, Y. Yu, W. Zhao, Self-tuning of software systems through dynamic quality tradeoff and value-based feedback control loop, J. Syst. Softw. 85 (12) (2012) 2707–2719.
[50] M. Vrbaski, G. Mussbacher, D. Petriu, D. Amyot, Goal models as run-time entities in context-aware systems, in: Proc. MRT, ACM, 2012, pp. 3–8.
[51] Software Engineering Institute (Carnegie Mellon University), Software Product Lines, 2014. URL: http://www.sei.cmu.edu/productlines/.
[52] H. Gomaa, M. Hussein, Dynamic software reconfiguration in software product families, in: Software Product-Family Engineering, in: LNCS, vol. 3014, Springer, 2004, pp. 435–444.
[53] S. Hallsteinsen, M. Hinchey, K. Schmid, Dynamic software product lines, IEEE Comput. 41 (4) (2008) 93–95.
[54] J. Bosch, Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach, ACM/Addison-Wesley, 2000.
[55] K. Pohl, G. Böckle, F.J. van der Linden, Software Product Line Engineering: Foundations, Principles and Techniques, Springer, 2005.
[56] S. Hallsteinsen, E. Stav, A. Solberg, J. Floch, Using product line techniques to build adaptive systems, in: Proc. SPLC, IEEE, 2006, pp. 141–150.
[57] J. Lee, K.C. Kang, A feature-oriented approach to developing dynamically reconfigurable products in product line engineering, in: Proc. SPLC, IEEE, 2006, pp. 131–140.
[58] M. Cordy, A. Classen, P. Heymans, A. Legay, P.-Y. Schobbens, Model checking adaptive software with featured transition systems, in: Assurances for Self-Adaptive Systems, in: LNCS, vol. 7740, Springer, 2013, pp. 1–29.
[59] K. Geihs, R. Reichle, M. Wagner, M.U. Khan, Modeling of context-aware self-adaptive applications in ubiquitous and service-oriented environments, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 146–163.
[60] S. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, G. Papadopoulos, A development framework and methodology for self-adapting applications in ubiquitous computing environments, J. Syst. Softw. 85 (12) (2012) 2840–2859.
[61] C. Atkinson, T. Kuhne, Model-driven development: a metamodeling foundation, IEEE Softw. 20 (5) (2003) 36–41.
[62] T. Vogel, A. Seibel, H. Giese, The role of models and megamodels at runtime, in: Models in Software Engineering, in: LNCS, vol. 6627, Springer, 2011, pp. 224–238.
[63] T. Vogel, H. Giese, Adaptation and abstract runtime models, in: Proc. SEAMS, ACM, 2010, pp. 39–48.
[64] T. Vogel, H. Giese, A language for feedback loops in self-adaptive systems: executable runtime megamodels, in: Proc. SEAMS, IEEE, 2012, pp. 129–138.
[65] G. Lehmann, M. Blumendorf, F. Trollmann, S. Albayrak, Meta-modeling runtime models, in: Models in Software Engineering, in: LNCS, vol. 6627, Springer, 2011, pp. 209–223.
[66] B. Morin, O. Barais, J.-M. Jézéquel, F. Fleurey, A. Solberg, Models@ Run.time to support dynamic adaptation, IEEE Comput. 42 (10) (2009) 44–51.
[67] G. Blair, N. Bencomo, R.B. France, Models@ Run.time, IEEE Comput. 42 (10) (2009) 22–27.
[68] I. Epifani, C. Ghezzi, R. Mirandola, G. Tamburrelli, Model evolution by run-time parameter adaptation, in: Proc. ICSE, IEEE, 2009, pp. 111–121.

[69] N. Bencomo, On the use of software models during software execution, in: Proc. MISE, IEEE, 2009, pp. 62–67.
[70] G.H. Alférez, V. Pelechano, Dynamic evolution of context-aware systems with models at runtime, in: Model Driven Engineering Languages and Systems, in: LNCS, vol. 7590, Springer, 2012, pp. 70–86.
[71] F. Fleurey, V. Dehlen, N. Bencomo, B. Morin, J.-M. Jézéquel, Modeling and validating dynamic adaptation, in: Models in Software Engineering, in: LNCS, vol. 5421, Springer, 2009, pp. 97–108.
[72] N. Bencomo, G. Blair, Using architecture models to support the generation and operation of component-based adaptive systems, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 183–200.
[73] D. Menasce, H. Gomaa, S. Malek, J.P. Sousa, SASSY: a framework for self-architecting service-oriented systems, IEEE Softw. 28 (6) (2011) 78–85.
[74] J. Zhang, B.H.C. Cheng, Model-based development of dynamically adaptive software, in: Proc. ICSE, ACM, 2006, pp. 371–380.
[75] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, J. Stafford, Documenting Software Architectures: Views and Beyond, second ed., Addison-Wesley, 2010.
[76] M. Shaw, D. Garlan, Software Architecture: Perspectives on An Emerging Discipline, Prentice Hall, 1996.
[77] E. Gat, On three-layer architectures, in: Artificial Intelligence and Mobile Robots: Case Studies of Successful Robot Systems, MIT/AAAI Press, 1998, pp. 195–210.
[78] S. White, J. Hanson, I. Whalley, D. Chess, J. Kephart, An architectural approach to autonomic computing, in: Proc. ICAC, IEEE, 2004, pp. 2–9.
[79] D. Weyns, Architecture-Based Design of Multi-Agent Systems, Springer, 2010.
[80] J.C. Georgas, A. van der Hoek, R.N. Taylor, Using architectural models to manage and visualize runtime adaptation, IEEE Comput. 42 (10) (2009) 52–60.
[81] J.C. Georgas, R.N. Taylor, Policy-based architectural adaptation management: robotics domain case studies, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 89–108.
[82] D. Sykes, W. Heaven, J. Magee, J. Kramer, From goals to components: a combined approach to self-management, in: Proc. SEAMS, ACM, 2008, pp. 1–8.
[83] W. Heaven, D. Sykes, J. Magee, J. Kramer, A case study in goal-driven architectural adaptation, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 109–127.
[84] D. Garlan, V. Poladian, B.R. Schmerl, J.P. Sousa, Task-based self-adaptation, in: Proc. WOSS, ACM, 2004, pp. 54–57.
[85] A.J. Ramirez, B.H.C. Cheng, Design patterns for developing dynamically adaptive systems, in: Proc. SEAMS, ACM, 2010, pp. 49–58.
[86] T. De Wolf, T. Holvoet, Design patterns for decentralised coordination in self-organising emergent systems, in: Engineering Self-Organising Systems, in: LNCS, vol. 4335, Springer, 2007, pp. 28–49.
[87] L. Gardelli, M. Viroli, A. Omicini, Design patterns for self-organizing multiagent systems, in: Proc. CEEMAS, in: LNCS, vol. 4696, Springer, 2007, pp. 123–132.
[88] O. Babaoglu, G. Canright, A. Deutsch, G. Caro, F. Ducatelle, et al., Design patterns from biology for distributed computing, ACM Trans. Auton. Adapt. Syst. 1 (1) (2006) 26–66.
[89] S.-W. Cheng, V.V. Poladian, D. Garlan, B.R. Schmerl, Improving architecture-based self-adaptation through resource prediction, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 71–88.
[90] P. Oreizy, N. Medvidovic, R.N. Taylor, Runtime software adaptation: framework, approaches, and styles, in: Proc. ICSE Companion, ACM, 2008, pp. 899–910.
[91] S.-W. Cheng, D. Garlan, B.R. Schmerl, J.P. Sousa, B. Spitnagel, P. Steenkiste, Using architectural style as a basis for system self-repair, in: Proc. WISCA, Kluwer, 2002, pp. 45–59.
[92] S.-W. Cheng, D. Garlan, B.R. Schmerl, P. Steenkiste, N. Nu, Software architecture-based adaptation for grid computing, in: Proc. HPDC, IEEE, 2002, pp. 389–398.
[93] S.-W. Cheng, D. Garlan, B.R. Schmerl, J.P. Sousa, B. Spitznagel, P. Steenkiste, N. Hu, Software architecture-based adaptation for pervasive systems, in: Trends in Network and Pervasive Computing—ARCS 2002, Springer, 2002, pp. 67–82.
[94] S. Malek, G. Edwards, Y. Brun, H. Tajalli, J. Garcia, I. Krka, N. Medvidovic, M. Mikic-Rakic, G.S. Sukhatme, An architecture-driven software mobility framework, J. Syst. Softw. 83 (6) (2010) 972–989.
[95] S. Sicard, F. Boyer, N. De Palma, Using components for architecture-based management: the self-repair case, in: Proc. ICSE, ACM, 2008, pp. 101–110.
[96] C. Canal, E. Pimentel, J.M. Troya, Specification and refinement of dynamic software architectures, in: Software Architecture, in: IFIP—The International Federation for Information Processing, vol. 12, Kluwer, 1999, pp. 107–126.
[97] J. Magee, N. Dulay, S. Eisenbach, J. Kramer, Specifying distributed software architectures, in: Software Engineering—ESEC'95, in: LNCS, vol. 989, Springer, 1995, pp. 137–153.
[98] I. Georgiadis, J. Magee, J. Kramer, Self-organising software architectures for distributed systems, in: Proc. WOSS, ACM, 2002, pp. 33–38.
[99] R. Allen, R. Douence, D. Garlan, Specifying dynamism in software architectures, in: Proc. Workshop on Foundations of CBSE, 1997, pp. 11–22.
[100] M. Endler, J. Wei, Programming generic dynamic reconfigurations for distributed applications, in: Proc. CDS, 1992, pp. 68–79.
[101] M. Wermelinger, Towards a chemical model for software architecture reconfiguration, in: Proc. CDS, IEEE, 1998, pp. 111–118.
[102] M. Wermelinger, A. Lopes, J.L. Fiadeiro, A graph based architectural (re)configuration language, ACM SIGSOFT Softw. Eng. Notes 26 (5) (2001) 21–32.
[103] D. Luckham, J. Kenney, L. Augustin, J. Vera, D. Bryan, W. Mann, Specification and analysis of system architecture using Rapide, IEEE Trans. Softw. Eng. 21 (4) (1995) 336–354.
[104] S.-W. Cheng, D. Garlan, Stitch: a language for architecture-based self-adaptation, J. Syst. Softw. 85 (12) (2012) 2860–2875.
[105] J.S. Bradbury, J.R. Cordy, J. Dingel, M. Wermelinger, A survey of self-management in dynamic software architecture specifications, in: Proc. WOSS, ACM, 2004, pp. 28–33.
[106] J. Malenfant, M. Jacques, F. Demers, A tutorial on behavioral reflection and its implementation, in: Proc. Reflection, 1996, pp. 1–20.
[107] D.G. Bobrow, R.P. Gabriel, J.L. White, CLOS in context: the shape of the design space, in: Object-Oriented Programming, MIT Press, 1993, pp. 29–61.
[108] B.C. Smith, Procedural reflection in programming languages (Ph.D. Thesis), Massachusetts Institute of Technology, 1982.
[109] P. Maes, Concepts and experiments in computational reflection, in: Proc. OOPSLA, ACM, 1987, pp. 147–155.
[110] J. Andersson, R. de Lemos, S. Malek, D. Weyns, Reflecting on self-adaptive software systems, in: Proc. SEAMS, IEEE, 2009, pp. 38–47.
[111] E. Tanter, J. Noyé, D. Caromel, P. Cointe, Partial behavioral reflection: spatial and temporal selection of reification, in: Proc. OOPSLA, ACM, 2003, pp. 27–46.
[112] F. Tisato, A. Savigni, W. Cazzola, A. Sosio, Architectural reflection: realising software architectures via reflective activities, in: Engineering Distributed Objects, in: LNCS, vol. 1999, Springer, 2001, pp. 102–115.
[113] D. Garlan, B.R. Schmerl, Using architectural models at runtime: research challenges, in: Software Architectures, in: LNCS, vol. 3047, Springer, 2004, pp. 200–205.
[114] R. Morrison, D. Balasubramaniam, F. Oquendo, B. Warboys, R.M. Greenwood, An active architecture approach to dynamic systems co-evolution, in: Software Architecture, in: LNCS, vol. 4758, Springer, 2007, pp. 2–10.
[115] S. Hallsteinsen, E. Stav, J. Floch, Self-adaptation for everyday systems, in: Proc. WOSS, ACM, 2004, pp. 69–74.
[116] J. Dowling, V. Cahill, The *K*-component architecture meta-model for self-adaptive software, in: Metalevel Architectures and Separation of Crosscutting Concerns, in: LNCS, vol. 2192, Springer, 2001, pp. 81–88.
[117] R. Razavi, J.-F. Perrot, N. Guelfi, Adaptive modeling: an approach and a method for implementing adaptive agents, in: Massively Multi-Agent Systems I, in: LNCS, vol. 3446, Springer, 2005, pp. 136–148.
[118] W. Cazzola, Evaluation of object-oriented reflective models, in: Object-Oriented Technology: ECOOP'98 Workshop Reader, in: LNCS, vol. 1543, Springer, 1998, pp. 386–387.
[119] F. Kon, M. Román, P. Liu, J. Mao, T. Yamane, C. Magalhã, R.H. Campbell, Monitoring, security, and dynamic configuration with the dynamicTAO reflective ORB, in: Middleware 2000, in: LNCS, vol. 1795, Springer, 2000, pp. 121–143.
[120] G.S. Blair, G. Coulson, A. Andersen, L. Blair, M. Clarke, et al. The design and implementation of open ORB 2, IEEE Distrib. Syst. Online, 2 (June) (2001).

[121] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, T. Sivaharan, A generic component model for building systems software, ACM Trans. Comput. Syst. 26 (1) (2008) Art. 1.
[122] F. Kon, F. Costa, G. Blair, R.H. Campbell, The case for reflective middleware, Commun. ACM 45 (6) (2002) 33–38.
[123] L. Capra, W. Emmerich, C. Mascolo, CARISMA: context-aware reflective middleware system for mobile applications, IEEE Trans. Softw. Eng. 29 (10) (2003) 929–944.
[124] P. Sawyer, N. Bencomo, J. Whittle, E. Letier, A. Finkelstein, Requirements-aware systems: a research agenda for RE for self-adaptive systems, in: Proc. RE, IEEE, 2010, pp. 95–103.
[125] C. Szyperski, Component Software: Beyond Object-Oriented Programming, second ed., Addison-Wesley, 2002.
[126] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, J.-B. Stefani, The FRACTAL component model and its support in Java, Softw. - Pract. Exp. 36 (11–12) (2006) 1257–1284.
[127] P.-C. David, T. Ledoux, An aspect-oriented approach for developing self-adaptive fractal components, in: Software Composition, in: LNCS, vol. 4089, Springer, 2006, pp. 82–97.
[128] G. Blair, T. Coupaye, J.-B. Stefani, Component-based architecture: the fractal initiative, Ann. Telecommun.—Ann. Télécommun. 64 (1–2) (2009) 1–4.
[129] C. Becker, M. Handte, G. Schiele, K. Rothermel, PCOM—a component system for pervasive computing, in: Proc. PerCom, IEEE, 2004, pp. 67–76.
[130] G. Huang, H. Mei, F.-Q. Yang, Runtime recovery and manipulation of software architecture of component-based systems, Autom. Softw. Eng. 13 (2) (2006) 257–281.
[131] H. Liu, M. Parashar, S. Hariri, A component based programming framework for autonomic applications, in: Proc. ICAC, IEEE, 2004, pp. 10–17.
[132] M. Aksit, Z. Choukair, Dynamic, adaptive and reconfigurable systems overview and prospective vision, in: Proc. ICDCS Workshops, IEEE, 2003, pp. 84–89.
[133] V.H. Nguyen, F. Fouquet, N. Plouzeau, O. Barais, A process for continuous validation of self-adapting component based systems, in: Proc. MRT, ACM, 2012, pp. 32–37.
[134] F. Irmert, T. Fischer, K. Meyer-Wegener, Runtime adaptation in a service-oriented component model, in: Proc. SEAMS, ACM, 2008, pp. 97–104.
[135] H. Klus, D. Niebuhr, A. Rausch, A component model for dynamic adaptive systems, in: Proc. ESSPE, ACM, 2007, pp. 21–28.
[136] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin, Aspect-oriented programming, in: ECOOP'97—Object-Oriented Programming, in: LNCS, vol. 1241, Springer, 1997, pp. 220–242.
[137] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W.G. Griswold, An overview of AspectJ, in: ECOOP 2001—Object-Oriented Programming, in: LNCS, vol. 2072, Springer, 2001, pp. 327–354.
[138] R. Haesevoets, E. Truyen, T. Holvoet, W. Joosen, Weaving the fabric of the control loop through aspects, in: Self-Organizing Architectures, in: LNCS, vol. 6090, Springer, 2009, pp. 38–65.
[139] A. Charfi, T. Dinkelaker, M. Mezini, A plug-in architecture for self-adaptive Web service compositions, in: Proc. ICWS, IEEE, 2009, pp. 35–42.
[140] P. Greenwood, L. Blair, A framework for policy driven auto-adaptive systems using dynamic framed aspects, in: Transactions on Aspect-Oriented Software Development II, in: LNCS, vol. 4242, Springer, 2006, pp. 30–65.
[141] T. Huang, G.-Q. Wu, J. Wei, Runtime monitoring composite Web services through stateful aspect extension, J. Comput. Sci. Tech. 24 (2) (2009) 294–308.
[142] O. Nierstrasz, M. Denker, L. Renggli, Model-centric, context-aware software adaptation, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 128–145.
[143] K. Czarnecki, U. Eisenecker, Generative Programming: Methods, Tools, and Applications, Addison-Wesley, 2000.
[144] M. Gouda, T. Herman, Adaptive programming, IEEE Trans. Softw. Eng. 17 (9) (1991) 911–921.
[145] U.A. Acar, G.E. Blelloch, R. Harper, Adaptive functional programming, ACM Trans. Program. Lang. Syst. 28 (6) (2006) 990–1034.
[146] C. Simpkins, S. Bhat, C. Isbell, M. Mateas, Towards adaptive programming: integrating reinforcement learning into a programming language, ACM SIGPLAN Not. 43 (10) (2008) 603–614.
[147] R. Keays, A. Rakotonirainy, Context-oriented programming, in: Proc. MobiDe, ACM, 2003, pp. 9–16.
[148] M. Autili, P. Di Benedetto, P. Inverardi, A programming model for adaptable Java applications, in: Proc. PPPJ, ACM, 2010, pp. 119–128.
[149] R. Hirschfeld, P. Costanza, O. Nierstrasz, Context-oriented programming, J. Object Technol. 7 (3) (2008) 125–151.
[150] B.H.C. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, et al., Software engineering for self-adaptive systems: a research roadmap, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 1–26.
[151] M. Shaw, Beyond objects: a software design paradigm based on process control, SIGSOFT Softw. Eng. Notes 20 (1) (1995) 27–38.
[152] T. Abdelzaher, Y. Diao, J.L. Hellerstein, C. Lu, X. Zhu, Introduction to Control Theory and Its Application to Computing Systems, in: Performance Modeling and Engineering, Springer, 2008, pp. 185–215.
[153] T. Söderström, P. Stoica, System Identification, Prentice-Hall, 1988.
[154] K.J. Astrom, B. Wittenmark, Adaptive Control, second ed., Addison-Wesley, 1994.
[155] M. Kokar, K. Baclawski, Y. Eracar, Control theory-based foundations of self-controlling software, IEEE Intell. Syst. 14 (3) (1999) 37–45.
[156] F. Křikava, P. Collet, R.B. France, Actor-based runtime model of adaptable feedback control loops, in: Proc. MRT, ACM, 2012, pp. 39–44.
[157] C. Hewitt, Viewing control structures as patterns of passing messages, Artif. Intell. 8 (3) (1977) 323–364.
[158] U. Richter, M. Mnif, J. Branke, C. Müller-Schloer, H. Schmeck, Towards a generic observer/controller architecture for organic computing, in: GI Jahrestagung (1), in: LNI, vol. 93, GI, 2006, pp. 112–119.
[159] H.A. Müller, M. Pezzè, M. Shaw, Visibility of control in adaptive systems, in: Proc. ULSSIS, ACM, 2008, pp. 23–26.
[160] R. Hebig, H. Giese, B. Becker, Making control loops explicit when architecting self-adaptive systems, in: Proc. SOAR, ACM, 2010, pp. 21–28.
[161] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, D. Phung, Self-managing systems: a control theory foundation, in: Proc. ECBS, IEEE, 2005, pp. 441–448.
[162] T. Patikirikorala, A. Colman, J. Han, L. Wang, A systematic survey on the design of self-adaptive software systems using control engineering approaches, in: Proc. SEAMS, IEEE, 2012, pp. 33–42.
[163] M.P. Papazoglou, P. Traverso, S. Dustdar, F. Leymann, Service-oriented computing: state of the art and research challenges, IEEE Comput. 40 (11) (2007) 38–45.
[164] E. Di Nitto, C. Ghezzi, A. Metzger, M. Papazoglou, K. Pohl, A journey to highly dynamic, self-adaptive service-based applications, Autom. Softw. Eng. 15 (3–4) (2008) 313–341.
[165] L. Liu, H. Schmeck, A roadmap towards autonomic service-oriented architectures, Int. Trans. Syst. Sci. Appl. 2 (3) (2006) 245–254.
[166] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, D.A. Menascé, Software adaptation patterns for service-oriented architectures, in: Proc. SAC, ACM, 2010, pp. 462–469.
[167] G. Di Marzo Serugendo, J. Fitzgerald, A. Romanovsky, MetaSelf—an architecture and a development method for dependable self-* systems, in: Proc SAC, ACM, 2010, pp. 457–461.
[168] E. Garcia, A. Giret, V. Botti, Software engineering for service-oriented MAS, in: Cooperative Information Agents XII, in: LNCS, vol. 5180, Springer, 2008, pp. 86–100.
[169] H. Cervantes, R.S. Hall, Autonomous adaptation to dynamic availability using a service-oriented component model, in: Proc. ICSE, IEEE, 2004, pp. 614–623.
[170] N.A. Qureshi, A. Perini, Requirements engineering for adaptive service based applications, in: Proc. RE, IEEE, 2010, pp. 108–111.
[171] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F.L. Presti, R. Mirandola, MOSES: a framework for QoS driven runtime adaptation of service-oriented systems, IEEE Trans. Softw. Eng. 38 (5) (2012) 1138–1159.
[172] T. De Wolf, T. Holvoet, Towards autonomic computing: agent-based modelling, dynamical systems analysis, and decentralised control, in: Proc. INDIN, IEEE, 2003, pp. 470–479.

[173] T. De Wolf, T. Holvoet, Emergence versus self-organisation different concepts but promising when combined, in: Engineering Self-Organising Systems, in: LNCS, vol. 3464, Springer, 2005, pp. 1–15.
[174] T. De Wolf, T. Holvoet, Towards a methodology for engineering self-organising emergent systems, in: Proc. SOAS, IOS Press, 2005, pp. 18–34.
[175] G. Tesauro, D.M. Chess, W.E. Walsh, R. Das, A. Segal, I. Whalley, J.O. Kephart, S.R. White, A multi-agent systems approach to autonomic computing, in: Proc. AAMAS—Vol. 1, IEEE, 2004, pp. 464–471.
[176] C. Bernon, V. Camps, M.-P. Gleizes, G. Picard, Tools for self-organizing applications engineering, in: Engineering Self-Organizaning Systems, in: LNCS, vol. 2977, Springer, 2004, pp. 283–298.
[177] M. Mamei, F. Zambonelli, Programming pervasive and mobile computing applications: the TOTA approach, in: Proc. PerCom, IEEE, 2004, pp. 263–273.
[178] M.H. Cruz Torres, T. Van Beers, T. Holvoet, (No) more design patterns for multi-agent systems, in: Proc. SPLASH Workshops, ACM, 2011, pp. 213–220.
[179] R. Nagpal, A catalog of biologically-inspired primitives for engineering self-organization, in: Engineering Self-Organising Systems, in: LNCS, vol. 2977, Springer, 2004, pp. 53–62.
[180] F. Zambonelli, M. Viroli, A survey on nature-inspired metaphors for pervasive service ecosystems, Int. J. Pervasive Comput. Commun. 7 (3) (2011) 186–204.
[181] H. Van Dyke Parunak, "Go to the ant": engineering principles from natural multi-agent systems, Ann. Oper. Res. 75 (1997) 69–101.
[182] M. Mamei, R. Menezes, R. Tolksdorf, F. Zambonelli, Case studies for self-organization in computer science, J. Syst. Archit. 52 (8–9) (2006) 443–460.
[183] S. Selvakennedy, S. Sinnappan, Y. Shang, A biologically-inspired clustering protocol for wireless sensor networks, Comput. Commun. 30 (14–15) (2007) 2786–2801.
[184] H. Zhang, J. Llorca, Nature-inspired self-organization, control, and optimization in heterogeneous wireless networks, IEEE Mob. Comput. 11 (7) (2012) 1207–1222.
[185] E. Bonabeau, M. Dorigo, G. Theraulaz, Swarm Intelligence: From Natural to Artificial Systems, Oxford University Press, 1999.
[186] M. Dorigo, G.A. Di Caro, L.M. Gambardella, Ant algorithms for discrete optimization, Artif. Life 5 (2) (1999) 137–172.
[187] M. Shackleton, F. Saffre, R. Tateson, E. Bonsma, C. Roadknight, Autonomic computing for pervasive ICT—a whole-system perspective, in: Intelligent Spaces, Computer Communications and Networks, Springer, 2004, pp. 323–335.
[188] R. Tateson, S. Howard, R. Bradbeer, Nature-inspired self-organisation in wireless communications networks, in: Engineering Self-Organising Systems, vol. 2977, Springer, 2004, pp. 63–74.
[189] R. Tateson, Self-organising pattern formation: fruit flies and cell phones, in: Parallel Problem Solving from Nature—PPSN V, in: LNCS, vol. 1498, Springer, 1998, pp. 732–741.
[190] U. Lee, E. Magistretti, M. Gerla, P. Bellavista, P. Lio, K.-W. Lee, Bio-inspired multi-agent data harvesting in a proactive urban monitoring environment, Ad Hoc Networks 7 (4) (2009) 725–741.
[191] M.B. Miller, B.L. Bassler, Quorum sensing in bacteria, Annu. Rev. Microbiol. 55 (2001) 165–199.
[192] G. Werner-Allen, G. Tewari, A. Patel, M. Welsh, R. Nagpal, Firefly-inspired sensor network synchronicity with realistic radio effects, in: Proc. SenSys, ACM, 2005, pp. 142–153.
[193] C. Bourjot, V. Chevrier, V. Thomas, A new swarm mechanism based on social spiders colonies: from Web weaving to region detection, Web Intell. Agent Syst. 1 (2003) 47–64.
[194] D. Dasgupta, Advances in artificial immune systems, IEEE Comput. Intell. Mag. 1 (4) (2006) 40–49.
[195] O. Khatib, Real-time obstacle avoidance for manipulators and mobile robots, in: Proc. Robotics and Automation—Vol. 2, IEEE, 1985, pp. 500–505.
[196] J.H. Reif, H. Wang, Social potential fields: a distributed behavioral control for autonomous robots, Robot. Auton. Syst. 27 (1999) 171–194.
[197] O. Simonin, Construction of numerical potential fields with reactive agents, in: Proc. AAMAS, ACM, 2005, pp. 1351–1352.
[198] D. Weyns, N. Boucké, T. Holvoet, A field-based versus a protocol-based approach for adaptive task assignment, in: Autonomous Agents and Multi-Agent Systems, vol. 17, Springer, 2008, pp. 288–319.
[199] J.-P. Banâtre, T. Priol, Chemical programming of future service-oriented architectures, J. Softw. 4 (7) (2009) 738–746.
[200] M. Viroli, M. Casadei, Biochemical tuple spaces for self-organising coordination, in: Coordination Models and Languages, in: LNCS, vol. 5521, Springer, 2009, pp. 143–162.
[201] C. Villalba, F. Zambonelli, Towards nature-inspired pervasive service ecosystems: concepts and simulation experiences, J. Netw. Comput. Appl. 34 (2) (2011) 589–602.
[202] H. Van Dyke Parunak, J. Sauter, M. Fleischer, A. Ward, The RAPPID project: symbiosis between industrial requirements and MAS research, Auton. Agents Multi-Agent Syst. 2 (2) (1999) 111–140.
[203] N. Salazar, J.A. Rodriguez-Aguilar, J.L. Arcos, Robust coordination in large convention spaces, AI Commun. 23 (4) (2010) 357–372.
[204] M. Güdemann, F. Ortmeier, W. Reif, Formal modeling and verification of systems with self-*x* properties, in: Autonomic and Trusted Computing, in: LNCS, vol. 4158, Springer, 2006, pp. 38–47.
[205] F. Nafz, H. Seebach, J.-P. Steghöfer, S. Bäumler, W. Reif, A formal framework for compositional verification of organic computing systems, in: Autonomic and Trusted Computing, in: LNCS, vol. 6407, Springer, 2010, pp. 17–31.
[206] H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Müller-Schloer, H. Schmeck, Organic control of traffic lights, in: Autonomic and Trusted Computing, in: LNCS, vol. 5060, Springer, 2008, pp. 219–233.
[207] M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach, W. Reif, A specification and construction paradigm for organic computing systems, in: Proc. SASO, IEEE, 2008, pp. 233–242.
[208] T. King, A. Ramirez, R. Cruz, P. Clarke, An integrated self-testing framework for autonomic computing systems, J. Comput. 2 (9) (2007).
[209] G. Smith, J.W. Sanders, Formal development of self-organising systems, in: Autonomic and Trusted Computing, in: LNCS, vol. 5586, Springer, 2009, pp. 90–104.
[210] F. Zambonelli, A. Omicini, Challenges and research directions in agent-oriented software engineering, Auton. Agents Multi-Agent Syst. 9 (3) (2004) 253–283.
[211] E. Vassev, J. Paquet, ASSL—autonomic system specification language, in: Proc. SEW, IEEE, 2007, pp. 300–309.
[212] A. Filieri, G. Tamburrelli, Probabilistic verification at runtime for self-adaptive systems, in: Assurances for Self-Adaptive Systems, in: LNCS, vol. 7740, Springer, 2013, pp. 30–59.
[213] C. Priesterjahn, D. Steenken, M. Tichy, Timed hazard analysis of self-healing systems, in: Assurances for Self-Adaptive Systems, in: LNCS, vol. 7740, Springer, 2013, pp. 112–151.
[214] D. Weyns, M.U. Iftikhar, D.G. de la Iglesia, T. Ahmad, A survey of formal methods in self-adaptive systems, in: Proc. C3S2E, ACM, 2012, pp. 67–79.
[215] A. Elkhodary, S. Malek, N. Esfahani, On the role of features in analyzing the architecture of self-adaptive software systems, in: Proc. MRT, ACM/IEEE, 2009, pp. 41–50.
[216] D. Fisch, E. Kalkowski, B. Sick, Collaborative learning by knowledge exchange, in: Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011, pp. 267–280.
[217] J. Dowling, V. Cahill, Self-managed decentralised systems using *K*-components and collaborative reinforcement learning, in: Proc. WOSS, ACM, 2004, pp. 39–43.
[218] C. Parra, D. Romero, S. Mosser, R. Rouvoy, L. Duchien, L. Seinturier, Using constraint-based optimization and variability to support continuous self-adaptation, in: Proc. SAC, ACM, 2012, pp. 486–491.
[219] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, 1998.
[220] S. Tomforde, H. Prothmann, F. Rochner, J. Branke, J. Hähner, C. Müller-Schloer, H. Schmeck, Decentralised progressive signal systems for organic traffic control, in: Proc. SASO, IEEE, 2008, pp. 413–422.
[221] S. Tomforde, H. Prothmann, J. Branke, J. Hähner, M. Mnif, C. Müller-Schloer, U. Richter, H. Schmeck, Observation and control of organic systems, in: Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011, pp. 325–338.

[222] E. Cakar, N. Fredivianus, J. Hähner, J. Branke, C. Müller-Schloer, H. Schmeck, Aspects of learning in OC systems, in: Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011, pp. 237–251.
[223] W. Brockmann, N. Rosemann, E. Maehle, A framework for controlled self-optimisation in modular system architectures, in: Organic Computing—A Paradigm Shift for Complex Systems, Springer, 2011, pp. 281–294.
[224] T. Bäck, H.-P. Schwefel, An overview of evolutionary algorithms for parameter optimization, Evol. Comput. 1 (1) (1993) 1–23.
[225] S. Abdelwahed, N. Kandasamy, S. Neema, A control-based framework for self-managing distributed computing systems, in: Proc. WOSS, ACM, 2004, pp. 3–7.
[226] D.M. Berry, B.H.C. Cheng, J. Zhang, The four levels of requirements engineering for and in dynamic adaptive systems, in: Proc. REFSQ, 2005.
[227] J. Whittle, P. Sawyer, N. Bencomo, B.H.C. Cheng, J.-M. Bruel, RELAX: a language to address uncertainty in self-adaptive systems requirement, Requir. Eng. 15 (2) (2010) 177–196.
[228] N. Bencomo, J. Whittle, P. Sawyer, A. Finkelstein, E. Letier, Requirements reflection: requirements as runtime entities, in: Proc. ICSE—Vol. 2, ACM/IEEE, 2010, pp. 199–202.
[229] L. Baresi, L. Pasquale, P. Spoletini, Fuzzy goals for requirements-driven adaptation, in: Proc. RE, IEEE, 2010, pp. 125–134.
[230] L. Pasquale, L. Baresi, B. Nuseibeh, Towards adaptive systems through requirements@runtime, in: Proc. MRT, CEUR-WS.org, 2011, pp. 13–24.
[231] V.A. Silva Souza, Requirements-based software system adaptation (Ph.D. Thesis), University of Trento, 2012.
[232] J. Whittle, P. Sawyer, N. Bencomo, B.H.C. Cheng, J. Bruel, RELAX: incorporating uncertainty into the specification of self-adaptive systems, in: Proc. RE, IEEE, 2009, pp. 79–88.
[233] A. Dardenne, A. van Lamsweerde, S. Fickas, Goal-directed requirements acquisition, Sci. Comput. Program. 20 (12) (1993) 3–50.
[234] E.S. Yu, Towards modelling and reasoning support for early-phase requirements engineering, in: Proc. RE, IEEE, 1997, pp. 226–235.
[235] N.A. Qureshi, A. Perini, Continuous adaptive requirements engineering: an architecture for self-adaptive service-based applications, in: Proc. RE@RunTime, IEEE, 2010, pp. 17–24.
[236] M. Morandini, L. Penserini, A. Perini, Modelling self-adaptivity: a goal-oriented approach, in: Proc. SASO, IEEE, 2008, pp. 469–470.
[237] P. Bresciani, A. Perini, P. Giorgini, F. Giunchiglia, J. Mylopoulos, Tropos: an agent-oriented software development methodology, Auton. Agents Multi-Agent Syst. 8 (3) (2004) 203–236.
[238] D. Harel, R. Marelly, Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine, Springer, 2003.
[239] D. Garlan, D. Siewiorek, A. Smailagic, P. Steenkiste, Project aura: toward distraction-free pervasive computing, IEEE Pervasive Comput. 1 (2) (2002) 22–31.
[240] C. Krupitzer, S. VanSyckel, C. Becker, FESAS: towards a framework for engineering self-adaptive systems, in: Proc. SASO, IEEE, 2013, pp. 263–264.
[241] J. Andersson, R. de Lemos, S. Malek, D. Weyns, Modeling dimensions of self-adaptive software systems, in: Software Engineering for Self-Adaptive Systems, in: LNCS, vol. 5525, Springer, 2009, pp. 27–47.
[242] J. Andersson, L. Baresi, N. Bencomo, R. de Lemos, A. Gorla, P. Inverardi, T. Vogel, Software engineering processes for self-adaptive systems, in: Software Engineering for Self-Adaptive Systems II, in: LNCS, vol. 7475, Springer, 2013, pp. 51–75.
[243] H. Seebach, F. Nafz, J.-P. Steghofer, W. Reif, A software engineering guideline for self-organizing resource-flow systems, in: Proc. SASO, IEEE, 2010, pp. 194–203.
[244] S. Lightstone, Foundations of autonomic computing development, in: Proc. EASe, IEEE, 2007, pp. 163–171.
[245] S. Frey, A. Diaconescu, I. Demeure, Architectural integration patterns for autonomic management systems, in: Proc. EASe, 2012.
[246] S. VanSyckel, D. Schäfer, G. Schiele, C. Becker, Configuration management for proactive adaptation in pervasive environments, in: Proc. SASO, IEEE, 2013, pp. 131–140.
[247] H.A. Müller, H.M. Kienle, U. Stege, Autonomic computing—now you see it, now you don't, in: Software Engineering, in: LNCS, vol. 5413, Springer, 2009, pp. 32–54.
[248] A.J. Ramirez, A.C. Jensen, B.H.C. Cheng, A taxonomy of uncertainty for dynamically adaptive systems, in: Proc. SEAMS, IEEE, 2012, pp. 99–108.