Chapter 8
State of the Art in Architectures for Self-Aware Computing Systems

Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, Nelly Bencomo, Kurt Geihs, Samuel Kounev, and Kirstie Bellman

Abstract In this chapter, we review the state of the art in self-aware computing systems with a particular focus on software architectures. Therefore, we compare existing approaches targeting computing systems with similar characteristics as self-aware systems to the architectural concepts for single and collective self-aware systems discussed in the previous chapters. These approaches are particularly reference architectures and architectural frameworks and languages. Based on this comparison, we discuss open challenges for architectures of self-aware computing systems.

Holger Giese
Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany, e-mail: `holger.giese@hpi.de`

Thomas Vogel
Hasso Plattner Institute for Software Systems Engineering at the University of Potsdam, Prof.-Dr.-Helmert-Str. 2-3, D-14482 Potsdam, Germany, e-mail: `thomas.vogel@hpi.de`

Ada Diaconescu
Telécom ParisTech, Equipe S3, Departement INFRES, 46 rue Barrault, 75013 Paris, France, e-mail: `ada.diaconescu@telecom-paristech.fr`

Sebastian Götz
TU Dresden, Germany, e-mail: `sebastian.goetz@acm.org`

Nelly Bencomo
Aston Institute for Systems Analytics, Aston University, Birmingham, UK, e-mail: `nelly@acm.org`

Kurt Geihs
University Kassel, Wilhelmshher Allee 73, D-34121 Kassel, Germany, e-mail: `geihs@uni-kassel.de`

Samuel Kounev
University of Wrzburg, Department of Computer Science, Am Hubland, D-97074 Wrzburg, Germany e-mail: `samuel.kounev@uni-wuerzburg.de`

Kirstie Bellman
Aerospace Integration Science Center, The Aerospace Corporation, US, e-mail: `Kirstie.L.Bellman@aero.org`

8.1 Introduction

We studied the architectural dimension of the vision of self-aware computing systems in the previous chapters. Particularly, we discussed generic architectural concepts in Chapter 5, which we used as a basis to investigate architectures for single and collective self-aware computing systems. Therefore, we discussed architectural designs and styles—inspired by control schemes and architectures of self-adaptive software—for pre-reflective, reflective, and meta-reflective self-awareness and how such systems may construct reflections of themselves and their environment in Chapter 6. We extended this discussion in Chapter 7 for collective self-aware computing systems and how they address pre-reflective, reflective, and meta-reflective self-awareness at the architectural level with respect to the interactions and the organization of the collective. In this chapter, we discuss the state of the art in architectures for self-aware computing systems by comparing existing approaches with the major ideas presented in Chapters 5, 6, and 7.

In this context, the self-aware computing paradigm sketched in Chapter 1 emphasize a development that is already partially supported by a number of research initiatives aiming for more flexible software systems. Examples of such initiatives are autonomic computing [50], self-* systems [5], self-adaptive and self-managed systems [20, 21, 25, 26, 54, 71], organic computing [1, 64], or cognitive computing [49]. All of these initiatives advocate a paradigm shift for software from design-time decisions and understanding toward resolving issues dynamically at runtime.

Such approaches traditionally consider *reactive* solutions that dynamically act in response to changes causing issues (cf. [32, 54]). In contrast, self-aware computing emphasizes anticipating future changes or reasoning about the long-term future and therefore advocates another paradigm shift from a *reactive* to a *proactive* operation that integrates the ability to learn, reason, and act at runtime (cf. [19, 22, 46]). This trend is well in line with the ideas centered around the notion of self-aware computing [1, 2, 47, 51, 57, 62, 81], runtime models [9, 11, 12, 75, 78, 80], and related terms [24, 28, 61, 67]. A broad discussion of such research initiatives with respect to self-aware computing systems can be found in Chapter 2.

In this chapter, we review the state of the art concerning self-aware computing systems with the particular focus on the *software architecture* [74]. Moreover, we focus on *specific* approaches—in contrast to research initiatives—and compare them to the architectural concepts for self-aware computing systems discussed in the previous chapters. Thus, we cover the basic architectural concepts (see Chapter 5) and how pre-reflective, reflective, meta-reflective self-awareness and the related observe, analysis, and react as well as learning, reasoning, and acting processes can be organized at the architectural level for an individual self-aware computing system (see Chapter 6) as well as for a collective of such systems (see Chapter 7). The specific approaches we discuss in this chapter are either reference architectures or architectural frameworks and languages for software systems that share similarities with self-aware computing systems.

The rest of the chapter is organized as follows. We discuss the state of the art in architectures for self-aware computing systems in three steps. We discuss existing reference architectures in Section 8.2, architectural frameworks and languages in Section 8.3, and open challenges for architectures of self-awareness computing systems in Section 8.4. Finally, we conclude the chapter in Section 25.5.

8.2 Reference Architectures

In this section, we discuss existing reference architectures and compare them to the developed architectural concepts for self-aware computing systems. These reference architectures address systems related to self-aware computing. Specifically, we discuss the *MAPE-K* loop from the autonomic computing field [50], the reference architecture for self-managed systems proposed by Kramer and Magee [54], the reference architecture for models@run.time systems proposed by Aßmann et al. [4], the reference architecture from the organic computing field [72], the reference architecture for requirements reflection [10], and finally, architectural principles from artificial intelligence and multi-agent systems.

8.2.1 MAPE-K Loop

Core of autonomic systems, as defined in autonomic computing (cf. Chapter 2), is an entity that realizes the self-* properties [48]. This entity, referred to as *autonomic manager*, can be understood as an executable software unit that implements the adaptation logic in order to continuously meet the system's operational goals.

To structure the principle of operation exhibited by autonomic managers, a reference architecture based on a control loop, referred to as the *MAPE-K loop* has been proposed [50]. This reference architecture has the advantage that it offers a clear way to identify and classify areas of particular focus and thus, it is used by many researchers to communicate the architectural concepts of autonomic systems.

The acronym MAPE-K reflects the five main constituent phases of autonomic operations, i.e., *Monitor*, *Analyze*, *Plan*, *Execute*, and *Knowledge* (see Figure 8.1). Basically, the *Monitor* phase collects information from the sensors provided by the managed system and its context. The *Analyze* phase uses the data of the *Monitor* phase to assess the situation and determine any anomalies or problems. The *Plan* phase generates an adaptation plan to solve a detected problem. The *Execute* phase normally applies the generated adaptation plan on the actual system. A cross-cutting aspect shared among all phases of the loop is the *Knowledge* about the managed system and its context, capturing aspects such as the software architecture, execution environment, and hardware infrastructure on which the system is running. The knowledge may also explicitly capture the operational goals of the system, for instance, the target quality-of-service level the managed system should provide. The representation of the knowledge can take any form, for example, a performance model describing the performance behavior of the system.

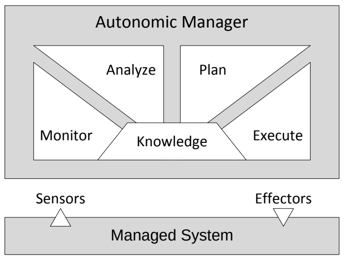

Fig. 8.1: MAPE-K (taken from Chapter 1).

The software engineering community uses a similar feedback loop concept, distinguishing the four phases *Collect*, *Analyze*, *Decide*, and *Act* (cf. [15,20]). Conceptually, the behavior of these phases is similar to the phases in the MAPE-K loop, however, this concept does not explicitly consider the *Knowledge* part. More details about the use of feedback loops in self-adaptive systems, such as the use of multiple, multi-level, positive, or negative feedback loops, can be found in [15, 20].

The activities or processes of a MAPE-K loop are similar to those of the LRA-M loop realized by self-aware computing systems (see Chapter 1). Nevertheless, the MAPE-K loop does not consider advanced forms of learning and reasoning but rather basic monitoring and analyses of the managed system. Both kinds of loops use runtime models representing parts of the system (especially the managed system in the case of MAPE-K) and goals, however, the MAPE-K loop uses operation-level goals and the LRA-M loop may use higher-level goals.

Separating an autonomic system into an autonomic manager reflecting on a managed system, we may consider the managed system to be the scope and the manager to be the span of awareness. Hence, the managed system is pre-reflective and the manager is reflective. Hence, systems realizing a MAPE-K loop may achieve reflective self-awareness (cf. Chapters 3 and 5). With respect to the forms of reflection discussed in Chapter 6, the MAPE-K loop adopts centralized reflection by employing a single autonomic manager. Additionally, hierarchical reflection can be achieved by structuring multiple autonomic managers in a hierarchy [50]. Consequently, the architectural style of MAPE-K is the hierarchical external approach (cf. Chapter 6). Finally, the MAPE-K loop as introduced in [50] does not consider collective systems such that the concepts proposed in Chapter 7 do not apply here.

8.2.2 Reference Architecture for Self-Managed Systems

Kramer and Magee [54] present a reference architecture for self-managed systems that is inspired by layered robot architectures. As depicted in Figure 8.2, the architecture has three layers and the functionality is distributed among them based on the execution time that increases from lower to higher layers.

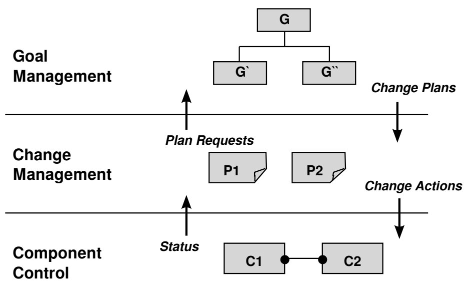

Fig. 8.2: Reference Architecture for Self-Managed Systems [54].

First, the *Component Control* layer implements the application functionality using a set of components. This layer reports its status to the next higher layer and supports change actions initiated by the higher layer for reconfiguring its component structure. Particularly, if the component control layer encounters a situation in which it cannot meet the application goals, it reports to the higher layer. In this case or if new goals are introduced by the topmost layer, the *Change Management* layer adapts the component control layer to be able to achieve the (new) goals in the current environmental situation. Therefore, the change management layer contains a set of pre-specified plans among which it selects an appropriate one for the current situation. If there is no appropriate plan available, a request is sent to the topmost layer to devise a plan. The *Goal Management* layer is responsible for planning, that is, creating plans to achieve the application goals. Planning is triggered by a request from the layer below or by introducing new goals. Created plans are then provided to the change management layer that enacts such plans by adapting the component control layer.

With respect to self-aware computing systems, this reference architecture is similar to the layered style discussed in Chapter 6. The component control layer is pre-reflective. The change management layer reflects on the component control layer and is therefore reflective. It typically will have some kind of awareness model of the lowest layer to reason how the lowest layer has to be adapted to execute a certain plan. The goal management layer explicitly maintains the application goals and reflects on the change management layer. For instance, introducing new goals requires new plans as well as potentially adjusted reasoning machinery at the change management layer to cope with new plans and their execution at the component control layer. Therefore, the topmost layer typically uses some awareness model addressing the change management and, by this, also the component control layer.

Considering the reference architecture depicted in Figure 8.2, we may interpret the arrows pointing bottom-up and top-down as awareness respectively expression links. In contrast to our discussion of layered architectures for self-aware systems, the reference architecture by Kramer and Magee [54] does not explicitly address the environmental context, the awareness and expression in terms of models, and processes that achieve such awareness and expression. The same holds for MORPH [13], a recent extension to the reference architecture, that explicitly distinguishes between adapting the configuration (e.g., changing the architectural structure) and adapting the behavior (e.g., changing the orchestration of the components) of the target system. This extension does not change the conceptual characteristics of the architecture with respect to self-awareness.

8.2.3 *Reference Architecture for Models@run.time Systems*

The models@run.time paradigm promotes the use of models and modeling techniques (e.g., model transformations and code generation) at runtime and can be applied to self-aware computing systems. Traditionally, models are only used during the design of systems with the aim to achieve platform independence, to increase reusability, and generally to improve the efficiency of software development, or as defined by Rothenberg:

> "Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality." [68]

The challenge posed by models@run.time is, thus, how to transfer existing modeling techniques, which focus on design time problems, to be applicable to runtime problems. Over the last decade, since the term models@run.time has been coined [12], a considerably large body of knowledge emerged from a vivid research community. The reference architecture proposed in [4] and depicted in Figure 8.3 summarizes a large subset of this knowledge by showing the major architectural constituents found in most of the approaches employing models@run.time.

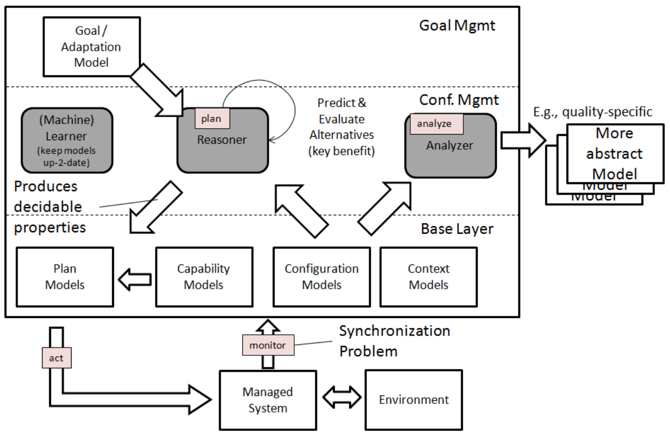

Fig. 8.3: A Reference Architecture for Models@run.time Systems [4].

A central characteristic of approaches based on the models@run.time paradigm is the distinction between two systems, where one monitors and acts upon the other. The former is often called the managing system while the latter is called the managed system. Some approaches additionally distinguish between the managed system and its environment. The managing system, on the other hand, can often be subdivided into three layers conforming to the reference architecture discussed in Section 8.2.2. The bottom layer, interfacing with the managed system, is comprised of models covering different concerns of the underlying managed system. The configuration model, often also simply called runtime model, reflects the current state of the underlying system. Additional models cover the managed system's capabilities (e.g., for adaptation, but also for use), context models focusing specifically on the managed system's environment, but also plan models constituting specifications of how to act upon the managed system. The middle layer consists of three active entities: a reasoner, an analyzer and a learner. The learner is responsible to keep all models of the lower layer in sync with the managed system. The analyzer provides means to further abstract (i.e., decompose) the managing system, which enables a hierarchical decomposition of models@run.time systems. Finally, the reasoner is in charge of processing the models from the lower layer with the aim of decision making. The reasoner also takes the third layer into account, which typically comprises requirements and goal models. The key benefits of models@run.time systems originate from the position of this reasoner, that is, the fact that this reasoner works on models. The level of abstraction can be adjusted to the respective reasoning task and, thus, allows to reduce the complexity of the reasoning tasks. Moreover, it is possible to use the models for predictions, which, in consequence, enables reasoning about possible, alternative, future states of the system. In summary, the key distinguishing features of models@run.time are predictive reflection and tractability by abstraction.

In comparison to self-aware computing as introduced in Chapter 1, models@run.time systems do not treat self-awareness as a first-class concept. It does, however, focus on the runtime model and observations to keep this model up-to-date. According to Chapter 6, the reference architecture of models@run.time denotes the hierarchical external reflection style. The concepts introduced in Chapter 7, which allow us to describe how systems can be composed or related to each other, are not an explicit part of the models@run.time paradigm, yet. This might change, due to increased interest in distributed models@run.time, where the focus is on integrating multiple models@run.time systems.

8.2.4 Organic Computing

The organic computing initiative is motivated by the ever increasing complexity of software systems and the need to enable such systems to adjust themselves to their users and not vice versa. The main objective of organic computing is the "controlled self-organisation" [72], that is, the ability of a system to self-adapt in accordance to external influences while at the same time providing guarantees in terms of trustworthy behavior. To reach this goal, the organic computing initiative introduced a novel architectural concept: the observer/controller architecture depicted in Figure 8.4.

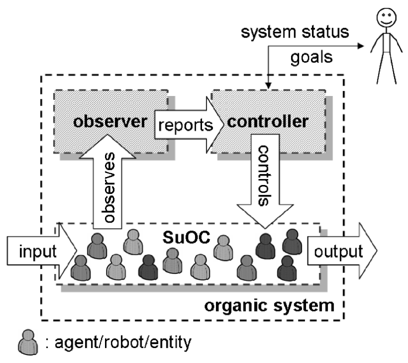

Fig. 8.4: The Observe/Control Loop of Organic Computing (taken from [72]).

The architecture is comprised of two top-level concepts: the organic system and a human user. While the organic system adheres to the basic input/computer/output principle of computing, the human user imposes goals on the organic system and is able to perceive the system status. The organic system is further decomposed into three major components: the system under observation and control (SuOC), the observer, and the controller. All human interaction is relayed by the controller. Notably, the input/compute/output principle is realized by the SuOC. The observer and controller impose a feedback loop upon the SuOC, where the former observes the SuOC and reports to the latter, which in turn controls the SuOC. An important characteristic of the SuOC is that it is comprised of agents (i.e., autonomous entities). In other words, the SuOC is already a set of self-organizing systems. The observer and controller enhance this system to achieve *controlled* self-organization.

Organic computing differs from the other reference architectures presented in this chapter in its initial situation and its main objective. While most other reference architectures aim at enhancing pre-reflective systems to become self-aware, in organic computing the starting point are autonomous systems and the objective is to enhance them such that they are enabled to provide trustworthy behavior. The fundamental idea of models@run.time, which is the use of abstract representations of the runtime system state (see Section 8.2.3), is present in organic computing as the link between the observer and controller, which is used to report "an aggregated quantified context (i.e., a description of the currently observed situation)" [72, p.3].

The MAPE-K loop in autonomic computing (see Section 8.2.1) can be mapped to the observe/control loop. The observer comprises the monitoring and analysis while the controller comprises the planning and execution. However, a shared knowledge base is not in the focus of organic computing. Moreover, the LRA-M loop of self-aware computing systems (see Chapter 1) has similarities to the observe/control loop, particularly, the external imposition of goals. In contrast to organic computing focusing on the realization of controlled behavior, the focus of self-aware computing is to structure the system under observation and control (i.e., the self) and to realize various forms of behavior such as self-explanation, self-modeling, etc.

With respect to the architectural concepts discussed in Chapter 5, the observation/control loop as a reference architecture (see Figure 8.4) is described at a higher level of abstraction. Therefore, organic computing systems do not explicitly consider the different levels of self-awareness (i.e., pre-reflective, reflective, and meta-reflective) although they include at least reflection since the observer and controller as the span reflect upon the SuOC as the scope. In this context, the observe and control links in Figure 8.4 can be interpreted as awareness and expression links. However, the reference architecture does not detail this reflection such that the potential use of awareness models, empirical data, or goal models is not explicitly covered.

Concerning the forms of reflection discussed in Chapter 6, the organic computing reference architecture realizes—similar to the MAPE-K loop—a hierarchical and centralized reflection as it employs a single observer and controller upon a set of agents. Moreover, the phenomena of overlapping, stacked, or cyclic reflection are not explicitly addressed by the reference architecture. Therefore, the reference architecture primarily adopts the hierarchical external architectural style without explicitly considering the other styles discussed in Chapter 6. Although organic computing targets open and self-organizing systems, the dimensions for collective self-aware systems discussed in Chapter 7 such as collective self-awareness and its levels, weak and strong self-aware collectives, different types of relations between systems of a collective, or the organizational patterns are not considered.

8.2.5 Requirements-Awareness

Traditionally, in requirements engineering (RE) it has been made the assumption that the environmental context is reasonably static and can be understood sufficiently well to permit the requirements model for a solution to be formulated with confidence during design time. However, effectively, environmental contexts are hardly static over long periods, and this can inhibit full understanding.

As discussed before, systems are being produced for environmental contexts that are subject to change over short periods and in ways that are badly understood. In part, this is because the machinery of self-adaptation and autonomic computing has improved, providing a means for systems to dynamically respond to changing contexts. As described in the book, self-awareness and self-adaptivity will become increasingly required properties for software systems. For this to become true, it is crucial that these software systems offer requirements-awareness to discover, reason about, and manage its own requirements that can dynamically change at runtime.

In this context, a key contribution is the seminal work on requirements monitoring [29]. Such monitoring is required because of deviations between the system's runtime behavior and the requirements model, which may trigger the need for an adaptation [8]. Such a deviation needs to be correlated with the state of the environment in order to diagnose the reasons and to perform appropriate adaptations. Where systems have the need to adapt dynamically in order to maintain satisfaction of their goals, RE ceases to be an entirely static, off-line activity and it additionally becomes a runtime activity. This is because design-time decisions about the requirements need to be made on incomplete and uncertain knowledge about the domain, context, and goals. There are clear benefits of being able to revise these decisions at runtime when more information can be acquired through runtime monitoring.

Therefore, requirements need to be runtime entities that can be reasoned over at runtime [10]. Implicit in the ability for a system to introspect (i.e., to be self-aware) on its requirements model is the representation of that model at runtime. The running system provides information as feedback to update the model and to increase its correspondence with reality, which is called causal connection. Analysis of the updated model may detect if a desired property (e.g., reliability and performance) is violated, causing self-adaptation actions that aim for guaranteeing the goals. Explicit use of computational reflection is a primary means to achieve requirements-awareness. The consequence is that there exists a runtime representation of the requirements model that is causally connected to the running system.

Requirements-awareness enables software systems to revise and re-assess design-time decisions at runtime when more information can be acquired about these by observing their own behavior (i.e., by being self-aware). Two research issues have been identified. One is the evolution of the requirements models and the maintenance of consistency between the requirements and the running systems during this evolution. To do this, it is necessary to specify the abstract adaptation thresholds that allow for uncertainty and unanticipated environmental conditions. The second issue is the need to maintain the synchronization (i.e., the causal connection) between the runtime requirements model and the architecture of the running system as either the requirements or the architecture may change (see Figure 8.5).

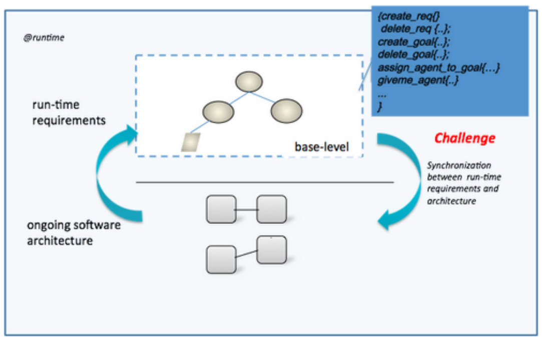

Fig. 8.5: Synchronization Between the Requirements and the Architecture [10].

Although requirement-awareness does not explicitly consider different levels of self-awareness, it employs reflection and achieves reflective self-awareness. The element shown in Figure 8.5 reflects upon a managed system to maintain representations of the system's runtime architecture and requirements. These runtime models correspond to the idea of system awareness models and goal models in self-aware computing systems (see Chapter 5). In this context, we can interpret the synchronization from the running system to the architectural runtime model and then to the requirements runtime model as an awareness link as well as the synchronization in the opposite direction as an expression link. Hence, reflection is considered as a general means to achieve requirements-awareness without detailing possible forms (e.g., local, centralized, hierarchical, or coordinated) or phenomena (e.g., overlapping, stacked, or cyclic) of reflection discussed in Chapter 6. Still, the step-wise synchronization to the architecture and then to the requirements can be seen as a specific form of hierarchical reflection that separates architectural and requirements concerns. Therefore, the basic adopted architectural style in requirements-awareness corresponds to the hierarchical external approach (see Chapter 6).

Finally, establishing requirements-awareness for collective systems is not discussed such that the architectural concepts for collective self-aware computing systems introduced in Chapter 7 do not apply here.

8.2.6 Decentralised Architectures from AI and MAS

In the following, we discuss decentralized architectures from the artificial intelligence (AI) and multi-agent systems (MAS) domains as they could provide inspiration for building self-aware systems. More background on MAS and their progressive transformation into self-aware systems are provided in Chapters 2 and 10.

Decentralized architectures for *rational*[1] systems have been proposed in AI since the early stages of this domain. The aim is to integrate a set of relatively simple modules with limited capabilities into a globally intelligent system (e.g., one that solves complicated problems). In the context of self-aware computing systems, a similar approach may be used to integrate a set of pre-reflective modules into an overall system featuring characteristics of self-awareness.

In general, decentralized architectures consist of a number of independent, specialized modules that execute in parallel and interact with each other. From a self-awareness perspective (cf. Chapter 1), we may distinguish two main cases for such modules. In the first case, modules are rational self-aware entities that pursue individual goals (similar to deliberative agents in MAS), and can have intentional relations with other entities (e.g., cooperation or competition). This case is detailed in Chapter 7 focusing on collectives of self-aware systems. In the second case, modules are non-self-aware (i.e., pre-reflective according to Chapter 3) such that they are not aware of any individual goals and merely react to inputs in predefined ways

[1] Here, *rational* refers to a system's ability to take the best action for optimizing a performance indicator or a goal. In the context of self-aware computing systems, we focus on rational systems that can learn, represent, and reason about knowledge in order to act and achieve their goals.

(i.e., they are reflexive and context-aware [70], but not reflective). Here, we focus on this kind of systems and their architectures proposed in the literature.

Generally, each module in the system is specialized in detecting and reacting to a set of stimuli. It implements a specific function and provides outputs when it is triggered. Modules may be triggered in parallel and some of their outputs may be conflicting. Hence, coordination becomes an important challenge for conflict resolution and synchronization of modules. Numerous coordination approaches have been proposed, which we categorize into coordination based on *shared memory* (e.g., a blackboard) [43, 66], and coordination based on *peer-to-peer communication* [14], sometimes mediated by control modules [23, 31, 60]. Both variants are illustrated in Figure 8.6 and will be discussed with respect to their implications on self-awareness.

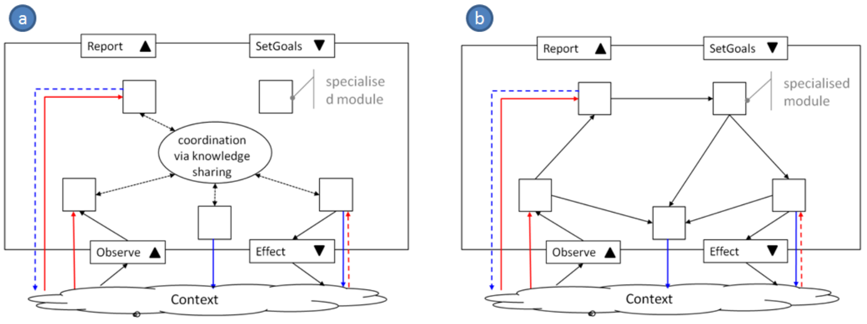

Fig. 8.6: Coordinated Architecture with a) Shared Memory or b) P2P Communication.

In a *blackboard-based system*, incoming events are perceived by dedicated modules and placed onto the blackboard. All modules read from the blackboard and each one reacts to content it is specialized in (e.g., based on event types or patterns). Thus, each module performs its function and writes its ensuing outputs on the blackboard. This, in turn, may cause a new wave of readings and writings by other modules.

Conflict resolution for incompatible module outputs is performed automatically at the blackboard level using various strategies. For instance, ACT* [3, 65] favors the output of modules triggered by conditions with higher matching degrees and specificities, or it considers each module's success history avoiding repetitive activations from the same module, or finally, it prioritizes outputs related to an active goal. In Soar [65], conflicts represent new problems to be solved by the blackboard. Hence, the blackboard does not control the modules directly with command messages but rather indirectly by transforming individual postings into coherent global results, which, in turn, impact the next wave of module activations.

With respect to *self-awareness in blackboard systems*, each module develops its own knowledge about the particular type of problem it is specialized in. Modules only share and combine the knowledge that is relevant to the problem being solved. Hence, the system only acquires knowledge when developing the solution to each problem. The acquired knowledge is not necessarily saved for later use and there are no global learning processes based on it. Each module updates, reasons about, and learns from the knowledge shared via the blackboard. This process is cooperative and opportunistic depending on the shared knowledge. However, modules are not necessarily aware of each-other's participations (i.e., no interaction-awareness) as they simply react to the changes produced by the others on the blackboard.

With respect to goal-awareness, the global goal to achieve (or problem to solve) can be displayed on the blackboard. In this case, the specialized modules that participate in solving the problem can be aware of the goal. In other cases, the system's execution is merely triggered by information obtained from the environment and displayed on the blackboard. Here, the system's overall goal is implicit in the way in which the modules are selected and implemented. Hence, the specialized modules are not goal-aware and they simply react to the state of the blackboard. Considering state-awareness, all modules are aware of the state of the problem being addressed and of the solution being developed since these are displayed on the blackboard.

The main difficulty in adopting a blackboard solution consists in implementing the blackboard involving several complicated functions that are typical for distributed systems (e.g., synchronization, transactions, security, and communication) and problem solving (e.g., conflict resolution). Increasing system scales exacerbates these issues. Moreover, such a solution typically addresses only one problem at a time. On the positive side, once a blackboard is implemented, it is fairly easy to dynamically add, remove, or update specialized modules. Also, specialists can be fairly simple to implement as they are focused on a specific type of problem.

In *peer-to-peer systems*, specialized modules react independently to events by processing them and sending new events through the system. These new events, in turn, trigger a new wave of module reactions. Hence, a collaboration chain of modules is formed in an opportunistic manner starting with modules observing external inputs (or receiving goals), through modules developing a solution progressively, and ending with modules performing external actions. This allows several solutions to be developed in parallel including conflict-resolution mechanisms.

A notable example of this approach is the multi-layered *subsumption architecture* [14] for robotic systems. Each layer consists of a set of well-integrated reflexes and offers functions of increasing complexity, which eliminates the need for an explicit "intelligent" element to mediate between the robot's perception and action. Examples from the autonomic computing domain also introduce special-purpose controllers in the coordination process (e.g., to resolve conflicts [23, 60]). This is useful in open systems, where modules may dynamically join or leave the system, which might even require decentralized coordination control [26, 31].

With respect to *self-awareness in (controlled) peer-to-peer systems*, knowledge about a problem is only transmitted progressively across the modules that contribute to solving the problem. This forms a problem-specific collaboration chain, which is different from a blackboard making the knowledge visible to the entire system.

The coordination logic does not necessarily form a reflective (self-aware) or meta-reflective (meta-self-aware) layer. It can be implemented with reflex modules (i.e., they are only stimulus-aware) that do not learn and reason about the modules they coordinate (e.g., [14]). Of course, a more advanced coordination logic that is goal- and time-aware and includes sophisticated learning and reasoning functions can be envisaged. When peer-to-peer systems are goal-unaware, their goals are implicit in the implementations of the specialized modules and their coordination. Nonetheless, goal-aware systems can also be implemented by providing explicit goals to a centralized coordination controller (e.g., an arbiter or scheduler), which can attempt to achieve the goal by controlling (i.e., triggering or inhibiting) various modules [23, 60]. In such cases, the global system is goal-aware even though the individual modules are not. Finally, a goal-aware system can also be achieved by providing the explicit goals to the specialized modules [27]. Here, both the overall system and (some of) its modules can be considered as goal-aware.

With respect to state-awareness, the specialized modules participating in a collaboration chain are only aware of the state of the system and context by receiving related information from other modules or by monitoring the system and context themselves. Except for the action modules at the end of the chain, none of the modules has a complete view of the solution being developed.

An important difficulty with peer-to-peer approaches relates to asynchronous communication and data-formatting standards. However, existing middleware can be used here to address many of these challenges. As with the blackboard, peer-to-peer solutions allow for the dynamic addition and removal of modules while an important advantage can be the ability to develop alternative solutions in parallel.

Finally, in both of the aforementioned approaches, a system can consist of heterogeneous modules with different self-awareness capabilities and levels (e.g., combining pre-reflective modules with reflective and meta-reflective modules, featuring various learning and reasoning capabilities, and having accumulated knowledge). A major difficulty when constructing and changing such system consists in finding and tuning the correct combination of specialists that together behave coherently to address system-level goals. Also, guaranteeing that such system will behave “as expected” is particularly difficult. In this context, self-awareness capabilities such as learning, reasoning, adaptation, and reporting can be of great help.

In general, the decentralized architectures from AI and MAS can be covered with the architectural concepts introduced in Chapter 5 and they mainly relate to the architectures for collective systems discussed in Chapter 7. The forms, phenomena, and architectural styles of reflection discussed in Chapter 6 usually do not apply to the decentralized architectures from AI and MAS since they only consider individual self-aware systems but not the interaction among multiple systems. In contrast, the concepts introduced in Chapter 7 such as the collective self-awareness with its levels, types of relations/interactions, organizational patterns, and weak/strong self-aware collectives are important aspects of decentralized architectures but they are often not explicitly covered by AI or MAS architectures.

8.3 Architectural Frameworks and Languages

In this section, we discuss specific approaches to develop computing systems that share similarities with self-aware systems. Focusing on architectures in this chapter, these approaches are mainly architectural frameworks and languages and we compare them to the architectural concepts for self-aware computing systems introduced in the previous chapters. Specifically, these specific approaches are reflective architectures, Mechatronic UML, MUSIC, EUREMA, MQuAT, and DML.

8.3.1 Reflective Architectures

In 1987, Maes [58] introduced *computational reflection* as a process of reasoning about and/or acting upon oneself. It is an engineered system's ability to reason about its own resources, capabilities, and limitations in the context of its current operational environment. Reflection capabilities can range from simple, straightforward adjustments of another program's parameters or behaviors (e.g., altering the step size on a numerical process or the application of rules governing which models are used at different stages in a design process) to sophisticated analyses of the system's own reasoning, planning, and decision processes (e.g., noticing when one's approach to a problem is not working and revising a plan).

Reflection processes must include more than the sensing of data, monitoring of an event, or perception of a pattern; they must also have some type of capability to reason about this information and to act upon this reasoning. However, although reflection is more than monitoring, it does not imply that the system is "conscious". Many animals demonstrate self-awareness; not only do they sense their environment but they are able to reason about their capabilities within that environment. For example, when a startled lizard scurries into a crevice, rarely does it try to fit into a hole that is too small for its body. If it is injured or tired, it changes the distance that it attempts to run or leap. This adaptive behavior reveals the ability of the animal system to somehow take into account the current constraints of the environment and of its own body within that environment [6, 7].

In order to bring out the ways in which the self-awareness processes and architectures could enhance and further develop reflective architectures, we will quickly overview one approach to implementing computational reflection and the building of reflection processes. By concentrating on what has been built for the Wrappings approach [55,56], we also want to show how an architecture can have very few fixed relationships and yet still function as an architecture. In fact, because of the lack of fixed relationships with a small amount of guiding infrastructure, the knowledge and the processes used to map resources into appropriate uses for different goals becomes the foundation for the systems flexible ability to reason potentially about any of its own parts and their use in a goal. And hence to have support for the building and utilization of self-models.

Wrappings uses both explicit meta-knowledge and recursively-applied algorithms to recruit and configure resources dynamically to "problems posed" to the system by users, external systems, or the system's own internal processing. The Problem Managers (PMs) algorithms use the Wrappings to choreograph seven major functions: discover, select, assemble, integrate, adapt, explain, and evaluate. Discover programs (or as called in Wrappings, resources) identify new resources that can be inserted into the system for a problem. Selection resources decide which resource(s) should be applied to this problem in this context. Assembly is syntactic integration and these resources help to set up selected resources so that they can pass information or share services. Integration is semantic integration, including constraints on when and why resources should be assembled. Adaptation resources help to adjust or set up a resource for different operational conditions. Explanation resources are more than a simple event history because they provide information on why and what was not selected. Evaluate includes the impact or effectiveness of a given use of this resource. The Wrappings "problem-posing" has many benefits, including separating problems from solution methods and keeping an explicit, analyzable trace of what problems were used to evoke and configure resources. Because all of the resource are wrapped, even the resources that support the wrappings processing, the system is computationally reflective—it can reason about the use of all of its resources. Additional information on the Wrappings approach to reflection and its use in a testbed of robotic cars is found in Chapter 9.

The Wrappings approach considers a more dynamic case to achieve self-awareness than the architectural concepts introduced in Chapters 5, 6 and 7. Therefore, it would be required to enrich the architectural concepts, which apply mainly to snapshots of architectures and do not make the dynamical aspects explicit, with support for dynamic architectures as discussed in Section 8.4 in order to cover the Wrappings approach. The architectural concepts introduced in Chapter 5 can be relevant for Wrappings but they are often not explicitly considered. For instance, the different levels of self-awareness (i.e., the pre-reflective, reflective, and meta-reflective levels) are not explicitly considered, even though they may very well exist in the Wrappings approach. In contrast, the Wrappings approach considers self-models that relate to awareness models and based on the purpose, some of these self-models can be considered as goal models.

For the forms and phenomena of reflection as well as for the architectural styles discussed in Chapter 6 holds that they may temporarily occur in the Wrappings approach, but they are not made explicit and visible at the architectural level. Therefore, local, centralized/hierarchical, and coordinated reflection can be seen as forms of reflection that result from the activities of the problem managers. In specific situations, the resources of a wrapping may be composed such that certain reflection phenomena (e.g., overlapping, stacked, or cyclic reflection) may occur temporarily and certain architectural styles are adopted temporarily. Concerning Chapter 7, the Wrappings approach employs a shared knowledge base such that the concepts of collective self-awareness and weak/strong self-aware collectives are hardly applicable. Likewise, the meta-architecture dimensions (i.e., collective self-awareness levels, types of relations, and organizational patterns) are not suitable for a central knowledge base but they may be applicable with a distributed knowledge base.

8.3.2 Mechatronic UML

Mechatronic UML [16] is a model-driven development approach targeting self-adaptive embedded real-time systems with substantial mechatronic elements. The core building bock for the architectural modeling with Mechatronic UML is the Operator-Controller Module (OCM) [17, 45] depicted in Figure 8.7.

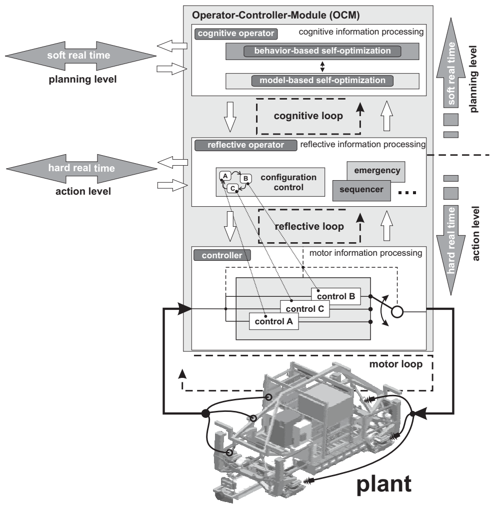

Fig. 8.7: The Operator-Controller Module as the Architectural Building Block [38].

The OCM is separated into a controller, a reflective operator, and a cognitive operator. While the controller could be seen as the pre-reflective core describing the regular operation of the system, the two operators realize different forms of reflective self-awareness. The reflective operator realizes a self-awareness where learning, reasoning, and acting is limited such that hard real-time guarantees can be given. In contrast, the cognitive operator is decoupled from the hard real-time processing and therefore it can use more powerful means for learning, reasoning, and acting. Thus, the two operators separate their tasks based on the required reaction time. The cognitive operator can operate at the same level as the reflective operator by steering the decisions of the reflective operator, but it can also operate at the meta level if it optimizes parameters of the reflective operator.

As illustrated in Figure 8.8, we structure multiple OCMs in a hierarchy to construct complex mechatronic systems (e.g., a shuttle). Moreover, collaborations of

OCMs at the top level of such hierarchies are used to flexibly compose systems (e.g., multiple shuttles to a convoy). The architectural modeling is supported by hybrid UML components [36] and hybrid, real-time statecharts extending UML state machines [17]. The modeling is complemented with a code-generation scheme that derives an implementation with guarantees that the regular operation and the reconfiguration steps stay within the real-time bounds specified in the models [16,18,37]. Therefore, an assurance scheme has been developed for verification [38].

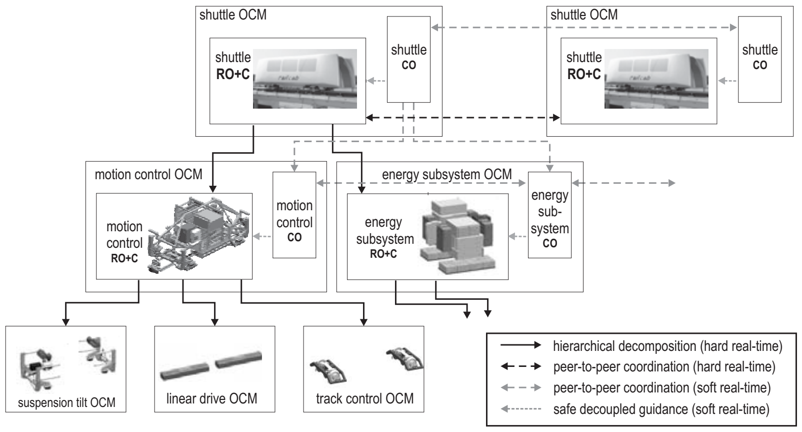

Fig. 8.8: Collaborating Hierarchy of OCMs [38].

Compared to self-aware computing, Mechatronic UML is tailored for the particular domain of self-adaptive embedded real-time systems with mechatronic elements. The supported architectural concepts indicate that a layered architectural style fits well to the high quality and safety required in this domain. An OCM can be seen as a local-reflective self-awareness module as discussed in Chapter 6 since it has a local feedback loop realized by its two operators. However, the OCM does not make the awareness and goal models as well as the awareness/expression links explicit (cf. Chapter 5). Instead, the reflective operator manages the controller by defining a particular configuration for each operation mode while the reflection by the cognitive operator is not specified at all in the architecture. Further supporting hierarchies of OCMs (see Figure 8.8), Mechatronic UML adopts a mixture of local and hierarchical reflection (cf. Chapter 6). Moreover, hierarchies of OCMs can be composed by collaborations (see Figure 8.8), which partially addresses collective systems as discussed in Chapter 7.

In general, Mechatronic UML suggest a more specific architectural design than the concepts introduced in Chapter 6. Thus, the pre-reflective and reflective levels of self-awareness are covered by the controller respectively by the reflective and cognitive operators of an OCM. In contrast, the OCM hierarchies realize hierarchical control and not meta-reflective self-awareness such that the meta-reflective level is not covered. Based on the adopted architecture implied by the OCMs and their hierarchical composition, Mechatronic UML employs local and hierarchical reflection while it excludes the phenomena of reflection, that is, overlapping, stacked, and cyclic reflection (cf. Chapter 6). Concerning the architectural styles discussed in Chapter 6, Mechatronic UML follows the hierarchical control approach. Concerning the specific concepts for collective self-aware systems introduced in Chapter 7, Mechatronic UML only supports the organizational patterns, particularly in the form of collaborations that combine mechatronic systems (see Figure 8.8).

8.3.3 MUSIC

The main goal of MUSIC was to simplify the development of adaptive applications that operate in open and dynamic computing environments and adapt seamlessly and without user intervention in reaction to context changes. The main innovations of MUSIC were a comprehensive development framework that consists of a model-driven development methodology with a tool chain for self-adaptive, context-aware applications as well as a corresponding extensible context management and adaptation middleware supporting the development [30, 42]. Context is understood in a broad sense as any information about the user needs and the application execution environment that may vary dynamically and impact the applications. Context parameters can be monitored using appropriate hardware and software mechanisms, called context sensors. MUSIC supports a variety of adaptation mechanisms such as changing configuration and application parameters, replacing components and service bindings, and redeploying components on the distributed infrastructure [33,69].

In the following, we provide an overview of the main ingredients of MUSIC and an attempt to position MUSIC with respect to the discussion of self-awareness.

To meet the challenges of dynamically adaptive, context-aware applications on mobile devices, MUSIC developed

- an adaptation modeling language which separates self-adaptation and business logic concerns to avoid the surge in complexity;
- generic, reusable middleware components which automate context management and application adaptation based on design-time models that are translated into run-time representations capturing the adaptation options and goals;
- tools which support the development of design models and their transformation into run-time representations for the MUSIC middleware.

An overview of the MUSIC framework is given in Figure 8.9. The middleware implements a control loop similar to the MAPE-K loop (see Section 8.2.1). It monitors the relevant context sensors, and when significant changes are detected, it triggers a planning process to decide if adaptation is necessary. When this is the case, the planning process finds a new configuration that fits the current context better than the currently used one, and triggers the adaptation of the running application. To do this, the middleware relies on an annotated QoS-aware architecture model of the application which is available at runtime. The model specifies the application's

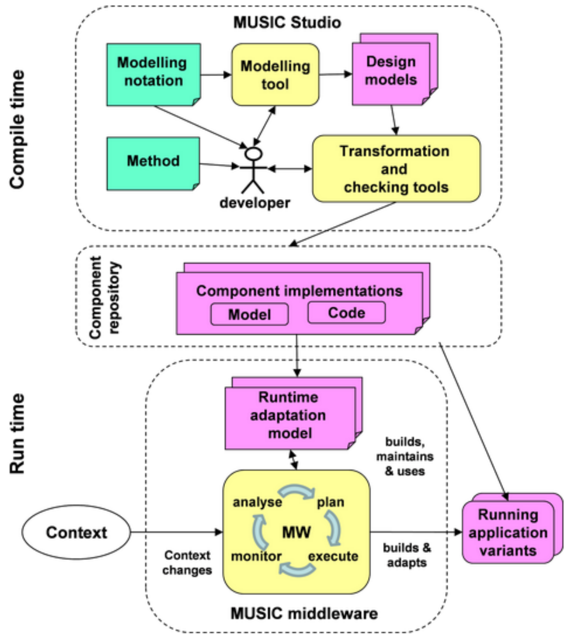

Fig. 8.9: Overview of the MUSIC Development Framework.

dependencies on context information, its adaptation capabilities, and the objective function for adaptation reasoning. The model corresponds to the “Knowledge” component of the MAPE-K loop. The planning process of MUSIC evaluates the utility of alternative configurations, selects the most suitable one for the current context (i.e., the one with the highest utility for the current context which does not violate any resource constraints), and triggers the application adaptation accordingly.

An overview of the MUSIC middleware and its adaptation loop is depicted in Figure 8.10. It shows the main components of the middleware, represented as rounded rectangles, and their mapping to the elements of MAPE-K. The managed system includes the applications, as indicated by the arrows connecting the MAPE-K loop to the application layer, as well as the underlying computing and communication infrastructure. A similar mapping can be done for the LRA-M loop (see Chapter 1) although MUSIC does not support any learning since it controls the applications based on the QoS-aware architecture model resulting from the development.

The context middleware of MUSIC is responsible for monitoring and analyzing context changes and for providing access to context information. It encapsulates the diversity of context information and maintains the context model storing and providing uniform access both to the current state and history of context data. Moreover, context reasoners may be added to the middleware to perform reasoning on the lower-level context data and derive higher-level information. For example, a

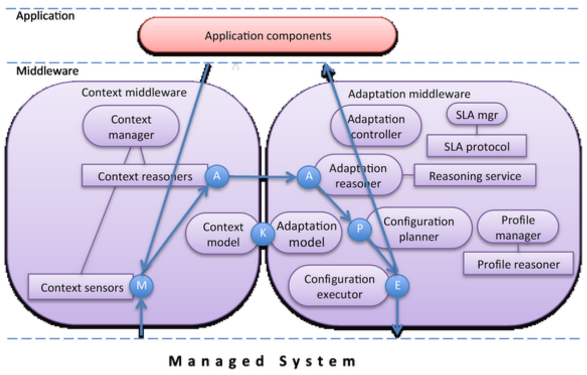

Fig. 8.10: Overview of the MUSIC Middleware.

reasoner could trigger adaptation planning only when there is major change in the long-term trend of some context parameter rather than on every small value change. With respect to the MAPE-K (LRA-M) loop, the middleware covers the monitoring (collecting empirical data), partially the analyzing (reasoning) of the context, and partially the knowledge (by maintaining context awareness models).

Thus, the very basic elements of a self-aware system are present in the MUSIC middleware. Even a learning component was considered conceptually in the design of the middleware but it was not implemented due to project constraints. Despite the support for multi-party adaptation [34], the MUSIC solution focuses on adaptation of applications on a single mobile computing device. The MUSIC framework misses a notion of collective or group awareness. This will be a target of our future research.

In general, MUSIC provides a complete model-driven approach rather than a framework focusing on architectural concepts as introduced in Chapter 5. Therefore, the pre-reflective and reflective levels of self-awareness are not explicitly considered but they result from the embedded support for self- and context-awareness. In contrast, meta-reflective self-awareness is not covered by MUSIC. However, the QoS-aware architecture and adaptation model of MUSIC can be seen as an awareness and goal model as it captures the system, context, options for adaptation, and the operationalized goals. In contrast, the awareness and expression links with their span and scope are not addressed explicitly but rather embedded in the model.

Using such a model, MUSIC employs centralized reflection and excludes the phenomena of overlapping, stacked, and cyclic reflection (cf. Chapter 6). Employing a single reflection step, the architectural styles discussed in Chapter 6 are not supported by MUSIC. Though MUSIC aims for distributed systems, it realizes a single, centralized reflection step that does not address the architectural concepts for collective self-aware computing systems discussed in Chapter 7.

8.3.4 ExecUtable RuntimE MegAmodels (EUREMA)

EUREMA (ExecUtable RuntimE MegAmodels) [76] is a seamless model-driven engineering approach for the specification and execution of feedback loops for self-adaptive software adopting the external approach (cf. [71]) and a layered architecture. The EUREMA modeling language is used to specify feedback loops, which makes them explicit in the design. The resulting models are kept alive at runtime and the EUREMA interpreter directly executes these models to run feedback loops.

The specification is done using two types of diagrams as shown in Figure 8.11 for a self-healing example. A feedback loop diagram (FLD) specifies a feedback loop with its activities (also called model operations such as Update and Repair), control flow among activities, and runtime models. Such models either represent the lower-layer entity to be adapted to achieve model-based adaptation (e.g., the Architectural Model), or they specify the activities (e.g., Repair strategies determining how the system should be healed if failures occur). An activity and a model are linked in the FLD if the activity accesses the model, for instance, to read, write, or annotate it. A feedback loop can be modularly specified by multiple FLDs. For instance, the analyze activity is defined in a distinct FLD (not shown here) that is invoked by the Self-repair FLD using a complex model operation (cf. left-hand side of Figure 8.11).

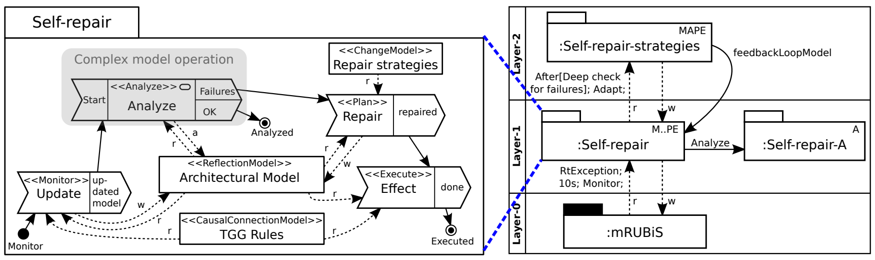

Fig. 8.11: Feedback Loop Diagram (FLD) and Layer Diagram (LD) (cf. [76]).

A layer diagram (LD)—as shown on the right-hand side of Figure 8.11—then structures modules, which are instances of feedback loops as defined in FLDs, in a layered architecture. The lowest layer contains the managed system, in this case the online marketplace *mRUBiS*, that is not specified by EUREMA and thus considered as a black box. This system is adapted by a self-repair feedback loop realized by the two modules :Self-repair and :Self-repair-A. While the latter module is invoked by the former module, the execution of the former module is triggered starting in its initial state Monitor if :mRUBiS emits an RtException event and if more than ten seconds have elapsed since its previous run.

Moreover, EUREMA supports the layering of feedback loops. For instance, the Repair strategies used by the Self-repair feedback loop will typically not be able to heal all kinds of failures since we cannot anticipate all of them during development. To adjust these strategies, we employ another feedback loop on top (see :Self-repair-strategies in the LD while we omit the related FLD here due to space restrictions) that provides new strategies to the self-repair feedback loop if the currently used one are not able to heal the failures. Hence, the higher-layer feedback loop reflects on the lower-layer feedback loop to analyze its performance and if needed, to adapt the lower-layer feedback loop to improve the performance. A benefit of EUREMA is that the self-repair feedback loop is specified by a model (see the FLD on the left-hand side of Figure 8.11) that is kept alive at runtime such that the higher-layer feedback loop can directly use this model to analyze and adapt the self-repair feedback loop. Nevertheless, it is also possible to define activities that maintain a distinct runtime model representing the self-repair feedback loop.

Concerning self-aware systems (cf. Chapter 5), EUREMA supports feedback loops (reflective layers) that reflect on a managed system (pre-reflective layer) by using abstract runtime models that are causally connected to the system. Such runtime models are similar to awareness models representing the pre-reflective layer and used by the reflective layer. A major difference is that EUREMA so far does not assume any learning processes but being an open framework, EUREMA can integrate arbitrary (model-driven) processes and techniques to realize the activities of a feedback loop such as learning, reasoning, and acting of the LRA-M loop.

Moreover, EUREMA explicitly supports meta-self-awareness by layering feedback loops on top of each others. Thereby, the specification, that is, the FLD model of a feedback loop can be directly used as the awareness model by the higher-layer feedback loop for learning, reasoning, and adaptation. Thus, EUREMA supports coordinated reflection and in particular, centralized reflection (cf. Chapter 6). Moreover, it addresses overlapping and stacked reflections but neglects cyclic reflections. With respect to the architectural styles (cf. Chapter 6), EUREMA fully supports the stacked external approach and partially the hierarchical and acyclic external approach. In contrast, the internal approach is not supported at all since it contradicts the layering of feedback loops. Finally, since EUREMA does not restrict the number of layers and supports adding/removing layers at runtime, the concept of dynamic layers leverages the long-term evolution of the system. Using (temporarily) an additional top-most layer, adaptations for maintenance can be performed [76, 77].

Recently, EUREMA has been extended to support distributed systems. The extension introduces collaborations to EUREMA, which describe the interactions among multiple feedback loops [79]. These collaborations support flexible means to coordinate multiple feedback loops and their runtime models as discussed in Chapter 7 for collectives of self-aware systems and awareness models.

8.3.5 Multi-Quality Auto-Tuning (MQuAT)

Multi-Quality Auto-Tuning (MQuAT) is a model-driven approach to develop and operate self-optimizing software systems using quality-of-service (QoS) contracts and models@run.time [39]. Figure 8.12 depicts an overview of MQuAT.

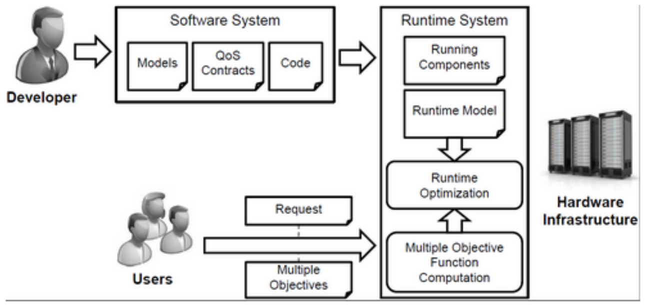

Fig. 8.12: Overview of Multi-Quality Auto Tuning.

The principle idea of MQuAT is to develop self-optimizing software in terms of individual software components (i.e., elements of reuse, which explicitly specify what they require and provide). Each component realizes a particular task (e.g., sorting, encryption, or compression) and can have multiple implementations, all providing the same functionality but differing in their non-functional properties (e.g., the time or energy required to perform the task). The non-functional behavior of each implementation is characterized by the developer with QoS contracts [40]. As concrete statements, for instance, about the execution time of a component, depend on the used hardware and the respective input data, which are both unknown during development, these contracts are refined at runtime by benchmarking. That is, the developer has to provide a benchmark for each implementation, which is used at runtime to derive a hardware-specific variant of the contract. When a user poses a request to the system, the input data is known and, hence, the hardware-specific contracts can be refined further to be request-specific. This version of the contracts now contains concrete statements about the non-functional properties of the respective implementation (e.g., the sorting task will take 30ms on this hardware for this request). Based on these contracts and a runtime model representing the current state of the system, MQuAT generates an optimization problem, which is solved using standard solvers. The resulting solution is translated to a reconfiguration script [41].

This process is depicted in Figure 8.13 and denotes a particular variant of a self-aware computing system adhering to the models@run.time paradigm and the LRA-M loop as introduced in Chapter 1. The running system is learning by observing itself and its environment with the aim to keep the runtime model of the system in sync with the actual system. This runtime model is reasoned upon by comparing it to QoS contracts to identify adaptation needs. The reasoning is realized by a model-to-text transformation of the runtime model, the design-time models, and QoS contracts to some optimization language (e.g., integer linear programming). Finally, if a better configuration is found, the system acts by reconfiguring itself.

In terms of the concepts for self-aware computing (see Chapter 5), MQuAT enables the development of reflective systems by using an awareness model. If the scheme is applied multiple times, MQuAT can cover the meta-reflective case, too.

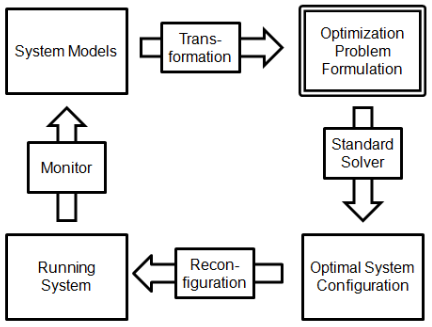

Fig. 8.13: The Role of Models@run.time in MQuAT.

The awareness model is kept synchronous to its underlying system by collecting empirical data (benchmarking). MQuAT offers a simple variant of goal models in the form QoS objectives (e.g., minimize energy consumption). Awareness and expression links are not explicitly represented. However, the accuracy of knowledge covered in an awareness model is explicitly captured by the coefficient of determination (R^2). In terms of the concepts introduced in Chapter 6, MQuAT uses coordinated reflection and follows the hierarchical external style. As MQuAT does not aim to support the integration of multiple systems, the concepts introduced in Chapter 7 are not applicable to it.

8.3.6 Descartes Modeling Language (DML)

The Descartes Modeling Language (DML)[2] is an architecture-level modeling language for quality-of-service (QoS) and resource management of modern dynamic IT systems and infrastructures [53]. DML is designed to serve as a basis for self-aware systems management during operation, ensuring that system QoS requirements are continuously satisfied while infrastructure resources are utilized as efficiently as possible. The term QoS refers here to performance (response time, throughput, scalability, and efficiency) and dependability (availability, reliability, and security). The current version of DML is focused on performance and availability, however, the language itself is generic and intended to eventually support further QoS properties.

DML has a modular structure and is provided as a set of meta-models for describing the resource landscape, the application architecture, the adaptation points, and adaptation processes of a system. The meta-models can be used both in online and offline settings for performance prediction and proactive system reconfiguration.

The designers of DML advocate a holistic model-based approach where systems are designed from the ground up with built-in self-reflective and self-predictive capabilities, encapsulated in the form of online system architecture models. The latter

[2] `http://descartes.tools/dml` (formerly also known as Descartes Meta-Model (DMM))

are assumed to capture the relevant influences (with respect to the system's operational goals) of the system's software architecture, its configuration, its usage profile, and its execution environment (e.g., physical hardware, virtualization, and middleware). The online models are also assumed to explicitly capture the system's goals and policies (e.g., QoS requirements, service level agreements, efficiency targets) as well as the system's adaptation space, adaptation strategies and processes. Figure 8.14 presents the DML vision of a self-aware system adaptation loop, based on the MAPE-K loop, in combination with the online models used to guide the system adaptation at run-time. The four phases of the loop are as follows:

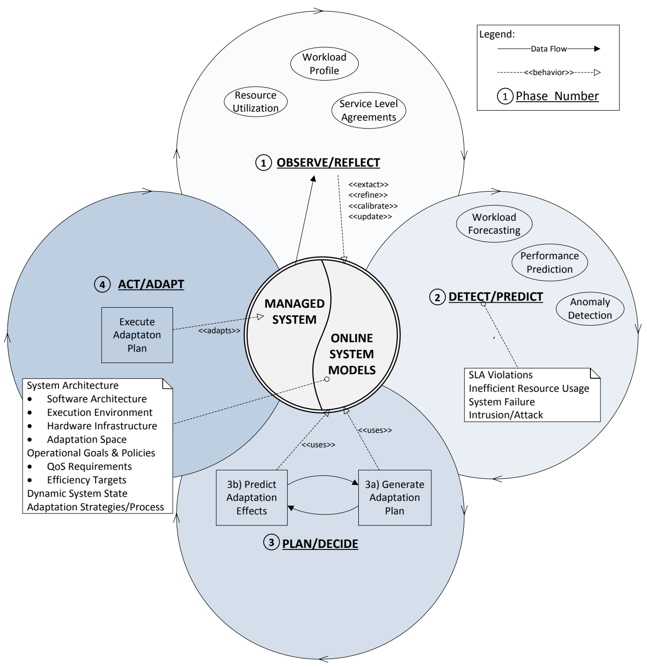

Fig. 8.14: DML System Adaptation Loop.

Phase 1 (Observe/Reflect): In this phase, the managed system is observed and monitoring data is collected and used to extract, refine, calibrate, and continuously update the online system models, reflecting the relevant influences that need to be captured in order to realize the self-predictive property with respect to the system's operational goals. Here, expertise from software engineering, systems modeling and analysis, as well as machine learning, is required for the automatic extraction, refinement and calibration of the online models based on the run-time observations.

Phase 2 (Detect/Predict): In this phase, the monitoring data and online models are used to analyze the current state of the system in order to detect or predict problems such as SLA violations, inefficient resource usage, system failures, network attacks, and so on. Workload forecasting combined with performance prediction and anomaly detection techniques can be used to predict the impact of changes in the environment (e.g., varying system workloads) and anticipate problems before they have actually occurred. Here, expertise from systems modeling, simulation, and analysis, as well as autonomic computing and artificial intelligence, is required to detect and predict problems at different time scales during operation.

Phase 3 (Plan/Decide): In this phase, the online system models are used to find an adequate solution to a detected or predicted problem by adapting the system at run-time. Two steps are executed iteratively in this phase: i) generation of an adaptation plan, and ii) prediction of the adaptation effects. In the first step, a candidate adaptation plan is generated based on the online models that capture the system adaptation strategies, taking into account the urgency of the problem that needs to be resolved. In the second step, the effects of the considered possible adaptation plan are predicted, again by means of the online models. The two steps are repeated until an adequate adaptation plan is found that would successfully resolve the detected or predicted problem. Here, expertise from systems modeling, simulation, and analysis, as well as autonomic computing, artificial intelligence, and data center resource management, is required to implement predictable adaptation processes.

Phase 4 (Act/Adapt): In this phase, the selected adaptation plan is applied on the real system at run-time. The actuators provided by the system are used to execute the individual adaptation actions captured in the adaptation plan. Here, expertise from data center resource management (virtualization, cluster, grid and cloud computing), distributed systems, and autonomic computing, is required to execute adaptation processes in an efficient and timely manner.

DML provides a language and a reference architecture for self-aware IT systems and infrastructures where self-awareness is focused on QoS and resource management. An overview of DML can be found in [53], a detailed specification in [52].

Concerning the levels of self-awareness introduced in Chapter 3, DML can be used to design self-awareness mechanisms at all three levels: pre-reflective, reflective, and meta-reflective. The online system architecture models can be seen as awareness models that are learned based on static information (model skeletons) provided at system design-time and empirical data collected at run-time (cf. Chapter 5). Operationalized goals are captured using DML's Strategies/Tactics/Actions (S/T/A) formalism, which includes a basic form of goal models. Awareness and expression links are not explicitly represented.

In terms of the concepts introduced in Chapter 6, DML can be used to design systems with hierarchical and centralized reflection, however, it currently does not explicitly support coordinated reflection. Overlapping reflection is directly supported by using alternative tactics for the same adaptation strategy; the stacked and cyclic reflection phenomena are not considered explicitly, however, they can manifest themselves when applying DML to manage different subsystems and/or layers of the system architecture. In terms of the architectural styles, DML assumes the generalized external approach, while not prescribing a particular style that has to be followed. Finally, as DML is not designed to support integrating independent systems, the concepts introduced in Chapter 7 are not applicable to it.

8.4 Open Challenges

Based on the previous sections, we now discuss open challenges for the fundamental architectural concepts introduced in Chapter 5 as well as for the generic architectures of individual and collective self-aware computing systems introduced in Chapters 6 and 7. We identified different classes of challenges such that we grouped them as follows:

- self-awareness and self-expression,
- encapsulation for self-awareness and self-expression,
- interference, separation, and emergence,
- dynamic architectures, and
- other challenges.

(1) Self-Awareness and Self-Expression

The first group of challenges relates to self-awareness and self-expression. It concerns the complex and potentially cyclic graphs resulting from the awareness and expression links, the support for detailed semantics of such links (cf. Chapter 3), the static knowledge supporting self-awareness, and finally, the uncertainty as a main driver for inaccurate and imprecise self-awareness.

Complex and cyclic awareness and expression link graphs: The notation we propose in Chapter 5 allows us to describe direct and indirect awareness and expression links. The graph resulting from the combination of such links can become complex when considering large systems and advanced forms of self-awareness (e.g., architectural elements acting as spans and scopes in various self-awareness relationships that differ in the type of awareness such as stimulus-, interaction-, time-, or goal-awareness). Concepts to describe such complex graphs of awareness and expression links and to understand their implications are currently missing.

Describing the awareness and expression links as a graph supports studying the dependencies between the individual spans and scopes of the awareness and expression. As known from the literature on self-adaptive systems (cf. [15, 63, 73]), not only the individual awareness or expression links but additionally their interplay that form feedback loops as well as the interferences of these loops are main challenges for the design of self-adaptive systems. Therefore, means to describe and analyze the interplay and interferences between awareness and expression links are required. In this context, cyclic relationships are a particular challenge, for instance, to guarantee termination, convergence, and an appropriate outcome of the learning, reasoning, and acting processes along the expression and awareness links.

A related challenge is which *detailed semantics of the awareness and expression links* should be covered at the architectural level to achieve separation of concerns. We address the different dimensions of awareness such as time-, interaction-, or state-awareness (cf. Chapter 3) with stereotypes attached to the awareness links (cf. Chapter 5). However, to address the subtle aspects of such dimensions and to enable their analysis, they have to be treated in a much more explicit manner such as describing the specific semantics of the dimensions in a particular case. The same holds for the expression links that only represent the fact that a span impacts a scope without substantiating their semantics such as the extend or kind of impact.

With our notation, we focused on how architectural elements dynamically obtain knowledge by collecting empirical data as well as learning and reasoning about that data (cf. LRA-M loop modeled in Chapter 5). However, we also observed the need to capture *static knowledge* that has been established at design time, embedded into modules, and exploited at runtime (see the discussion of capturing the feed-forward and adaptive control schemes in Chapter 6). Such static knowledge supports self-awareness and it might be enriched by dynamic knowledge at runtime. Hence, means to describe static knowledge along awareness and expression links is required to distinguish it from dynamic knowledge. This further requires means to describe and analyze the composition of static and dynamic knowledge.

Regardless of static or dynamic knowledge, ruling the *uncertainty* that can be tolerated for the runtime models and other elements of the architecture is a major design issue (cf. [35]). Currently, the architectural descriptions do not show the degree of uncertainty that exists in the system as well as the extent to which processes are used to counteract the uncertainty. This calls for specifying or measuring the accuracy and precision of empirical data, awareness models, and actions, which should be supported by reflection interfaces (cf. Chapter 6) as well as how processes address uncertainty in this context. Such information is the basis for determining the uncertainty in the self-awareness of the system, which can be a critical dimension in the design space for self-aware computing systems.

(2) Encapsulation for Self-Awareness and Self-Expression

The second group of challenges addresses the trade-off between the power of self-awareness and self-expression and the need for encapsulation. While strong encapsulation demands explicit and restrictive interfaces, powerful self-awareness and self-expression require generic interfaces for computational reflection with as little encapsulation as possible. Furthermore, concepts for reflection interfaces are required that are able to bridge the abstraction gap between the span and the scope.

Encapsulation and computational reflection: The proposed notation allows us to describe the encapsulation of modules by means of ports and interfaces as well as their links to elements contained in these modules such as awareness models, empirical data, or processes (see Chapter 5). As an alternative, we can model the direct access to a module's internals, which denotes a form of computational reflection. However, in practice a compromise between explicitly encoded access through ports and computation reflection may be needed. Our notation primarily supports the explicitly encoded access while it lacks concepts to make various forms of computational reflection (e.g., declarative and procedural reflection [58]) explicit.

In this context, finding an appropriate notion for a *reflection interface* (cf. Chapter 6) that separates different levels of reflection such as the pre-reflective and reflective levels is a crucial challenge. We either have local reflection interfaces where a single interface encapsulates only one monolithic module, which results in a reasonably low complexity for the interface and its realization. However, if a reflection interface covers a group of modules, it becomes more complicated since multiple modules have to be jointly observed and adapted while they and their interconnections may dynamically change (e.g., considering a flexible collaboration of modules). Nevertheless, the increasing complexity of realizing the reflection interface shields the reflecting elements from the complexity of the reflected elements by using the interface. Hence, the design of a reflection interface impacts the complexity of elements performing the reflection and hence establishing self-awareness.

Another dimension that has to be considered for a reflection interface is the degree of *observability* and *controllability*. Observability determines the scope of an awareness link, that is, what can be observed by a span. Controllability determines the action and influence scope of an expression link, that is, what can be directly or indirectly changed by a span. Consequently, this dimension of a reflection interface has to be properly chosen as it determines the system's self-awareness and self-expression capabilities that should be sufficiently powerful to achieve the goals.

To develop a notion of a reflection interface, a promising research direction can be the use of models@run.time [12] serving as interfaces to a managed system.

Finally, the design of the reflection interfaces and the restriction implied by the design may limit the evolution of the system as any reflection can only happen within the anticipated observability and controllability. This might require means to support dynamically adjusting the reflection interfaces to increase (or in certain situation even to decrease) the degree of observability and controllability.

(3) Interference, Separation, and Emergence

This group of challenges targets the existence of multiple awareness and expression links and raises questions of the granularity of self-awareness and self-expression as well as of their separation and dependencies (e.g., interferences). While the separation of different self-awareness concerns such as self-healing or self-configuration is a general problem, the following challenges additionally arise for collectives of self-aware systems: How collective self-awareness can emerge from individual self-awareness? How changes at the individual level can impact the collective level? How concrete architectures for collective self-awareness can be developed?

Achieving a suitable *separation of self-awareness concerns* in an architecture, which reduces the complexity of the problem for individual self-aware systems, is challenging. It is well known from control theory that multiple feedback loops can only be employed if a careful analysis ensures that they are sufficiently decoupled. For self-adaptive software, it has been advocated that feedback loops should be treated as first class entities in the architectural design and analysis [15, 73]. Likewise, we know from the agent domain that conflicting goals require our attention when designing the agents. Consequently, when multiple awareness and expression links overlap or compete, a suitable approach to study such phenomena at the architectural level is required. However, such an approach is lacking although the concepts introduced in Chapter 5 already go beyond the state of the art (e.g., [44]).

Particularly for emergent architectures, means to ensure the runtime detection and resolution of conflicts are required as such conflicts cannot be necessarily anticipated and addressed by construction. Here, an additional problem may occur since not only feedback loops may compete but cyclic networks of awareness and expression links may require means to ensure convergent behavior achieving the goals. Addressing such problems at the architectural level is further complicated as *abstraction* may lead to spurious cycles (cf. Chapter 5 and Chapter 6), which requires to identify the right level of detail to identify and understand such problems.

Another challenge is to achieve *collective self-awareness from individual self-awareness* of subsystems. Here, the main question is how the self-awareness at the collective level can emerge from behavior and self-awareness at the individual level. A related challenge is to identify and control the *impact of changes in individual self-awareness on collective self-awareness*. How can such effects be addressed at the architectural level such that the envisioned self-awareness at the collective level is achieved. Finally, it remains an overall challenge of how to translate the answers to the above questions into *concrete architectures for collectives of self-aware systems* that feature the targeted self-awareness levels and can meet their high-level goals.

(4) Dynamic Architectures

Challenges that apply to individual and collective self-aware systems concern the dynamic architecture, that is, the architecture may dynamically change either as a result of external changes or of self-adaptation/self-expression. A first challenge is that we lack a notation that is able to capture the architectural dynamics in a comprehensible manner. Furthermore, it is major challenge to rule the dynamics at the level of individual systems as well as at the level of collectives such that the self-awareness and self-expression mechanisms are able to cope with the dynamics. Moreover, the coherent behavior of individual systems in a collective is crucial since each system may evolve independently without any joint, overall management.

Capturing Dynamics: The proposed notation allows us to model static or snapshots of architectures that show the static existence of self-awareness. However, an architecture can be dynamic by supporting structural changes [59]. The notation proposed in Chapter 5 lacks concepts to make the dynamics and the variability of the architecture explicit. Moreover, it lacks concepts to describe and understand the impact of such dynamics on (the existence and evolution of) self-awareness.

Therefore, the challenge is to cover the *architectural dynamics of an individual self-aware system*, which concern the establishing of self-awareness, for instance, when pre-reflective or reflective modules dynamically join or leave the system. In this context, the generalized external approach that separates the reflecting from the reflected modules (cf. Chapter 6) seems to be better suited for handling dynamic cases since it can employ centralized reflection to learn global models of the pre-reflective layer and hence, it can have global knowledge of this layer and observe its dynamic changes. In contrast, the generalized internal approaches often relies on a pre-coded wiring among the modules, which might limit the dynamics to local changes. For instance, an architecture adopting the generalized external approach can address such dynamics by employing architectural awareness models of each pre-reflective, reflective, meta-reflective, meta-meta-reflective etc. layer (cf. [76]). In addition, to control the dynamics of each layer, behavior models specifying the architectural dynamics seem to be necessary (e.g., [16]). However, it is not clear whether the ideas of such approaches are generically applicable or whether they are apply only to specific styles and domains with their implied architectural restrictions such as a strict layering of modules. Generalizing and applying such ideas to other styles and domains are worth to be investigated.

Moreover, the *variability* in the architecture of self-aware systems should be made explicit to span the solution space, which can be explored and even extended by self-awareness and self-expression. However, this inherent architectural aspect is not addressed by the concepts we proposed in the previous chapters. An extension to these concepts is needed and feature models can be a starting point for that.

In addition to a single, dynamically changing self-aware systems, the *architectural dynamics of a collective self-aware system* is a big challenge including questions of how to dynamically adapt and evolve such a collective to the changing goals, self-awareness levels, internal resources, and execution environment, and of how to *ensure a coherent behavior of collectives* when individual subsystems change.

(5) Other Challenges

Finally, there are aspects and related challenges that we do not address at all with the architectural concepts discussed in Chapter 5, 6, and 7. For instance, we neglected the aspect of integrating legacy software, enabling reuse, and supporting particular domains with their specific needs and limitations.

The integration of *legacy software* with its architecture respectively used middleware/framework is challenging because of having to provide reflections on it. The proposed architectural concepts either consider explicit reflection interfaces or the direct access to the internals of a modules to realize computational reflection. Integrating a legacy software into such an architecture imposes the challenge that the architecture of the legacy software might adopt a different style and does not provide a compatible mechanisms to reflect upon it.

Furthermore, to enable *reuse*, some approaches advocate using a generic adaptation mechanism offered by the middleware and only encode the possible solution space and optimization criteria (e.g., MUSIC as discussed in Section 8.3.3). How can we cover such approaches at the architectural level—to benefit from the reuse—without embedding the infrastructure/middleware in the architecture. On the other hand, approaches such as Rainbow [32] advocate architectural frameworks that predefine large fragments of the architecture and thus restrict the design space of the system. What is the appropriate granularity of reuse that still provides the freedom to explore architectural choices for individual systems depending on the domain.

Finally, *domain-specific extensions* are required to address the additional constraints and requirements imposed by a particular domain. To realize self-awareness for cyber-physical systems (cf. [16]), sensor networks, or internet of things scenarios, typical domain-specific aspects are predictable real-time behavior, safety issues, or severe resource constraints. For enterprise application, besides the computing part also humans in the loop play an important role. How can we develop and integrate such domain-specific extensions into a generic framework for self-aware systems.

8.5 Conclusion

In this chapter, we reviewed the state of the art concerning self-aware computing systems with the particular focus on the software architecture. Particularly, we compared state-of-the-art approaches with the fundamental architectural concepts for self-aware systems (cf. Chapter 5, the generic architectures for individual self-aware systems (cf. Chapter 6), and the generic architectures for collectives of self-aware systems (cf. Chapter 7). The approaches we included in the comparison are either reference architectures or architectural frameworks and languages for software systems that share similarities with self-aware computing systems.

The comparison of the proposed architectural concepts with the state-of-the-art approaches demonstrated that the concepts are helpful and oftentimes allows us to make explicit or at least to emphasize specific aspects relevant to self-awareness. Existing reference architectures or approaches often only support such aspects implicitly, for instance, by assumptions. For example, the MAPE-K reference architecture keeps the knowledge part abstract while we detailed this part. More specifically, we made the awareness models explicit and further linked these models to spans and scopes to denote the self-awareness and self-expression in the architecture.

Moreover, in contrast to the reference architectures and approaches that usually suggest a rather specific architectural style, the proposed concepts support modeling architectures of a broader spectrum ranging from layered architectures with a single centralized self-awareness module to those architectures where self-awareness emerges from the coupled operation of several modules at the level of an individual system up to the level of collectives of systems.

Based on the comparison, we identified open challenges. These challenges showed that the proposed concepts for awareness and expression links are too limited since they do not address analyzing cyclic links, specifying detailed semantics of such links, as well as capturing static knowledge and uncertainty. Furthermore, additional challenges relate to the trade-off between the power of self-awareness and self-expression and the need for encapsulation for reflection as well as to the design and analysis of separating self-awareness and self-awareness concerns. Another group of challenges pointed out the need to address architectural dynamics that either result from external changes or self-expression. Finally, we sketched aspects and challenges that we completely neglected with the proposed architectural concepts. These aspects are legacy software, reuse, and domain-specific extensions for self-aware computing systems.

As future directions, we suggest approaching the open challenges as well as conducting studies to obtain more practical experience with developing architectures for individual and collective self-aware computing systems.

Acknowledgment

This chapter is the result of stimulating discussions among the authors and other participants, especially Paola Inverardi and Peter Lewis, during the seminar on Model-driven Algorithms and Architectures for Self-Aware Computing Systems at Schloss Dagstuhl in January 2015 (`http://www.dagstuhl.de/15041`).

References

1. Anant Agarwal and Bill Harrod. Organic computing. Technical Report White paper, MIT and DARPA, 2006.
2. Anant Agarwal, Jason Miller, Jonathan Eastep, David Wentziaff, and Harshad Kasture. Self-aware computing. Technical Report AFRL-RI-RS-TR-2009-161, MIT, 2009.
3. John R. Anderson, editor. *The Architecture of Cognition*. Harrvard University Press, 1983.
4. Uwe Aßmann, Sebastian Götz, Jean-Marc Jézéquel, Brice Morin, and Mario Trapp. A Reference Architecture and Roadmap for Models@run.time Systems. In Nelly Bencomo, Robert France, Betty H. C. Cheng, and Uwe Aßmann, editors, *Models@run.time: Foundations, Applications, and Roadmaps*, pages 1–18. Springer, 2014.
5. Ozalp Babaoglu, Mark Jelasity, Alberto Montresor, Christof Fetzer, Stefano Leonardi, Aad van Moorsel, and Maarten van Steen, editors. *Self-star Properties in Complex Information Systems: Conceptual and Practical Foundations*, volume 3460 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2005.
6. Kirstie L. Bellman and Christopher Landauer. Towards an Integration Science. *Journal of Mathematical Analysis and Applications*, 249(1):3–31, 2000.
7. Kirstie L. Bellman, Christopher Landauer, and Phyllis R. Nelson. Systems Engineering for Organic Computing: The Challenge of Shared Design and Control between OC Systems and their Human Engineers. In *Organic Computing*, pages 25–80. Springer, 2008.
8. Nelly Bencomo. Quantun: Quantification of uncertainty for the reassessment of requirements. In *Proceedings of 23rd International Conference on Requirements Engineering (RE)*, pages 236–240. IEEE, 2015.
9. Nelly Bencomo, Amel Bennaceur, Paul Grace, Gordon S. Blair, and Valérie Issarny. The role of models@run.time in supporting on-the-fly interoperability. *Computing*, 95(3):167–190, 2013.
10. Nelly Bencomo, Jon Whittle, Pete Sawyer, Anthony Finkelstein, and Emmanuel Letier. Requirements reflection: Requirements as runtime entities. In *Proceedings of the 32nd International Conference on Software Engineering - Vol. 2*, ICSE '10, pages 199–202. ACM, 2010.
11. Amel Bennaceur, Robert France, Giordano Tamburrelli, Thomas Vogel, Pieter J Mosterman, Walter Cazzola, Fbio M. Costa, Alfonso Pierantonio, Matthias Tichy, Mehmet Aksit, Pr Emmanuelson, Huang Gang, Nikolaos Georgantas, and David Redlich. Mechanisms for Leveraging Models at Runtime in Self-adaptive Software. In Nelly Bencomo, Robert France,

Betty H.C. Cheng, and Uwe Assmann, editors, *Models@run.time*, volume 8378 of *Lecture Notes in Computer Science (LNCS)*, pages 19–46. Springer, 2014.
12. Gordon Blair, Nelly Bencomo, and Robert France. Models@run.time. *Computer*, 42(10):22–27, 2009.
13. Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. Morph: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st International Workshop on Control Theory for Software Engineering*, CTSE 2015, pages 9–16. ACM, 2015.
14. Rodney A. Brooks, editor. *Cambrian Intelligence: The Early History of the New AI*. MIT Press, 1999.
15. Yuriy Brun, Giovanna Di Marzo Serugendo, Cristina Gacek, Holger Giese, Holger Kienle, Marin Litoiu, Hausi Müller, Mauro Pezzè, and Mary Shaw. Engineering Self-Adaptive Systems through Feedback Loops. In Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science (LNCS)*, pages 48–70. Springer, 2009.
16. Sven Burmester, Holger Giese, Eckehard Münch, Oliver Oberschelp, Florian Klein, and Peter Scheideler. Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. *Journal on Software Tools for Technology Transfer (STTT)*, 10(3):207–222, 2008.
17. Sven Burmester, Holger Giese, and Oliver Oberschelp. Hybrid UML Components for the Design of Complex Self-optimizing Mechatronic Systems. In J. Braz, H. Araújo, A. Vieira, and B. Encarnacao, editors, *Informatics in Control, Automation and Robotics I*. Springer, 2006.
18. Sven Burmester, Holger Giese, and Wilhelm Schfer. Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In *Proceedings of the European Conference on Model Driven Architecture - Foundations and Applications (ECMDA-FA)*, volume 3748 of *Lecture Notes in Computer Science (LNCS)*, pages 25–40. Springer, 2005.
19. Radu Calinescu, Lars Grunske, Marta Z. Kwiatkowska, Raffaela Mirandola, and Giordano Tamburrelli. Dynamic qos management and optimization in service-based systems. *IEEE Trans. Software Eng.*, 37(3):387–409, 2011.
20. Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, Jeff Magee, Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle. Software Engineering for Self-Adaptive Systems: A Research Roadmap. In Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science (LNCS)*, pages 1–26. Springer, 2009.
21. Betty H.C. Cheng, Holger Giese, Paola Inverardi, Jeff Magee, and Rogério de Lemos, editors. *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2009.
22. Shang-Wen Cheng, VaheV. Poladian, David Garlan, and Bradley Schmerl. Improving architecture-based self-adaptation through resource prediction. In Betty H.C. Cheng, Rogerio de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee, editors, *Software Engineering for Self-Adaptive Systems*, volume 5525 of *Lecture Notes in Computer Science (LNCS)*, pages 71–88. Springer, 2009.
23. David M. Chess, Alla Segal, and Ian Whalley. Unity: Experiences with a prototype autonomic computing system. In *Proceedings of the First International Conference on Autonomic Computing*, ICAC '04, pages 140–147. IEEE, 2004.
24. M.T. Cox. Metacognition in computation: A selected research review. *Art. Int.*, 169(2):104–141, 2005.
25. Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors. *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2013.
26. Rogério de Lemos, Holger Giese, Hausi Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano

Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl Goeschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, Joao P. Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke. Software Engineering for Self-Adaptive Systems: A second Research Roadmap. In Rogério de Lemos, Holger Giese, Hausi Müller, and Mary Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475 of *Lecture Notes in Computer Science (LNCS)*, pages 1–32. Springer, 2013.

27. Bassem Debbabi, Ada Diaconescu, and Philippe Lalanda. Controlling self-organising software applications with archetypes. In *IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 69–78. IEEE, 2012.
28. Marco Dorigo, Vito Trianni, Erol Şahin, Roderich Groß, Thomas H. Labella, Gianluca Baldassarre, Stefano Nolfi, Jean-Louis Deneubourg, Francesco Mondada, Dario Floreano, and Luca M. Gambardella. Evolving self-organizing behaviors for a swarm-bot. *Autonomous Robots*, 17:223–245, 2004.
29. Stephen Fickas and Martin S. Feather. Requirements monitoring in dynamic environments. In *Proceedings of the Second IEEE International Symposium on Requirements Engineering (RE)*, pages 140–147. IEEE, 1995.
30. J. Floch, C. Fr, R. Fricke, K. Geihs, M. Wagner, J. Lorenzo, E. Soladana, S. Mehlhase, N. Paspallis, H. Rahnama, P.A. Ruiz, and U. Scholz. Playing music building context-aware and self-adaptive mobile applications. *Software: Practice and Experience*, 43(3):359–388, 2013.
31. Sylvain Frey, Ada Diaconescu, and Isabelle M. Demeure. Architectural Integration Patterns for Autonomic Management Systems. In *Proceedings of the 9th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (EASe 2012)*, 2012.
32. David Garlan, Shang-Wen Cheng, An-Cheng Huang, Bradley R. Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.
33. K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav. A comprehensive solution for application-level adaptation. *Software: Practice and Experience*, 39(4):385–422, 2009.
34. K. Geihs, C. Evers, R. Reichle, M. Wagner, and M. U. Khan. Development support for qos-aware service-adaptation in ubiquitous computing applications. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, SAC '11, pages 197–202. ACM, 2011.
35. Holger Giese, Nelly Bencomo, Liliana Pasquale, AndresJ. Ramirez, Paola Inverardi, Sebastian Wtzoldt, and Siobhan Clarke. Living with Uncertainty in the Age of Runtime Models. In Nelly Bencomo, Robert France, Betty H.C. Cheng, and Uwe Assmann, editors, *Models@run.time*, volume 8378 of *Lecture Notes in Computer Science (LNCS)*, pages 47–100. Springer, 2014.
36. Holger Giese, Sven Burmester, Wilhelm Schäfer, and Oliver Oberschelp. Modular Design and Verification of Component-Based Mechatronic Systems with Online-Reconfiguration. In *Proceedings of 12th ACM SIGSOFT Foundations of Software Engineering 2004 (FSE 2004)*, pages 179–188. ACM, 2004.
37. Holger Giese, Stefan Henkler, and Martin Hirsch. A multi-paradigm approach supporting the modular execution of reconfigurable hybrid systems. *SIMULATION*, 87(9):775–808, 2011.
38. Holger Giese and Wilhelm Schfer. Model-Driven Development of Safe Self-Optimizing Mechatronic Systems with MechatronicUML. In Javier Camara, Rogrio de Lemos, Carlo Ghezzi, and Antónia Lopes, editors, *Assurances for Self-Adaptive Systems*, volume 7740 of *Lecture Notes in Computer Science (LNCS)*, pages 152–186. Springer, 2013.
39. Sebastian Götz, Claas Wilke, Sebastian Cech, and Uwe Aßmann. *Sustainable ICTs and Management Systems for Green Computing*, chapter Architecture and Mechanisms for Energy Auto Tuning, pages 45–73. IGI Global, June 2012.

40. Sebastian Götz, Claas Wilke, Sebastian Richly, and Uwe Aßmann. Approximating quality contracts for energy auto-tuning software. In *Proceedings of First International Workshop on Green and Sustainable Software (GREENS 2012)*, 2012.
41. Sebastian Götz, Claas Wilke, Sebastian Richly, Georg Püschel, and Uwe Assmann. Model-driven self-optimization using integer linear programming and pseudo-boolean optimization. In *Proceedings of The Fifth International Conference on Adaptive and Self-Adaptive Systems and Applications (ADAPTIVE)*, pages 55–64. XPS Press.
42. S. Hallsteinsen, K. Geihs, N. Paspallis, F. Eliassen, G. Horn, J. Lorenzo, A. Mamelli, and G. A. Papadopoulos. A development framework and methodology for self-adapting applications in ubiquitous computing environments. *J. Syst. Softw.*, 85(12):2840–2859, December 2012.
43. B. Hayes-Roth. A blackboard architecture for control. In *Artificial Intelligence*, volume 26-3, pages 251–321, 1985.
44. Regina Hebig, Holger Giese, and Basil Becker. Making control loops explicit when architecting self-adaptive systems. In *Proceedings of the Second International Workshop on Self-organizing Architectures*, SOAR '10, pages 21–28. ACM, 2010.
45. Thorsten Hestermeyer, Oliver Oberschelp, and Holger Giese. Structured Information Processing For Self-optimizing Mechatronic Systems. In Helder Araujo, Alves Vieira, Jose Braz, Bruno Encarnacao, and Marina Carvalho, editors, *1st International Conference on Informatics in Control, Automation and Robotics (ICINCO 2004)*, pages 230–237. INSTICC Press, 2004.
46. Julia Hielscher, Raman Kazhamiakin, Andreas Metzger, and Marco Pistore. A framework for proactive self-adaptation of service-based applications based on online testing. In Petri Mahonen, Klaus Pohl, and Thierry Priol, editors, *Towards a Service-Based Internet*, volume 5377 of *Lecture Notes in Computer Science (LNCS)*, pages 122–133. Springer, 2008.
47. Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, , and Anant Agarwal. Seec: A general and extensible framework for self-aware computing. Technical Report MIT-CSAIL-TR-2011-046, MIT CSAIL, 2011.
48. IBM. An Architectural Blueprint for Autonomic Computing, 2003. White Paper.
49. John E. Kelly and Steve Hamm. *Smart machines : IBM's Watson and the era of cognitive computing*. Columbia Business School Publishing, 2013.
50. Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
51. Samuel Kounev. Self-Aware Software and Systems Engineering: A Vision and Research Roadmap. In *GI Softwaretechnik-Trends, 31(4), 2011*, Karlsruhe, Germany, 2011.
52. Samuel Kounev, Fabian Brosig, and Nikolaus Huber. The Descartes Modeling Language. Technical report, Department of Computer Science, University of Wuerzburg, October 2014.
53. Samuel Kounev, Nikolaus Huber, Fabian Brosig, and Xiaoyun Zhu. A Model-Based Approach to Designing Self-Aware IT Systems and Infrastructures. *IEEE Computer Magazine*, pages 53–61, July 2016.
54. Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268. IEEE, 2007.
55. Christopher Landauer and Kirstie L. Bellman. Knowledge-Based Integration Infrastructure for Complex Systems. *International Journal of Intelligent Control and Systems*, 1(1):133–153, 1996.
56. Christopher Landauer and Kirstie L. Bellman. New architectures for constructed complex systems. *Applied Mathematics and Computation*, 120(1–3):149–163, 2001.
57. Peter R. Lewis, Arjun Chandra, Funmilade Faniyi, Kyrre Glette, Tao Chen, Rami Bahsoon, Jim Torresen, and Xin Yao. Architectural aspects of self-aware and self-expressive computing systems: From psychology to engineering. *IEEE Computer*, 48(8):62–70, 2015.
58. Pattie Maes. Concepts and experiments in computational reflection. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 147–155. ACM, 1987.
59. Jeff Magee and Jeff Kramer. Dynamic structure in software architectures. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, pages 3–14. ACM, 1996.

60. Yoann Maurel, Philippe Lalanda, and Ada Diaconescu. Towards a service-oriented component model for autonomic management. *2014 IEEE International Conference on Services Computing*, 0:544–551, 2011.
61. Janet Metcalfe and Arthur P. Shimamura, editors. *Metacognition: Knowing about knowing*. MIT Press, Cambridge, MA, USA, 1994.
62. Melanie Mitchell. Self-awareness and control in decentralized systems (Tech Report SS-05-04). In *AAAI Spring Symposium on Metacognition in Computation*, Menlo Park, 2005. AIII Press.
63. Hausi A. Müller, Mauro Pezzè, and Mary Shaw. Visibility of Control in Adaptive Systems. In *Proceedings of the 2nd International Workshop on Ultra-large-scale Software-intensive Systems*, ULSSIS '08, pages 23–26. ACM, 2008.
64. Christian Muller-Schloer, Hartmut Schmeck, and Theo Ungerer, editors. *Organic Computing - A Paradigm Shift for Complex Systems*. Birkhuser, 2011.
65. A. Newell, P. S. Rosenbloom, and J. E. Laird. Symbolic architectures for cognition. In M. Posner, editor, *Foundations of Cognitive Science*, pages 93–132. MIT Press, 1989.
66. H. P. Nii. Blackboard systems, part one: The blackboard model of problem solving and the evolution of blackboard architectures. In *AI Magazine*, volume 7, pages 38–53, 1986.
67. L.D. Paulson. DARPA creating self-aware computing. *Computer*, 36(3):24, 2003.
68. J. Rothenberg. The nature of modeling. In Lawrence E. Widman, Kenneth A. Loparo, and Norman R. Nielsen, editors, *Artificial Intelligence, Simulation & Modeling*, pages 75–92. John Wiley & Sons, Inc., New York, NY, USA, 1989.
69. Romain Rouvoy, Frank Eliassen, Jacqueline Floch, Svein Hallsteinsen, and Erlend Stav. Composing components and services using a planning-based adaptation middleware. In *Software Composition*, pages 52–67. Springer, 2008.
70. Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.
71. Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.
72. Hartmut Schmeck, Christian Müller-Schloer, Emre Çakar, Moez Mnif, and Urban Richter. Adaptivity and self-organization in organic computing systems. *ACM Trans. Auton. Adapt. Syst.*, 5(3):10:1–10:32, 2010.
73. Mary Shaw. Beyond objects: A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20(1):27–38, 1995.
74. Mary Shaw and David Garlan. An Introduction to Software Architecture. *V. Ambriola and G. Tortora (ed.): Advances in Software Engineering and Knowledge Engineering*, 2:1–39, 1993.
75. Thomas Vogel and Holger Giese. Adaptation and Abstract Runtime Models. In *Proceedings of the 5th Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010)*, pages 39–48. ACM, 2010.
76. Thomas Vogel and Holger Giese. Model-Driven Engineering of Self-Adaptive Software with EUREMA. *ACM Trans. Auton. Adapt. Syst.*, 8(4):18:1–18:33, 2014.
77. Thomas Vogel and Holger Giese. On Unifying Development Models and Runtime Models. In *Proceedings of the 9th International Workshop on Models@run.time*, volume 1270 of *CEUR Workshop Proceedings*, pages 5–10. CEUR-WS.org, 2014.
78. Thomas Vogel, Andreas Seibel, and Holger Giese. The Role of Models and Megamodels at Runtime. In Juergen Dingel and Arnor Solberg, editors, *Models in Software Engineering*, volume 6627 of *Lecture Notes in Computer Science (LNCS)*, pages 224–238. Springer, 2011.
79. Sebastian Wtzoldt and Holger Giese. Modeling Collaborations in Adaptive Systems of Systems. In *Proceedings of the European Conference on Software Architecture Workshops*, EC-SAW. ACM, 2015.
80. Eric Yuan, Naeem Esfahani, and Sam Malek. Automated mining of software component interactions for self-adaptation. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, SEAMS 2014, pages 27–36. ACM, 2014.

81. Franco Zambonelli, Nicola Bicocchi, Giacomo Cabri, Letizia Leonardi, and Mariachiara Puviani. On self-adaptation, self-expression, and self-awareness in autonomic service component ensembles. In *Proc. of the Fifth IEEE Conference on Self-Adaptive and Self-Organizing Systems Workshops (SASOW)*, pages 108–113. IEEE, 2011.