Pre-print version for personal use only!

Chapter 5
Architectural Concepts for Self-Aware Computing Systems

Holger Giese, Thomas Vogel, Ada Diaconescu, Sebastian Götz, and Samuel Kounev

Abstract Self-awareness in a computing system is achieved by implementing a model-based learning, reasoning, and acting loop (LRA-M loop). Similar to feedback loops for self-adaptive software, we argue that the LRA-M loop should be addressed during the architectural design of self-aware computing systems. This allows engineers to explicitly decide and reason about the system's self-awareness capabilities. This chapter therefore introduces the relevant architectural concepts to address and make the LRA-M loop visible in the architectural design. Based on these concepts, we discuss how context-awareness, self-awareness, and meta-self-awareness become manifest in an architecture. Finally, we relate the presented architectural concepts to the definition and framework for self-aware computing systems introduced in previous chapters.

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

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

5.1 Introduction

The vision of self-aware computing as introduced in Chapter 1 promises that self-aware systems achieve their goals in a flexible manner despite the dynamic and uncertain nature of their environments and goals. To achieve their goals, such systems continuously learn and reason about themselves, their environment, and their goals, and if needed, take appropriate actions. For instance, based on their self-awareness, such systems are able to self-adapt at runtime, steer their behavior directly as required, or report to their users to explain what happened.

There are a number of initiatives aiming for more flexible software systems such as autonomic computing [22], self-* systems [3], self-adaptive and self-managing systems [9,10,13,14,24,34], organic computing [1,29], or cognitive computing [21] that advocate a paradigm shift for software from design-time decisions and understanding toward resolving issues dynamically at runtime—typically by equipping the system with a feedback loop [7].

While these approaches traditionally looked only into *reactive* classes of solutions that act at runtime in response to changes without anticipating future changes or reasoning about the long-term future (cf. [17,24]), recently an additional paradigm shift from a *reactive* to a *proactive* operation can be observed that aims to integrate the ability to learn, reason, and act at runtime (cf. [8, 11, 19]). This trend is well in line with the ideas centered around the notion of self-aware computing [1, 2, 20, 23, 25, 27, 43], runtime models [4–6, 39, 41, 42] and related terms [12, 15, 26, 33] that gained momentum in recent years.[1]

In this chapter, we will look at the solution space for self-aware computing systems with the particular focus on *software architecture* as "a collection of computational components—or simply components—together with a description of the interactions between these components—the connectors." [36, p. 4]. Therefore, this chapter explores which concepts are required to describe architectures of self-aware systems. As introduced by the definition of a self-aware computing system, the concepts we address include runtime *models* of the context and the system itself as well as learning, reasoning, and acting *processes* (cf. Chapter 1). In this context, we consider the pre-reflective and reflective forms of self-awareness (cf. Chapter 3).

It is important to emphasize that the chapter does not propose a dedicated architectural language for development or a set of well-established, concrete architectures but rather aims to provide an initial basis to compare approaches as well as to explore and discuss the possible solution space. Consequently, the concepts we discuss and capture in the examples in this chapter need not to be relevant for every but only for specific applications of self-aware computing systems and often only depict a fragment of an architecture rather than a complete architecture. Furthermore, the concepts are the building blocks for such systems but not necessary ingredients. Therefore, they also support modeling architectures of systems that are yet not self-aware. Our goal is to provide an architectural language that allows us

[1] A broader discussion of other related work including also agents and multi-agent systems can be found in Chapter 2 of this book.

Pre-print version for personal use only!

(1) to discuss the whole spectrum of self-aware computing systems (see Chapter 3), (2) to classify whether a given system is self-aware, (3) to study systems that may evolve into such self-aware computing systems and (4) to derive steps to adjust the architecture of a non-self-aware system to migrate it into a self-aware system.

Therefore, we do not claim that these concepts are generally relevant for self-aware computing systems. In contrast, they should be considered as a source of inspiration when conducting research or developing an architecture and design for such a system. Future results, experiments, and solutions may then confirm, refine or even contradict the usefulness of the various concepts we propose. In any of these cases, the purpose of the proposed concepts to start-off research and work on architectures for self-aware computing systems would have largely been fulfilled.

To start-off such research and work, we propose concepts that emerged from the discussions at the Dagstuhl seminar on self-aware computing system and that make the specifics of such systems explicit and visible in the architectural design. We argue that these specifics should become first-class entities of the architectural design such that they can be properly addressed during development. Similarly, Shaw [35], Müller et al. [28, 30], and Brun et al. [7] argue that feedback loops as the essential characteristic of self-adaptive software should be made explicit and visible in the architectural design, for instance, to also make design decisions explicit and to enable reasoning on the design. Consequently, we borrowed several ideas from approaches of the authors of this chapter. Especially, we borrowed ideas from EUREMA [40], addressing the explicit modeling of feedback loops in self-adaptive software, as well as from MechatronicUML [18], supporting collaborations in flexible architectures. However, none of these approaches targets self-aware computing systems in particular.[2] In the context of this chapter, applying all of the proposed concepts may lead to a too detailed model that might be considered more like a specific design rather than a general architecture. However, our intention is to be able to express also subtle differences between solutions in *one* notation rather than finding an appropriate compromise between expressiveness and ease of use. Consequently, an important aspect that will need further attention is to determine under which circumstances the proposed concepts are really helpful for architecture modeling and when they are too detailed and rather concerned with the more fine-grained design.

The concepts we propose in this chapter are the foundations for the following chapters. Particularly, Chapter 6 will explore the specific needs of architectures for a single self-aware computing system while Chapter 7 will explore collectives of self-aware systems. Furthermore, Chapter 8 will review the state of the art and contrast it with the proposals of this chapter and of Chapters 6 and 7. In addition, Chapters 12 and 13 will target the detailed algorithmic questions of how learning, reasoning, and acting are realized within a single or collective of systems, which is not covered by the chapters on the architectures for self-aware systems.

This chapter is organized as follows: In Section 6.2, we introduce the running example we use throughout this chapter as well as basic notational concepts of the UML to describe software architectures. Then, we discuss the proposed concepts

[2] A comparison of these related approaches to self-aware computing systems can be found in Chapter 8 of this book.

Pre-print version for personal use only!

Pre-print version for personal use only!

in terms of elements and relations that are specific to self-aware computing systems in Sections 5.3 and 5.4. Using these concepts, we describe different cases of self-awareness in Section 5.5. In Section 5.6, we discuss these concepts concerning architectural views as well as their coverage of the definition and framework of self-aware computing systems (see Chapters 1 and 3). Finally, we conclude the chapter in Section 25.5.

5.2 Preliminaries

In this section, we introduce the running example, a smart home system, that we use to illustrate the architectural and design concepts for self-aware computing systems. Then, we introduce basic notational concepts from the Unified Modeling Language (UML) [32] to describe software architectures, which are extended in this chapter to address the specifics of self-aware computing systems. Finally, we summarize the definition and framework for such systems from Chapters 1 and 3.

5.2.1 Running Example: Smart Home

To discuss the architectural concepts for self-aware computing systems, we use a running example based on the smart home exemplary scenario presented in Chapter 4. In contrast to the original scenario, we use here a smart home system with a more complicated architecture of pre-reflective components that control devices in a house and that are coordinated by a *house manager* (see Figure 5.1).

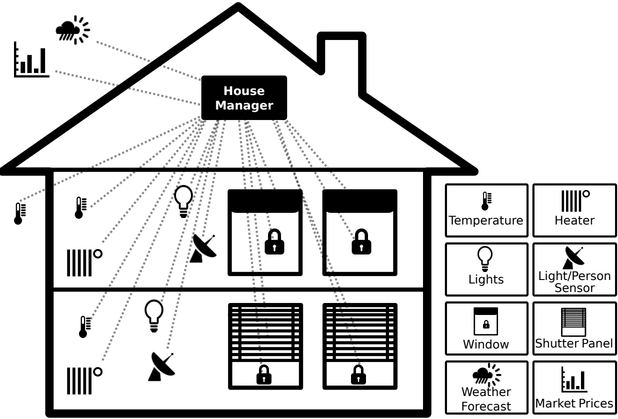


Fig. 5.1: Overview of the Smart Home System with a Centralized House Manager.

Pre-print version for personal use only!

The house manager reports to the user if something goes wrong (e.g., if failures are detected), self-adapts (e.g., to optimize energy consumption) and actuates the device controllers in the house (e.g., in the case of emergencies). Besides a centralized house manager that coordinates the device controllers located in the house, we further consider variants of less hierarchical interaction schemes such as collaborations or self-organization to achieve the coordination among the device controllers.

A house consists of several floors and rooms. Each room is equipped with devices such as sensors to perceive the in- and outdoor temperature, lighting conditions, and persons, as well as controllers for the *heater* (start or stop heating), *lights* (switch on or off the lights), *windows* (tilt, open, or close the window), and *shutter panels* (open or close the panels). Each controller works independently, for example, one controls the heater based on the temperature, another one the windows based on time. This might result in conflicts such as heating up the room while opening the windows. The task of the house manager is to coordinate the controllers according to some goals. Therefore, the manager aims for (1) self-healing and (2) self-optimization while we leave out other self-* capabilities to keep the example simple.

(1) Self-healing:

  a. If a sensor in a certain room is broken, the house manager relies on the sensor data from the neighboring rooms.
  b. If a single point of failure is affected, such as the window cannot be closed any more, a person in the house who is close to the window is notified.

(2) Self-optimization:

  a. The energy consumption should be optimized while achieving the goals such as maintaining a certain room temperature.
  b. Various influencing factors for the optimization can be considered, such as market prices, weather forecasts, government subsidies, user preferences, etc.

In this chapter, we illustrate the architectural concepts for self-aware computing systems with this example. To discuss a particular concept, we often present a fragment of the example emphasizing this concept instead of a complete architecture.

5.2.2 Architectural Modeling with UML

Before we introduce the architectural concepts that are specific to self-aware computing systems, we provide a summary of UML-based concepts [32] for architecture modeling that serve as a basis for this chapter. These UML concepts are modules

Pre-print version for personal use only!

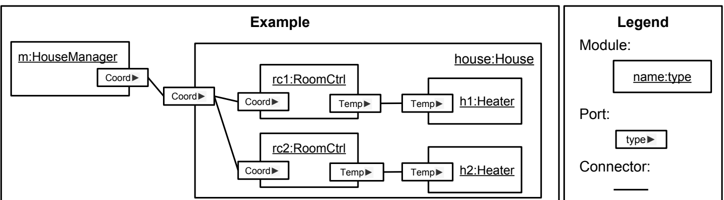


Fig. 5.2: Generic UML Elements for Architecture Modeling.

(e.g., components[3]), ports provided or required by modules, connectors, collaborations, and participation links between modules and collaborations.

Modules have a name and a type and they can be hierarchically composed to *UML hierarchies*. If the name of a module is not relevant, we may omit it (cf. anonymous module). We may also omit the type of a module if it is not relevant for the discussion. Modules may provide or require *ports* that encapsulate functionality and restrict access. Ports are defined by their types. The direction of the arrows (▲, ▼, ►, ◄) within a port denotes whether the port is provided or required by a module.[4] A module provides (requires) a port if the direction of the port's arrow points from the module inwards (outwards). Provided and required ports of the same type are wired by a *connector* to visualize interactions among the corresponding modules.

The example depicted in Figure 5.2 shows two modules of different types: m of type HouseManager and house of type House. The manager m coordinates the controllers rc1 and rc2 that are located in different rooms of the house. Therefore, the house is hierarchically decomposed into room controllers each with its own heater to control. The manager requires the port Coord that is provided by the house. Both ports are connected and the manager coordinates the controllers in the house. The house forwards the coordination commands from the manager to the individual controllers that eventually set the temperature to the heaters in the corresponding rooms. The manager's responsibility is to achieve similar temperatures in both rooms.

In addition to modules with their connectors and hierarchical (de)composition, more flexible forms of cooperating behavior can be modeled with *collaborations*. Collaborations are depicted by ellipses and they are wired to the modules that collaborate by *participation links*. For instance, Figure 5.3 shows a collaboration in which four room controllers agree on a common temperature for each room.

In general, UML hierarchies with shared aggregation and UML collaborations may overlap to some extent. For instance, we may employ collaborations to capture the interaction in a UML hierarchy, but also to capture non-hierarchically structured compositions. However, participations of elements in UML collaborations

[3] Oftentimes the modules may be in fact *components* [38]. However, as components imply a certain degree of encapsulation that might not be the case for the system elements considered in this chapter, we use the more general term of a module here.

[4] This can be seen as simple generalization of the different flow properties for ports in SysML [31].

Pre-print version for personal use only!

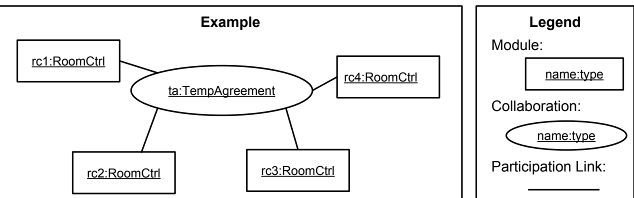


Fig. 5.3: Generic UML Elements for Modeling Collaboration of Modules.

may change rather frequently while memberships in a hierarchy are usually considered more stable and durable (even though such memberships may also change).

5.2.3 Self-Awareness Terminology and Framework

The terminology and framework for self-aware computing systems introduced in Chapters 1 and 3 provide a definition and several dimensions spanning the overall spectrum of such systems. In the following, we briefly summarize the definition and dimensions since they provide the foundation for the architectural concepts of self-aware computing systems, which we discuss in this chapter. At first, we recap the definition of self-aware computing systems given in Chapter 1:

> Self-aware computing systems are computing systems that:
>
> 1. *learn models* capturing *knowledge* about themselves and their environment (such as their structure, design, state, possible actions, and run-time behavior) on an ongoing basis and
> 2. *reason* using the models (for example predict, analyze, consider, plan) enabling them to *act* based on their knowledge and reasoning (for example explore, explain, report, suggest, self-adapt, or impact their environment)
>
> in accordance with *higher-level goals*, which may also be subject to change.

Based on this definition, we may sketch a self-aware computing system with the conceptual *Learn-Reason-Act-Model (LRA-M)* loop (see Figure 5.4). This loop shows the relevant aspects of the definition. Particularly, the system collects *empirical observations* of the self and of phenomena outside the self. *Learning* and *reasoning* processes produce and use *models* that capture knowledge derived from the observations. Based on the knowledge, the system may *act* upon itself and on its context. The processes operate according to higher-level *goals* that may dynamically change.

Consequently, the definition and the LRA-M loop introduce the concepts of *empirical data/observations*, *models*, and *goals*, which are used by *learning*, *reason-*

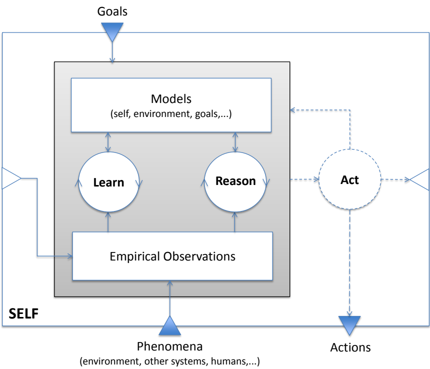


Fig. 5.4: The *LRA-M* Loop Introduced in Chapter 1.

ing, and *acting processes*. We can elaborate this use relationship by describing the *data flow* within the LRA-M loop. The system observes itself and its environmental context, for which models are learned and used for reasoning and acting. Thus, a system realizing such a loop becomes aware of itself and of its context.

These aspects are refined by Chapter 3 providing several dimensions for self-aware computing systems, which we consider as a conceptual framework. This framework covers different levels of self-awareness: A *pre-reflective self-awareness* level denoting simple subjective observations, a *reflective self-awareness* level if learning and reasoning with awareness models are involved, and a *meta-self-awareness* level where the object of the reflection is a reflective self-awareness process. Moreover, the framework distinguishes a subject (i.e., the *span*) and an object of awareness (i.e., the *scope*) while the span reflects on the scope. In this context, the notion of *action scope* that includes all entities that the system may act directly upon as well as the notion of *influence scope* that refers to entities upon which the system may only act indirectly are introduced. Finally, the framework refines the notion of awareness by emphasizing different aspects of awareness such as *identity*, *state*, *interaction*, *time*, *behavior*, *appearance*, *goal*, *expectation*, and *belief awareness*.

In the rest of this chapter, we will discuss architectural concepts for self-aware computing systems, which address these aspects and dimensions by applying and extending the generic UML concepts for architecture modeling. To illustrate these architectural concepts, we use the introduced running example.

5.3 Architectural Elements for Self-Awareness

In this section, we propose general elements (i.e., building blocks) for describing architectures of self-aware systems, which are motivated by the aspects and dimensions of self-aware computing systems introduced in Chapters 1 and 3. Thereby, the architectural descriptions target concrete architectures of systems and therefore,

Pre-print version for personal use only!

Pre-print version for personal use only!

the emphasis is on describing specific instance configurations. An overview of the proposed elements is given in the appendix.

5.3.1 System, Environmental Context, and Modules

As depicted in Figure 5.5a, we may first distinguish a *system* from its *environmental context*. The environmental context, represented by a cloud, is the fragment of the environment (including possibly other systems) scoped by the system's capacities of sensing and exploration. Furthermore, we may distinguish *modules* that represent a system and individual elements that compose the system. Both are depicted by rectangles but a system is illustrated with a bold border in contrast to an individual module having only a thin border. If we do not want to distinguish whether we refer to a system or to a module, we just use a rectangle.

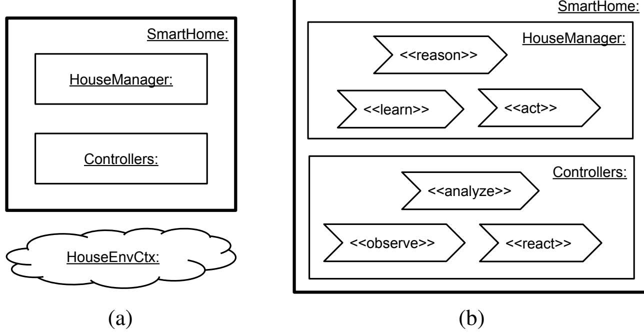


Fig. 5.5: Notation for System, Modules, Environmental Context, and Processes.

The example shown in Figure 5.5a describes the SmartHome system with two modules, Controllers and HouseManager, and the environmental context HouseEnvCtx.

5.3.2 Reflective and Pre-Reflective Processes

We now show how to model processes within systems and modules. In this context, we will distinguish between processes for pre-reflective and reflective self-awareness. The former considers basic subjective observations of the system while the latter additionally considers learning and reasoning activities with awareness models. An example is given in Figure 5.5b. Processes are labeled with ≪learn≫

Pre-print version for personal use only!

when they capture how models are learned based on observations, ≪reason≫ when they analyze the situation or plan actions, or ≪act≫ when they have an external or internal impact and are driven by the results of reasoning.

In the example depicted in Figure 5.5b, the learning process located in the House-Manager learns about the underlying Controllers module. Then, a reasoning process identifies shortcomings of the Controllers and plans their circumvention. Finally, the act process will enact the planned adaptation by effecting the Controllers accordingly.

These processes can be refined and classified into reflective self-awareness and self-expression (see Figure 5.6). For self-awareness, we have already introduced the processes of *learning* models and *reasoning* on such models for acting. For self-expression (i.e., acting), we consider processes that have an external influence such as *adapting* the system itself or other systems, *effecting* the context or other systems, and *reporting* to the user or to superordinated system entities.[5]

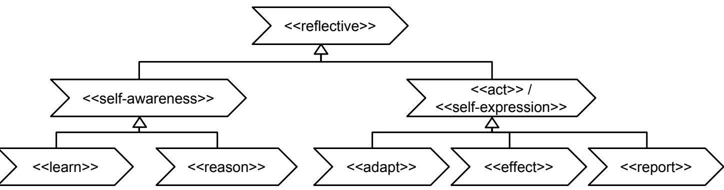


Fig. 5.6: Classification of Reflective Processes.

Similarly, we may classify the processes for pre-reflective self-awareness (see Figure 5.7). *Observe* denotes measuring the system itself, other systems, or the environmental context, *analyze* covers simple variants of analysis based on the observations, and *react* describes the reaction to specific situations either directly observed or identified by the analysis. Such pre-reflective processes can be allocated within systems and modules as illustrated for the Controllers module in Figure 5.5b.

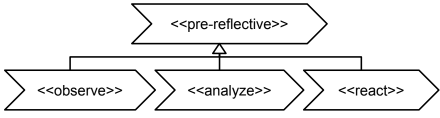


Fig. 5.7: Classification of Pre-Reflective Processes.

The classifications of reflective and pre-reflective processes shown in Figures 5.6 and 5.7 are derived from the definition of self-aware computing systems (cf. Chap-

[5] We distinguish adapting and effecting an entity as the former involves changing an entity (e.g., modifying the entity's structure) while the latter denotes interactions between entities that do not require any substantial changes of the entity (e.g., by exchanging knowledge among entities).

Pre-print version for personal use only!

ter 1) and should therefore be considered at the conceptual level. In practice, these classifications can be further refined or extended given the specific problem at hand.

5.3.3 *Awareness Models, Empirical Data (Models), and Goal Models*

As described by the learn-reason-act-model (LRA-M) loop for self-aware computing systems sketched in Section 6.2.2, *awareness models (AMs)* and *empirical data (ED)* are used online. To capture the scope that is represented by the model or data, we use the stereotypes ≪ctx≫ in the case of the environmental context and ≪sys≫ in the case of the system itself or parts of it.

AMs are employed online[6] and represent originals outside the system (e.g., the environmental context or other systems) or inside the system (e.g., modules or processes of the self). Such models can be subjective and not a perfect representation of the originals as they might be based on individual measurements as part of specific learning processes. In general, AMs are usually obtained by ≪learn≫ processes and they are subject to ≪reason≫ and ≪act≫ processes. AMs are depicted as blue-shaded, rounded boxes in our notation (see Figure 5.8).

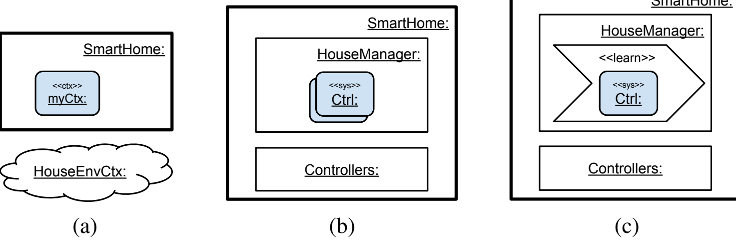


Fig. 5.8: Notation for Awareness Models within Systems, Modules, or Processes.

In the example depicted in Figure 5.8a, the system has an AM of its environmental context. In Figure 5.8b, the HouseManager module has an AM of the Controllers

[6] The awareness models (AMs) discussed here overlap with the concept of *models@run.time* [6] when there is a causal connection with the system itself. Our assumption is that not every AM is or needs to be causally connected to the running system. Another view on runtime models considers any models that are used within the system and that either represent (parts of) the system or context for reflection or specify (parts of) the system for execution [40, 41]. In this view, the reflection models correspond to the idea of AMs. In general, we do not restrict the scope of an AM. If an AM represents the context, it usually supports establishing context-awareness while an AM representing (parts of) the system supports establishing self-awareness.

module, which is thus a system model. Finally, in Figure 5.8c, a learning process within the HouseManager module maintains such a system model locally.

Either a single or a group of models is depicted if one or more models are used online. An AM or a group of them can be located within a system, a module, or a process. A group of AMs covers different aspects of the same scope, for instance, a timing aspect to capture the history of AMs. Otherwise, we use completely separate boxes for the AMs if they refer to different scopes.

In addition to AMs, the LRA-M loop addresses *empirical data (ED)* such as sensor data obtained and used by pre-reflective processes (cf. Chapter 1). Thus, ED is usually obtained by ≪observe≫ processes and subject to ≪analyze≫ and ≪react≫ processes. Such data is specified in the notation as blue-shaded, rounded boxes with a dashed border (see Figure 5.9).

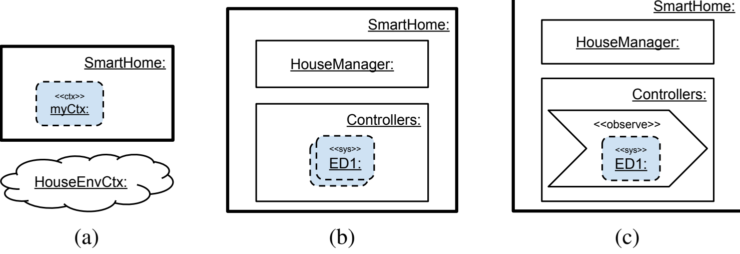


Fig. 5.9: Notation for Empirical Data within Systems, Modules, or Processes.

As outlined in the definition of self-aware computing systems in Chapter 1, a self-aware system *is* driven by goals and it *may be* necessary to be able to cope with changing goals or even dynamically generated goals. Such runtime goals can be explicitly represented in one or more *goal models (GMs)*. Such models are depicted as red-shaded, rounded boxes stereotyped with ≪goal≫ in the notation (see Figure 5.10). In our example of Figure 5.10, the GM describes the criteria indicating the direction that the self-optimization should steer the SmartHome system to, for instance, reducing the energy consumption while considering constraints such as “do not shut down the heater in a room if there is a person in this room”.

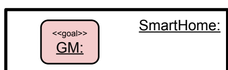


Fig. 5.10: Notation for Goal Models.

Runtime goals must be explicitly captured by online GMs. In contrast, design-time goals influence the system during development, for instance, by determining

Pre-print version for personal use only!

Pre-print version for personal use only!

the type of models and processes to be developed. Some of them may not be explicitly represented and they can remain *implicit* in the implementation—if they do not change dynamically. Goals that may change dynamically must be *explicitly* represented to be able to handle such changes. Thus, if a system needs to be goal-aware, then the goals are typically explicitly represented, otherwise not necessarily.

If goals of another system or element are derived by observations, we use AMs rather than GMs to describe these goals. We only use GMs to denote those goals that are imposed on the system either from the outside (e.g., by the user) or from the system itself generating the goals (e.g., based on some observations).

Splitting the system into multiple layers, GMs can be part of each layer though the goals might be of a different kind. Goals in the lowest layer refer to the domain functionality while goals at higher layers refer to awareness such as to the success/failure of lower-layer goals (cf. awareness requirements [37]). For example, a higher-layer goal may prescribe that the controllers have to achieve the desired room temperature in 90% of the time. The corresponding lower-layer goal prescribes that the desired room temperature should be as close to 22 C as possible.

5.4 Architectural Relations for Self-Awareness

Besides the architectural elements for self-aware computing systems discussed in Section 5.3, the definition for self-aware computing systems given in Chapter 1 and its refinements in Chapter 3 introduce explicitly or implicitly several relations (cf. Section 6.2.2). These relations seem helpful for architectural considerations and they are discussed in the following. An overview of the proposed relations is additionally given in the appendix.

5.4.1 Data Flow Related to Self-Awareness

The first relation we introduce is the *data flow* between models, empirical data, processes, modules, other systems, and the environmental context. This is motivated by the data flow that forms the learn-reason-act-model (LRA-M) loop introduced in Chapter 1. As shown in Figure 5.11, the data flow is represented by a solid black arrow whose direction indicates the direction of the data flow. Thus, the data flow extends the UML connector (see Section 5.2.2) by connecting arbitrary elements and representing a flow of data. Note that such a data flow may be realized by quite different technical means such as procedure calls, messages, or flows.

Using the data flow relation, we can describe that a ≪learn≫ process obtains AMs guided by the goals of the self, a ≪reason≫ process uses the AMs and GMs to reason, and finally, that an ≪adapt≫ process uses the AMs to dynamically change a module (see Figure 5.11a).

Pre-print version for personal use only!

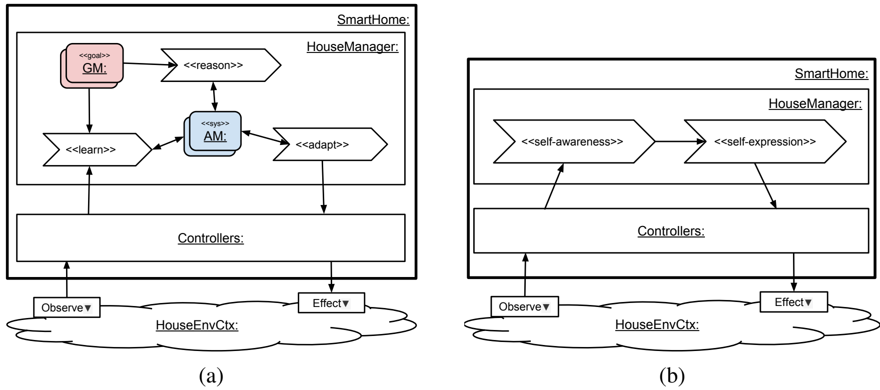


Fig. 5.11: Data Flow for Self-Awareness and Self-Expression.

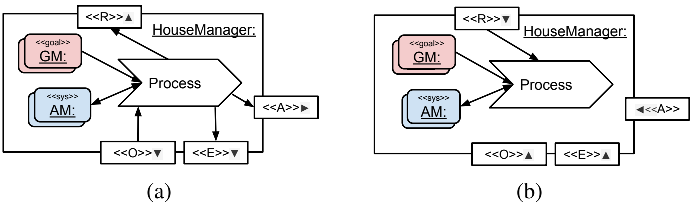
Considering our example and Figure 5.11a, the HouseManager uses the goal model GM, which may prescribe that the desired room temperature should be achieved in 90% of the time, to learn about the performance of the Controllers module. Learning results in producing a set of awareness models AM describing the Controllers' performance, which are then used to reason about the performance and the achievement of the goals. If the goals are not achieved, the HouseManager may adapt the Controllers module based on the AM, for instance, by changing the control strategy to react more quickly to disturbances of the actual room temperature. For this example, Figure 5.11b shows an abstraction that only considers the self-awareness (encapsulating the learning and reasoning) and the self-expression (encapsulating the acting) processes while hiding the employed models. Consequently, the figure only captures the data flow between these two processes and the Controllers module.

In general, we can describe detailed views making the data flow between individual processes, models, and modules explicit (see Figure 5.11a) or abstract views that, for instance, hide the models and the detailed processes (see Figure 5.11b). Thus, a data flow from or to a module or process may be refined to a data flow from or to an element contained in the module or process. Such contained elements can particularly be AMs and GMs that are used in architectural views to emphasize the role of models in self-aware computing. In this sense, the diagram in Figure 5.11a refines the one in Figure 5.11b. It refines the self-awareness and self-expression processes and makes the AMs and GMs explicit. Likewise, we may refine the Controllers module and denote which processes or ED exist within this module.

When refining a module by describing the contained elements such as processes and models, we can emphasize the encapsulation and interactions of these elements by *ports*. A port describes the functionality that is provided or required by modules and connected ports make the interaction among modules explicit. The functionality can be specific to self-aware computing, which we denote by stereotypes. A module can observe (≪O≫) and effect (≪E≫) an element or the context, adapt (≪A≫) an element, or report (≪R≫) to the user or to another element. In this case, the

Pre-print version for personal use only!

module requires corresponding ports for these functionalities (see Figure 5.12a). Moreover, if a module can be observed, effected, adapted, or reported to, it provides the corresponding ports (see Figure 5.12b). Finally, the ports of a module may be connected to elements contained in this module (e.g., to delegate incoming reports to a process).



(a) (b)

Fig. 5.12: Required (a) and Provided (b) Ports of a Module.

In general, we may omit the ports in the diagrams if they are not relevant for the selected architectural view. For instance, a view might emphasize the models and neglect the encapsulation and interaction between modules.

5.4.2 Awareness and Expression Links

Two important aspects of self-aware computing are self-awareness and self-expression. As discussed in Chapter 3, self-awareness has a domain and enables that a subject of the awareness (i.e., the span) reflects about an object of awareness (i.e., the scope) by means of a model employed online. Based on the introduced elements, we can therefore illustrate with an *awareness link* that a scope is represented by a model maintained by a span. Thus, the span is aware of the scope. As shown in Figure 5.13 by the red bold arrow, usually a model or a group of models represents another module or the context. If we want to abstract the models in an architectural view, we link the scope to a process or module containing the (hidden) models.

Additionally, we may have an *expression link* in the opposite direction. Such a link is denoted by a blue bold arrow pointing from a model maintained by a span to the scope (see Figure 5.13). Such a link illustrates that the span's self-expression impacts the scope. Again, if we abstract the model in an architectural view, we link the process or module containing the (hidden) model to the scope.

In our example, the HouseManager has an awareness model of the Controllers module as illustrated by an awareness link in the diagram of Figure 5.13a. Since the HouseManager also adapts the Controllers module, we additionally have an expression link in the opposite direction. For the other three cases in Figure 5.13, an awareness link illustrates that the context is known by the SmartHome system and represented in an awareness model, by a process of the system, and by a module

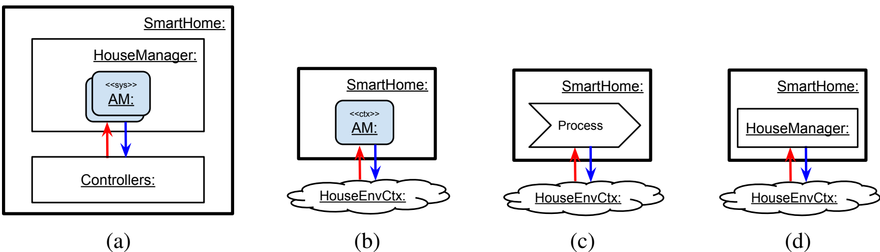


Fig. 5.13: Notation for Awareness and Expression Links.

of the system. These three diagrams also describe that the expression of the corresponding span impacts the context—as visualized by expression links.

According to Chapter 3, the scope of an awareness can be further refined. In our example, a group of awareness models maintained by the HouseManager may be connected via awareness and expression links to dedicated controller modules (see Figure 5.14). Furthermore, the *aspect* of the reflection for each element in the scope may differ (cf. Chapter 3). We therefore attach stereotypes to the awareness links to distinguish, among others, stimulus awareness (≪sa≫), interaction awareness (≪ia≫), time awareness (≪ta≫), and goal awareness (≪ga≫).

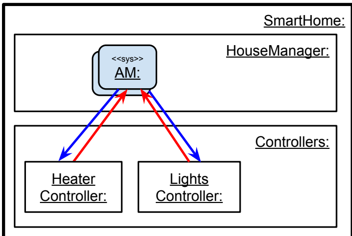


Fig. 5.14: Awareness and Expression Links with a Complex Scope.

As depicted in Figure 5.15, besides *direct* awareness (solid red arrow) and expression (solid blue arrow) we consider *indirect* awareness (dashed red arrow) and expression (dashed blue arrow) to address the *action scope* and *influence scope* introduced in Chapter 3. Typical cases where such scopes become relevant is when modules exploit the awareness and expression capabilities of other modules.

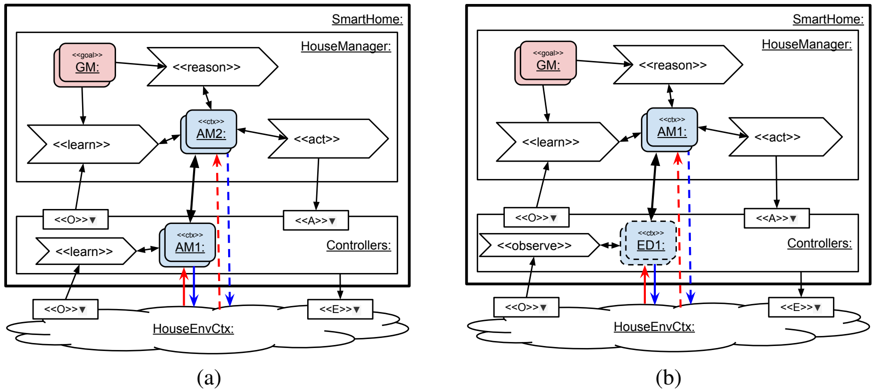
In the example of Figure 5.15a, the Controllers module learns about the context and produces AMs of the context, which is exploited by the HouseManager by feeding its own AMs from these ones through a data flow (black arrow). Consequently, the HouseManger's AMs cover aspects of the environmental context although the HouseManager does not directly observe the context. The same holds for the expression. The HouseManager may indirectly effect the context by adapting the Controllers module. A variant of this example is shown in Figure 5.15b. The Controllers mod-

Pre-print version for personal use only!

ule observes the context and only maintains ED, that is, it does not perform any learning. However, the HouseManager may (re)use this data to perform the learning and produce AMs. Thus, in the example, a module such as the HouseManager does not have to perform the observing or learning processes itself but it can rely on the observations or learned information from other modules such as the Controllers.

Another view of indirect awareness is to derive information about a module by observing and learning its environmental context. This is illustrated in Figure 5.16 showing two modules that do not explicitly interact with each other. However, the Optimizing module effects the environment (see expression link) and the result of its effects may be observed and learned by the Healing module (see awareness link). The resulting awareness model *AM1* is then used to learn about the (behavior of) the Optimizing module. The learned knowledge is captured in the awareness model AM-Opt. Consequently, the Healing module is indirectly aware of the Optimizing module (see indirect awareness link). However, the learned knowledge about the Optimizing module may not be accurate since it is the result of interpreting and speculating about changes in the environment and possible causes of these changes.

As depicted in Figure 5.17, the fact that a system/module is indirectly aware of another system/module can be realized without having to learn a context and system model. Instead, a phenomenon (bold dot) in the environmental context is connected



(a) (b)

Fig. 5.15: Notation for Direct and Indirect Awareness and Expression.

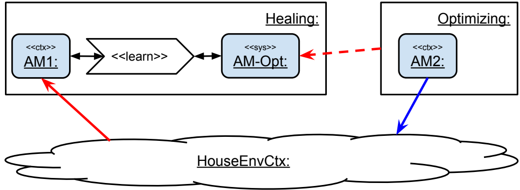


Fig. 5.16: Indirect Awareness via the Environmental Context.

Pre-print version for personal use only!

to the observed system/module with an expression link and to the observing system/module with an awareness link. In the example, the Optimizing module effects the environment (see the expression link), which causes a phenomenon such as a huge increase of the room temperature. This phenomenon is the observable fragment of the Healing module in the context. The Healing module is directly aware of the phenomenon and therefore can be indirectly aware of the Optimizing module.

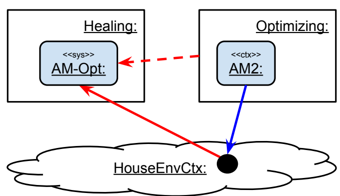


Fig. 5.17: Indirect Awareness via a Specific Environmental Phenomenon.

Finally, the notation can be used similarly to describe direct and indirect awareness if knowledge is obtained through collaborations. This is illustrated in Figure 5.18 showing the Sensor module that senses and learns about the context, which results in the awareness model AM. This module shares the learned knowledge through the collaboration with the other modules. The ReasonC and ReasonE modules reason about the obtained knowledge independent of each other to identify a heating configuration that is comfortable for the user respectively energy-efficient. The LearnUser modules uses the obtained knowledge to learn about the behavior of the user. Finally, the DecideAct module obtains the knowledge created by the ReasonC, ReasonE, and LearnUser modules to make a decision of how to adjust the heating configuration in the house and to eventually enact the adjustments. This example illustrates that modules can be indirectly aware of the context by obtaining knowledge about it through a collaboration, one of whose participants is directly aware of the context.

5.5 Self-Awareness and Architecture

Based on the elements and relations defined in the preceding Sections 5.3 and 5.4, we now approach the question of when and how self-awareness is denoted by awareness links in an architecture. In this context, we discuss that not every occurrence of an awareness link needs to result in self-awareness as defined in this book (cf. Chapter 1). In addition, we study the characterization of specific forms of self-awareness such as meta-self-awareness at the architectural level.

In general, it has to be noted that self-awareness is always relative to a given scope. Usually, the scope is the considered system and environmental context. How-

Pre-print version for personal use only!

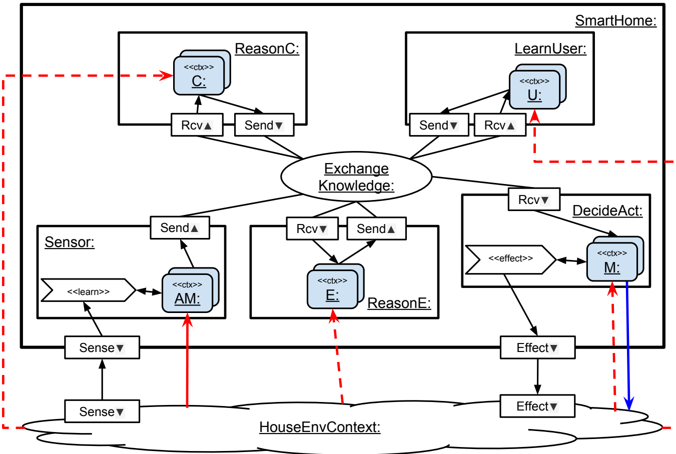


Fig. 5.18: Direct and Indirect Awareness with Collaborations.

ever, we may consider just the context, a particular module of a system, or any other element of a system such as a process.

5.5.1 Self-Awareness: Awareness of the Context

One particular aspect of self-awareness for a system is depicted in Figure 5.19. According to the definition of a self-aware system (cf. Chapter 1), a system must be aware of its environmental context and must have processes capable of learning awareness models and reasoning about the context using the learned models. Furthermore, the system may act upon the models to effect the context. In our example, the SmartHome may have a contextual awareness model of the context capturing information such as the current outdoor temperature and other weather conditions.

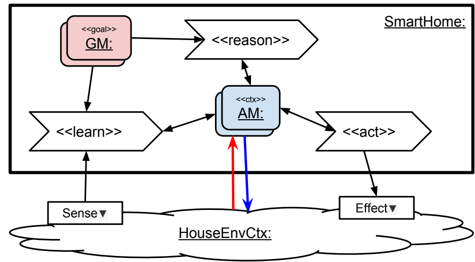


Fig. 5.19: Self-Awareness: Aspect of Context-Awareness.

Pre-print version for personal use only!

5.5.2 *Self-Awareness: Awareness of its own Elements*

A key aspect of self-awareness is that the system is aware of itself or elements of itself, which is illustrated in Figure 5.20. The elements, a system can reflect on and be aware of, are modules or processes. In contrast, we consider reflecting on an awareness model, empirical data, or a goal model as insufficient since we require the existence of the reflective self-awareness processes in terms of learning, reasoning, and acting.

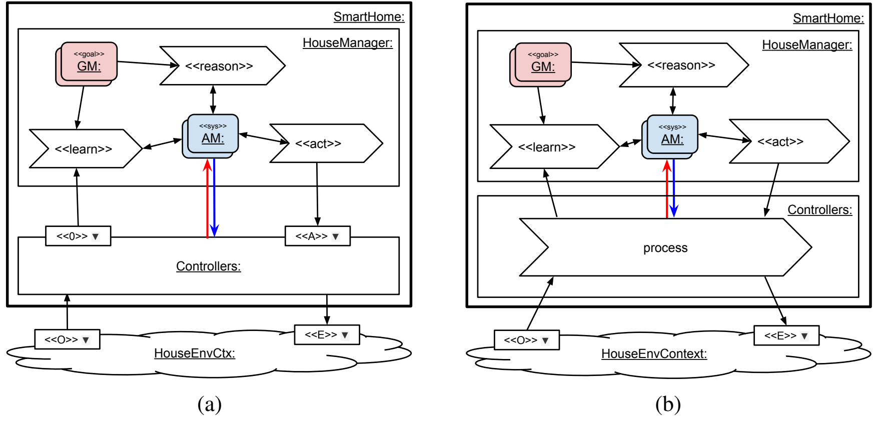


Fig. 5.20: External Self-Awareness Concerning Elements of the System.

When a clear *separation* between the element being the scope and the element being the span of awareness is given, this is similar to the *external approach* in self-adaptive software that separates the managing from the managed element [34]. This approach may simplify the treatment of self-awareness. First, it promotes separation of concerns. Second, the scope need not to be altered for realizing self-awareness (maybe besides adding some sensors and effectors) and the capabilities of learning, reasoning, and potentially acting have to be established only in the span. We name this case *external self-awareness*. In contrast, *internal self-awareness* describes that an element can be aware of itself without any architectural separation between the span and scope. In the context of self-adaptive software, this case is called the *internal approach* [34]. We discuss *internal self-awareness* in the subsequent sections.

Figure 5.20 shows the case of external self-awareness. The HouseManager reflects on the Controllers module using several awareness models that the processes learn, reason, and act upon (Figure 5.20a). In addition, the learning and reasoning processes take into account the goal model. A variant of this case is shown in Figure 5.20b, where the HouseManager reflects on a particular process of the Controllers module. Consequently, the scope of awareness can be individual architectural elements.

Pre-print version for personal use only!

Moreover, multiple awareness links may exist and jointly describe self-awareness. The scopes of these awareness links may overlap and therefore learning, reasoning and acting processes refer to overlapping scopes. This is illustrated in Figure 5.21 showing the Optimizing and Healing modules that are both aware of the Controllers module. While for the awareness links the overlap is generally not a problem, for the expression links the overlap may require coordination of the individual processes, for instance, to avoid conflicting adaptations (cf. [16, 40]).

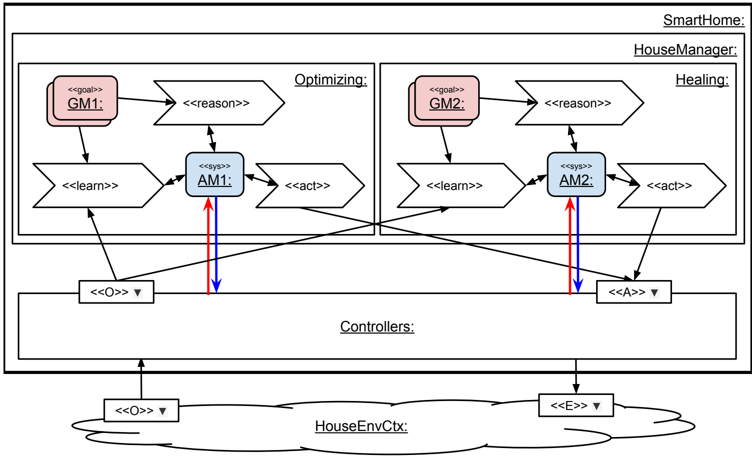


Fig. 5.21: Overlapping Self-Awareness and Self-Expression.

As depicted in Figure 5.21, the HouseManager module consists of two modules, Optimizing and Healing. The former realizes the self-optimization and the latter the self-healing capabilities of the smart home (cf. Section 6.2.1). Each of these two modules runs an LRA-M loop with individual awareness models as well as learning, reasoning, and acting processes. The independent learning and reasoning in both modules are not problematic at the conceptual level since the acting processes have to be coordinated. However, from a practical point of view it may be a waste of resources to let the optimization run an expensive reasoning process to optimize a faulty Controllers configuration until this configuration has been healed.

5.5.3 Self-Loops and Cyclic Self-Awareness

In this section, we discuss the notion of *self-loops* that may occur due to abstraction or internal self-awareness, as well as the related notion of *cyclic self-awareness*. Both notions are neither explicitly covered nor excluded in the Chapters 1 and 3. However, we consider them here because of their architectural implications.

Pre-print version for personal use only!

5.5.3.1 Self-Loops

The notion of a *self-loop* is illustrated by the examples of Figure 5.22. A self-loop denotes that an element is aware of itself, that is, the span and scope of the awareness are not disjoint. For instance, the whole system, a module, or a process can be aware of itself (see examples from left to right in Figure 5.22). The second example additionally emphasizes that the system maintains an awareness model of itself. Similarly to denoting such kind of self-awareness, if an element acts upon itself, we use an expression link as a self-loop to describe the self-expression. Moreover, if an embedded element has a self-loop, we may optionally depict this self-loop at the level of the embedding element to make it visible at the higher level of abstraction.

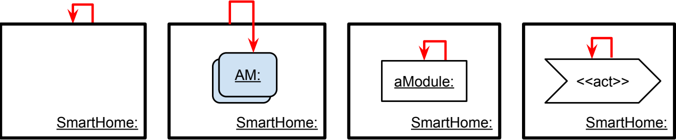


Fig. 5.22: Examples of Self-Loops.

Self-loops may occur because of two reasons: abstraction or internal self-awareness. For the first reason, we abstract from fine- to coarse-grained architectural views while during this abstraction step the awareness and expression links can be lost. However, to make self-awareness and self-expression visible in the architectural views, self-loops are used. For instance, the architecture shown in Figure 5.21 shows that within the SmartHome system, the Optimizing and Healing modules are aware of and act upon the Controllers module. If we abstract from the submodules of SmartHome, we can state that the SmartHome is aware of itself although the awareness is partial since its scope is only the Controllers module. To denote this, we use a self-loop as shown in the leftmost example in Figure 5.22.

The other reason for occurrences of self-loops is more fundamental and based on design decisions or constraints. In this case, we cannot or do not want to separate the span and scope of the self-awareness in the architectural design. In the context of self-adaptive software, this case is called the *internal approach* [34] as one element performs both the managing and the managed part of the self-adaptation. For a self-aware computing system, this results in situations in which one element is (partially) aware of itself. We call this phenomenon *internal self-awareness*, which is denoted by self-loops as depicted in Figure 5.22. Likewise, we may use (blue) expression self-loops to denote the self-expression of an element, that is, an element acts upon itself.

Pre-print version for personal use only!

5.5.3.2 Cyclic Self-Awareness

Besides self-loops, another variant is cyclic self-awareness. An example is given in Figure 5.23 depicting two modules that are aware of each other. This constitutes a cycle since Apartment1Controller is aware of Apartment2Controller and vice versa. In general, longer/bigger cycles involving more than two modules may exist. Moreover, cycles may exist for awareness or expression links such that we may have arbitrary networks of awareness or expression links forming a directed, cyclic graph. Such cycles can complicate achieving stable behavior as modules may continuously be triggered through awareness or expression links. Hence, cycles should be made visible in the architectural design such that they are explicitly handled.

For our example in Figure 5.23, each apartment controller is aware of itself as well as of the other controller. Based on this awareness, it controls the heating in its own apartment and optimizes the energy consumption.

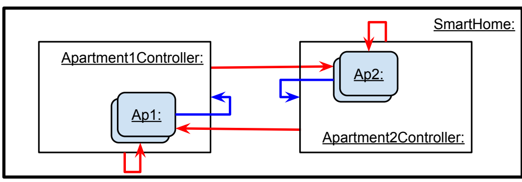


Fig. 5.23: Cycles of Awareness Links.

Similar to self-loops, cyclic self-awareness may occur because of two reasons: abstraction or internal self-awareness. Cyclic self-awareness resulting from abstraction disappears in the architectural design at a more fine-grained level. For instance, Figure 5.24a shows the refinements of the apartment controllers into heater controller and heater modules that resolve the cycle existing in the more abstract design (cf. Figure 5.23). The refinements show that Heater1Controller is aware of Heater2 that has no awareness of Apartment1Controller. The same holds the other way around such that there is no cyclic self-awareness present. Having awareness of its own and the other heater, a controller knows about the temperatures in the different apartments and it may act upon this knowledge, especially, to change the heating settings in its own apartment. This example illustrates that a cyclic self-awareness can be resolved when refining the architectural design.

However, there is also the case of internal self-awareness where no refinement of the design exists that resolves the cycles. This is illustrated in Figure 5.24b showing the persisting cycle of awareness links among the Apartment1Controller and Apartment2Controller, particularly, among their top submodules Heater1Controller and Heater2Controller. These submodules are both aware of each other and each of them can be aware of that the other submodule is aware of it. In this example, a heater controller adjusts the own heater based on the temperature of the own apartment and on the behavior of the heater controller of the neighboring apartment.

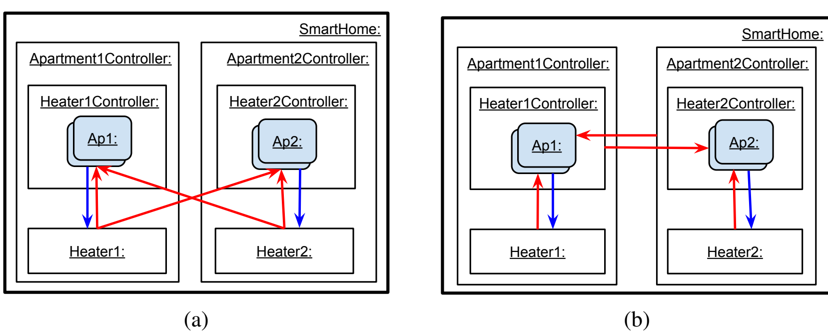


Fig. 5.24: (a) Resolved and (b) Remaining Cyclic Self-Awareness in a Refined Design.

If a cycle of awareness or expression links remains, the depth of the awareness is not clear, that is, the extent to which mutual awareness of the awareness exists. Potentially, there could be an infinite cycle of awareness of awareness, which has to be handled by the reflective learning and reasoning processes. Despite the infinite cycle, the processes have to converge and eventually produce knowledge based on learning and reasoning such that the system may act upon the knowledge.

5.5.4 Meta-Self-Awareness

A particular case of self-awareness is *meta-self-awareness*, that is, a system is aware of its self-awareness (see Chapter 3). Considering external and internal self-awareness, we may combine them to describe meta-self-awareness at the architectural level. Such combinations make the meta-self-awareness explicit in the architectural design. However, to actually realize meta-self-awareness, appropriate reflections and LRA-M loops are required, which are able to identify the self-awareness capabilities of the reflected subsystem. In the following, we focus on making meta-self-awareness visible in the architectural design.

At first, we may combine twice external self-awareness by stacking as depicted in Figure 5.25. The HouseManagerAdjuster reflects on and is aware of the HouseManager that reflects on and is aware of the Controllers. Especially, the HouseManagerAdjuster is aware of the self-awareness established by the HouseManager.

To make the meta levels of reflective self-awareness visible in the architecture, we may stereotype the system with ≪self-aware≫ if it has self-awareness and with ≪meta-self-aware≫ if it has meta-self-awareness capabilities (see the SmartHome Figure 5.25). Similarly, we may stereotype modules if they reflect on other system elements. Modules that do not reflect on any other element are *pre-reflective* and thus stereotyped with ≪pre-reflective≫ (see the Controllers in Figure 5.25). Mod-

Pre-print version for personal use only!

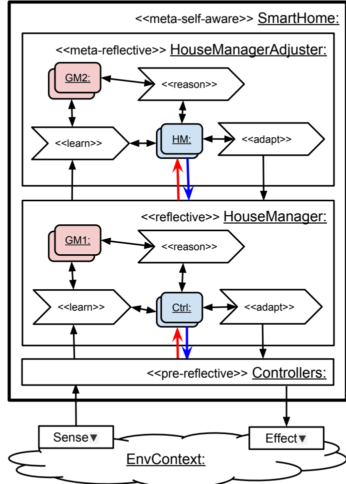


Fig. 5.25: Meta-Self-Awareness by Combining External Self-Awareness.

ules that reflect on a pre-reflective module are *reflective* and thus stereotyped with ≪reflective≫ (see the HouseManager in Figure 5.25). Modules that reflect on a reflective module are *meta-reflective* and thus stereotyped with ≪meta-reflective≫ (see HouseManagerAdjuster in Figure 5.25).

A second possible case is that we stack internal self-awareness twice as depicted in Figure 5.26a. In contrast to the former case, there is no separation between the individual spans and scopes such that we get a compact visual representation of stacking the self-loops for awareness. Likewise, self-loops for expressions (i.e., blue self-loops) are conceivable if the meta-self-awareness includes meta-self-expression.

Finally, we may conceive combinations of internal and external self-awareness to achieve meta-self-awareness. This is illustrated in Figure 5.26b. In general, the self-awareness relationship is not restricted concerning its depth. Thus, we may apply the external or internal self-awareness more than twice to obtain meta-meta self-awareness, meta-meta-meta self-awareness, etc.

5.6 Discussion

In this section, we discuss the proposed architectural concepts for modeling self-aware computing systems. First, we relate them to architectural views and then we discuss the coverage of the needs raised by Chapters 1 and 3.

Pre-print version for personal use only!

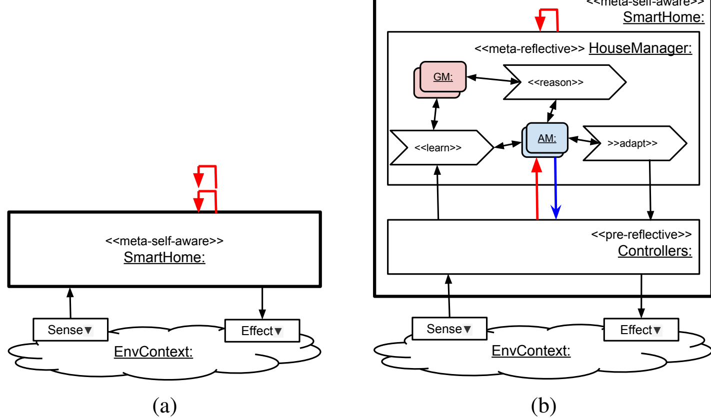


Fig. 5.26: Meta-Self-Awareness by Combining (a) Twice Internal and (b) Internal and External Self-Awareness.

5.6.1 *Architectural Views*

We have introduced several concepts to describe architectures for self-aware computing systems such as modules, processes, goals, and models. Depending on the purpose of architecture modeling, we may consider different architectural views that focus on specific concepts and therefore on specific dimensions of self-aware systems. In this context, we have already identified the following views/dimensions:

System and Module View: The system and modules form the basic structure of an architecture such that this view provides an architectural overview. However, each diagram may consider a different level of abstraction. To avoid too complex diagrams, we may omit the breakdown of a system or module into further submodules by abstracting from the internal design. We rather expect that a diagram of this view represents the whole architecture while it may support different abstraction levels for individual parts. A basic example of such a view is shown in Figure 5.5a.

Self-Awareness Process View: In addition to the system and module view, the pre-reflective and reflective self-awareness processes are an important dimension of the architecture. Therefore, a process view considers processes in addition to or instead of the modules. For a process view and a given level of abstraction, we expect that all processes of the system are covered. A basic example of such a view is shown in Figure 5.5b.

Self-Awareness and Self-Expression View: This view emphasizes the different awareness models and the empirical data used within a system as well as the awareness and expression links. This view, thus, focuses on the awareness and expression relationships potentially neglecting the processes that operate on the models or data. Basic examples of such a view are shown in Figures 5.8 and 5.9.

Pre-print version for personal use only!

Pre-print version for personal use only!

Self-Awareness and Goal View: This view emphasizes the different goals that are used within the system (see Chapter 7 for more architectural concepts concerning goals that may populate the view). This view, thus, focuses on goal models possibly neglecting the processes and the other models and data. However, a goal view can be used together with the process view to show the impact of the goals on the behavior. A basic example of such a view is shown in Figure 5.10.

Such views help in reducing the complexity in the architecture by focusing on the specific dimensions while abstracting from others. Finally, such views can be combined if multiple dimensions are relevant. For instance, the diagram in Figure 5.27 uses processes and models to describe the LRA-M loop, thus, combining the process and the self-awareness/self-expression views. In Chapters 6 and 7 we will study in detail the different concepts and views for various architectural styles.

5.6.2 Coverage

We discuss in the following how the concepts introduced in this chapter cover the needs of architectures for self-aware computing systems. First, we look at the basic needs raised by the definition of self-aware computing systems in Chapter 1, then at the refined needs raised in Chapter 3.

5.6.2.1 Coverage of the Definition of Chapter 1

How self-awareness manifests in the architecture highly depends on the concrete notion of self-aware computing systems that is employed. Therefore, we first look at the definition of self-aware computing systems given in Chapter 1 and consider the related *Learn-Reason-Act-Model (LRA-M)* loop.

The definition of self-aware computing systems emphasizes that these systems employ *models* for capturing knowledge about themselves and their environment, and that these models are *learned* and used for *reasoning* according to their higher-level *goals*, which may be subject to change. The reasoning enables these systems to *act* (e.g., to report to the user or to self-adapt).

The notation for modeling architectures of self-aware computing systems that we introduce in this chapter considers the concepts of awareness models, goal models as well as learning, reasoning, and acting processes. These concepts allow us to model all the aspects mentioned in the definition as part of an architecture. Moreover, the definition considers models for knowledge that refers to the system itself or to the system's environment. Therefore, we distinguish between system and context awareness models, that is, we make explicit whether the scope of reflection is the system itself or the context by applying corresponding stereotypes to awareness models. Similarly, we consider further stereotypes to specialize learning, reasoning, and acting processes. For instance, variations of acting such as explore, explain, re-

Pre-print version for personal use only!

port, suggest, self-adapt, or impact on the environment are captured by corresponding stereotypes.

Consequently, we may conclude that the notation with its concepts covers all of the aspects mentioned in the definition of self-aware computing systems. These aspects can be interpreted as basic needs a system has to satisfy to be self-aware. Hence, our notation with its concepts addresses these needs at the architectural or design level and is therefore a preliminary approach to model architectures and designs of self-aware computing systems.

Besides the definition of self-aware computing systems, Chapter 1 introduces the LRA-M loop as depicted in Figure 5.4. This conceptual loop illustrates the activities and artifacts that are implemented by a self-aware system. Particularly, the system (i.e., the self) collects empirical observations of itself and the environment, uses these observations to learn and reason on models, which eventually enables the system to act upon itself or the environment.

Using the proposed concepts, we can describe the conceptual LRA-M loop. The notation supports modeling the system and environment as well as refining the system to modules, processes, awareness models, goal models, and empirical data while wiring all of them with data flow connectors. This is sufficient to model the LRA-M loop as depicted in Figure 5.27. Compared to Figure 5.4, we extended the LRA-M loop with a pre-reflective ≪observe≫ process to describe how the empirical data is obtained. That is, this process monitors the self and the environmental context to obtain the empirical observations. This data is used by a learning process to obtain awareness models, which is guided by the goals of the self. The reasoning process uses the awareness and goal models to reason. Finally, the act process may influence the context or the self, for instance, by performing a self-adaptation.

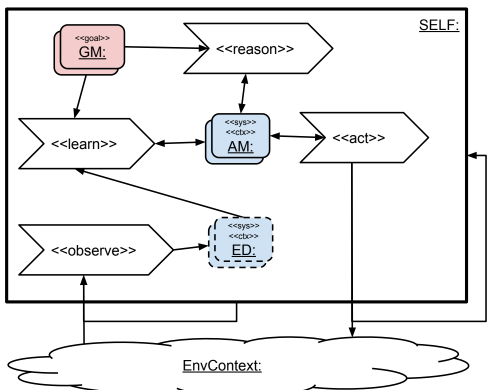
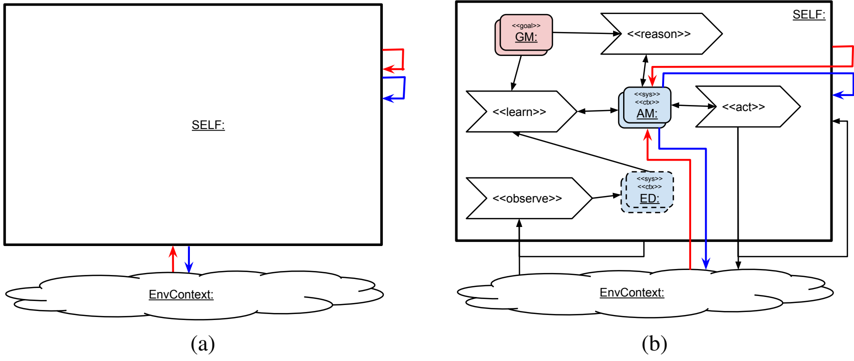


Fig. 5.27: The LRA-M Loop Modeled with the Proposed Concepts.

Using the awareness and expression relations of our notation, the essence of the LRA-M loop can be captured in a more abstract way—as depicted in Figure 5.28a—to illustrate the self-awareness and self-expression of the system. Rather

Pre-print version for personal use only!

than describing data flows that implement some form of awareness, we model the self-awareness and self-expression in a declarative way by using awareness and expression links. Besides being more abstract, this version is also more explicit than the one in Figure 5.27 in making the self-awareness and self-expression visible. Thereby, the awareness and expression target the self as well as the context of the self.



(a) (b)

Fig. 5.28: The LRA-M Loop Extended with Awareness and Expression Links.

In the case of Figure 5.27, the data flow links are required to form a LRA-M loop that realizes the self-awareness. However, the existence of data flow links does not necessarily imply that self-awareness has really been realized since such links may describe quite different data flows (e.g., at the pre-reflective level) that do not necessarily lead to self-awareness. Therefore, Figure 5.28a explicitly denotes the existence of self-awareness by awareness and expression links.

Finally, we can combine both, the data flow and the awareness/expression links, as shown in Figure 5.28b. On the one hand, the data flow among processes and models makes the realization of the self-awareness (i.e., the internal design of the system) visible. On the other hand, the awareness and expression links explicitly emphasizes the self-awareness and self-expression of the system.

5.6.2.2 Coverage of the Framework of Chapter 3

The conceptual framework for self-aware computing systems as introduced in Chapter 3 proposes various dimensions. In the following, we discuss the coverage of these dimensions by the proposed concepts for architecture modeling.

One dimension is the level of self-awareness, which can be *pre-reflective* or *reflective*. We address both levels in Section 5.3.2 by considering self-awareness processes, especially observe, analyze, and react for the pre-reflective level as well as learn, reason, and act for the reflective level. Similarly, we support stereotyping of

systems and modules depending on the level they are operating on (see Section 5.5.4 for the ≪pre-reflective≫, ≪reflective≫, and ≪meta-reflective≫ stereotypes). In this context, we additionally discussed *meta-self-awareness* by providing stereotypes to label a system as ≪self-aware≫ or ≪meta-self-aware≫.

Concerning goals and goal models introduced in Chapter 1 and addressed here in Section 5.3.3, Chapter 3 refines this aspect for self-awareness by discussing the domain of a goal in terms of their span and scope. These refinements are not supported by the architectural concepts proposed here but we will come back to them when discussing collective self-aware systems in Chapter 7.

In general, self-awareness has a domain and enables that a subject of the awareness (i.e., the *span*) reflects about an object of awareness (i.e., the *scope*) (see Chapter 3). This distinction between a span and a scope is addressed by awareness and expression links that connect the span and scope of a self-awareness relationship (see Section 5.4.2). A refinement of such relationships in terms of *action scope* that includes all entities that the system may act directly upon and *influence scope* that refers to entities upon which the system may only act indirectly is introduced in Chapter 3. We cover this refinement with direct and indirect expression links (see Section 5.4.2). Similarly to the expression links, direct and indirect awareness links are further supported.

Finally, we may further refine the awareness links using stereotypes to distinguish, among others, *stimulus awareness* (≪sa≫), *interaction awareness* (≪ia≫), *time awareness* (≪ta≫), and *goal awareness* (≪ga≫). These stereotypes cover different aspects of awareness as discussed in Chapter 3.

Finally, the notion of self-loops discussed in Chapter 3 are addressed by recurrent or cyclic awareness and expression links in Sections 5.5.3.

5.6.2.3 Summary

The definition and framework for self-aware computing systems (see Chapters 1 and 3) propose several dimensions that we have discussed previously and that we summarize Table 5.1. These dimensions provide the necessary architectural and design concepts to explore the solution space of self-aware computing system.

Besides these dimensions that are specific to self-aware computing systems, we consider general architectural concepts such as systems, modules, composition/aggregation and collaborations of systems/modules, ports, connectors and so on to provide the necessary means for describing software architectures. In this context, the dimensions enrich the general concepts, which results in a kind of domain-specific architectural language for the domain of self-aware computing systems.

Pre-print version for personal use only!

Pre-print version for personal use only!

Table 5.1: Dimensions and Section of this Chapter Covering each Dimension.

Dimensions	**Section**
Dimensions Introduced in Chapter 1	
Awareness Models	5.3.3
Empirical Data	5.3.3
Goal Models	5.3.3
Data Flow	5.4.1
Awareness of Context	5.5.1
Awareness of Itself	5.5.2
Dimensions Introduced in Chapter 3	
Pre-Reflective Self-Awareness	5.3.2, 5.5.4
Reflective Self-Awareness	5.3.2, 5.5.4
Meta-Reflective Self-Awareness	5.5.4
Domain (Span and Scope) of Awareness	5.4.2
Direct and Indirect Awareness/Expression	5.4.2
Aspects of Awareness (stimulus, interaction, ...)	5.4.2
Self-loops and Cyclic Self-Awareness	5.5.3

5.7 Conclusion

In this chapter, we developed basic concepts for describing architectures for self-aware computing systems as defined in Chapters 1 and 3.

First, we identified the core architectural elements required for self-aware computing systems. In addition to the system and its refinement to modules, the relevant environmental context the system is aware of has to be identified. Furthermore, we noticed and addressed the need to allocate pre-reflective (observe, analyze, and react) and reflective (learn, reason, and act) self-awareness processes within the architecture. Besides processes representing behavior, the models (e.g., awareness, empirical data, and goal models) have been identified as core ingredients—as they capture the knowledge of self-aware systems—that should be made visible at the architectural level and in the refined design. In addition to all these elements, their linkage is relevant to describe their interactions. The data flow connects modules, processes, and models, which is required for realizing the LRA-M loop and therefore the self-awareness and self-expression. In this context, we aim for making the self-awareness and self-expression explicit by emphasizing when a span is directly or indirectly aware of a scope (cf. awareness link) and when a span directly or indirectly acts upon a scope (cf. expression link).

Based on these concepts we studied how context- and self-awareness can be addressed in the architectural design of self-aware computing systems. Moreover, we discussed specific cases of self-awareness such as multiple overlapping scopes and spans, self-loops, cyclic self-awareness, and meta-self-awareness. Finally, we showed in Table 5.1 that the proposed architectural concepts cover the needs for self-aware computing systems as raised by Chapters 1 and 3. The main concepts covering the needs for describing architectures of self-aware computing systems are relevant for individual (see Chapter 6) as well as collectives of such systems (see Chapter 7).

The concepts extending the UML offer the necessary elements to discuss specifics of self-aware computing systems such as the LRA-M loop at the architectural level. The sketched use of multiple architectural views provide means to emphasize certain specifics such as the processes, models, or the self-awareness/self-expression in the architecture. Making such specifics explicit in the architectural design supports engineers in deciding and reasoning about the system's self-awareness capabilities, which eventually supports development.

In this chapter, we motivated the need for the introduced architectural concepts without an in-depth discussion of their novelty. This discussion will be provided when reviewing the state of the art and research field of architectures for individual and collective self-aware computing systems in Chapter 8.

Acknowledgment

This chapter is the result of stimulating discussions among the authors and other participants, especially Peter Lewis, Nelly Bencomo, Kurt Geihs, Kirstie Bellman, Chris Landauer, and Paola Inverardi, 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. 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.
4. 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.
5. 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.
6. Gordon Blair, Nelly Bencomo, and Robert France. Models@run.time. *Computer*, 42(10):22–27, 2009.
7. 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

Pre-print version for personal use only!

Pre-print version for personal use only!

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.

8. 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.
9. 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.
10. 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.
11. 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.
12. M.T. Cox. Metacognition in computation: A selected research review. *Art. Int.*, 169(2):104–141, 2005.
13. 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.
14. 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.
15. 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.
16. Sylvain Frey, Ada Diaconescu, and Isabelle M. Demeure. Architectural Integration Patterns for Autonomic Management Systems. In *Proc. of the 9th IEEE International Conference and Workshops on the Engineering of Autonomic and Autonomous Systems (EASe 2012)*, 2012.
17. 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.
18. 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.
19. 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

Pre-print version for personal use only!

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.
20. 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.
21. John E. Kelly and Steve Hamm. *Smart machines : IBM's Watson and the era of cognitive computing*. Columbia Business School Publishing, 2013.
22. Jeffrey O. Kephart and David M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
23. Samuel Kounev. Self-Aware Software and Systems Engineering: A Vision and Research Roadmap. In *GI Softwaretechnik-Trends, 31(4), November 2011*, Karlsruhe, Germany, 2011.
24. Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In *FOSE '07: Future of Software Engineering*, pages 259–268. IEEE, 2007.
25. 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.
26. Janet Metcalfe and Arthur P. Shimamura, editors. *Metacognition: Knowing about knowing*. MIT Press, Cambridge, MA, USA, 1994.
27. Melanie Mitchell. Self-awareness and control in decentralized systems (Tech Report SS-05-04). In *AAAI Spring Symp. on Metacognition in Computation*, Menlo Park, 2005. AIII Press.
28. 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.
29. Christian Muller-Schloer, Hartmut Schmeck, and Theo Ungerer, editors. *Organic Computing - A Paradigm Shift for Complex Systems*. Birkhuser, 2011.
30. Hausi A. Mller, Holger M. Kienle, and Ulrike Stege. Autonomic Computing Now You See It, Now You Don't. In Andrea Lucia and Filomena Ferrucci, editors, *Software Engineering: International Summer Schools, ISSSE 2006-2008, Salerno, Italy, Revised Tutorial Lectures*, volume 5413 of *Lecture Notes in Computer Science (LNCS)*, pages 32–54. Springer, 2009.
31. Object Management Group. OMG Systems Modeling Language (OMG SysML^{TM}), 2015. Version 1.4, formal/2015-06-03.
32. Object Management Group. OMG Unified Modeling Language^{TM} (OMG UML), 2015. Version 2.5, formal/2015-03-01.
33. L.D. Paulson. DARPA creating self-aware computing. *Computer*, 36(3):24, 2003.
34. Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Trans. Auton. Adapt. Syst.*, 4(2):1–42, 2009.
35. Mary Shaw. Beyond objects: A software design paradigm based on process control. *ACM SIGSOFT Software Engineering Notes*, 20(1):27–38, 1995.
36. Mary Shaw and David Garlan. An Introduction to Software Architecture. volume 2, pages 1–39. World Scientific Publishing Company, 1993.
37. Vítor E. Silva Souza, Alexei Lapouchnian, William N. Robinson, and John Mylopoulos. Awareness requirements for adaptive systems. In *Proc. of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 60–69. ACM, 2011.
38. Clemens Szyperski, Dirk Gruntz, and Stephan Murer. *Component Software Beyond Object-Oriented Programming*. Component Software. Addison-Wesley, New York, NY, USA, 2nd edition, 2002.
39. 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, May 2010.
40. 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.
41. 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, Workshops and Symposia at MODELS 2010, Reports and Revised Selected Papers*, volume 6627 of *Lecture Notes in Computer Science (LNCS)*, pages 224–238. Springer, 2011.

42. 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'14, pages 27–36. ACM, 2014.
43. 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.

Summary of the Notation

The following table provides a summary of the concepts for modeling architectures of self-aware computing systems. For each concept, its name, syntactic construct (notational element), description, and rationale are listed.

Table 5.2: Architectural Concepts for Self-Aware Computing Systems.

Name	Syntax	Description	Rationale
System	name:type	A *system* with a name and type. We may omit either its name (anonymous system) or type. It can be hierarchically decomposed into *modules*. Stereotypes such as ≪self-aware≫, ≪meta-self-aware≫ and so on indicate whether a system is self-aware, meta-self-aware etc.	The entirety of the modeled system distinguished from the *environmental context*.
Module	name:type	A *module* with a name and type. We may omit either its name (anonymous module) or type. It can be hierarchically decomposed into modules. The stereotypes ≪reflective≫ and ≪pre-reflective≫ indicate whether the module reflects on any other module or not. Stereotypes for higher forms of reflection are ≪meta-reflective≫, ≪meta-meta-reflective≫ etc.	*Modules* are required to decompose a *system* or other *modules*.
Environmental Context	name:type	An *environmental context* of a *system* describes the fragment of the environment that is scoped by the system's capacities of sensing and exploration. It has a name and type, one of which we may omit.	The portion of the overall environment that is relevant for the *system*.

Pre-print version for personal use only!

Pre-print version for personal use only!

Port	type ▼	A *port* describes provided or required functionality of a *system*, *module*, or *context*. It is characterized by its type. The direction of the arrow indicates a required (arrow points outwards the element requiring the functionality) or provided (arrow points inwards the element providing the functionality) port. Stereotypes indicate the kind of functionality: effect ≪E≫, adapt ≪A≫, observe ≪O≫, and report ≪R≫.	A *port* supports encapsulation of elements by making the providing or requiring functionality explicit.
Data Flow	⟶	A *data flow* describes the interactions between *systems*, *modules*, *processes*, and *models* by means of exchanging data.	*Data flow* connectors make the compositional structure explicit, that is, how elements are wired.
Composition	name:type / name:type / name:type	In a *composition* (UML composite structure diagram), we refine modules to other modules while we distinguish between exclusive (solid border of the embedded module) and shared (dashed border of the embedded module) membership of an embedded module.	A *composition* allows us to decompose the system into modules and a module into other modules and to distinguish the kind of membership of a contained module.
Collaboration	name:type	A *collaboration* describes the cooperating behavior among *systems*, *modules*, and *processes*. The concrete behavior of the cooperation is described within the collaboration.	A *collaboration* supports modeling more flexible cooperations among *systems*, *modules*, and *processes* compared to wiring all of the elements using *data flow* connectors.
Participation	——	A *participation* connects a *system*, *module*, or *process* to a *collaboration* such that the *system*, *module*, or *process* participates in the *collaboration*.	A *participation* makes explicit which elements cooperate via a *collaboration* in contrast to directly exchanging data through *data flow* connectors.
Process	Process	A *process* describes activities within a *system* or *module* and therefore emphasizes the behavior within these structural elements.	Self-aware systems have specific *processes* such as learning *awareness models*, reasoning, or acting that should be made visible in the architecture.

Pre-print version for personal use only!

Awareness Model	name:type	An *awareness model* represents learned aspects of a scope. The scope is often (part of) the *system* itself, other *systems*, or the *environmental context*. In the former case, the model is stereotyped with ≪sys≫, in the latter case with ≪ctx≫. An awareness model has a name and type, one of which can be omitted.	An *awareness model* makes explicit that a span maintains a model representing the scope of the awareness.
Empirical Data	name:type	*Empirical data* represents observations of a scope. The scope is often (part of) the *system* itself, other *systems*, or the *environmental context*. In the former case, the empirical data model is stereotyped with ≪sys≫, in the latter case with ≪ctx≫. An empirical data has a name and a type, one of which we may omit.	An *empirical data* model makes explicit that a span collects (sensor) data about a scope.
Goal Model	≪goal≫ name:type	A *goal model* describes the goals (parts of) the *system* should achieve. Goals are imposed to the system from outside (e.g., by the user) or internally produced. A goal model has a name and type, one of which can be omitted. It is stereotyped with ≪goal≫.	Explicit *goal models* are required since self-aware systems are driven by goals and they should be able to handle dynamically changing goals.
Awareness Link	→	An *awareness link* denotes that a span is directly aware of a scope. If a span exploits awareness knowledge about a scope that has been established by another span, the former span is indirectly aware of the scope. Indirect awareness is transitive and can be explicitly represented by dashed awareness links. Awareness links can be specialized by indicating their type: ≪sa≫ for stimulus, ≪ia≫ for interaction, ≪ta≫ for time, and ≪ga≫ for goal awareness.	An *awareness link* connects the span and the scope to make explicit which element learns and reasons about which other element, and specifically, which element is the original represented in an *awareness model*.

Expression Link	→	An *expression link* denotes that a span directly impacts a scope. If a span indirectly impacts a scope via another span, the former span can be connected to the scope to make the indirect expression visible. Indirect expressions are transitive and can be explicitly represented by dashed expression links. To specialize the expression type, a link is stereotyped with effect ≪E≫, adapt ≪A≫, observe ≪O≫, or report ≪R≫.	An *expression link* connects the span and the scope to make explicit which element acts upon another element.

Pre-print version for personal use only!