Software and Systems Modeling
https://doi.org/10.1007/s10270-024-01157-1

REGULAR PAPER

A model-based reference architecture for complex assistive systems and its application

Judith Michael[1] · **Volodymyr A. Shekhovtsov**[2,3]

Received: 16 March 2023 / Revised: 22 January 2024 / Accepted: 5 February 2024

Abstract
Complex assistive systems providing human behavior support independent of the age or abilities of users are broadly used in a variety of domains including automotive, production, aviation, or medicine. Current research lacks a common understanding of which architectural components are needed to create assistive systems that use models at runtime. Existing descriptions of architectural components are focused on particular domains, consider only some parts of an assistive system, or do not consider models at runtime. We have analyzed common functional requirements for such systems to be able to propose a set of reusable components, which have to be considered when creating assistive systems that use models. Such components constitute a reference architecture that we propose within this paper. To validate the proposed architecture, we have expressed the architectures of two assistive systems from different domains, namely assistance for elderly people and assistance for operators in smart manufacturing in terms of compliance with such architecture. The proposed reference architecture will facilitate the creation of future assistive systems.

Keywords Assistive systems · Context-aware · Reference architecture · Model-based software engineering · Daily activities support · Assistive digital twin

1 Introduction

Motivation. Complex assistive systems use sensory information and data to provide situational support for human behavior at the time the person needs it or asks for it. Nowadays, assistive functionalities are broadly used in a variety of domains including automotive [9, 99], production [77, 95, 96], aviation [31], smart homes and ambient assisted living [65] or medicine [48, 86].

The development of such systems spans different domains in informatics and is highly complex, so such an undertaking has to be well-justified in terms of costs and benefits. A challenge is that assistive systems often target niche application areas, with only a few potential application users with very specific needs. Still, delivering such systems for these niche areas can be of high importance, as they provide critical functionality, e.g., ensuring safety in potentially dangerous work environments [38], supporting elderly people with cognitive impairments [53], or ensuring safety for others one creates products for [86].

While developing systems for such areas, the goal is to reduce the resource needs and the development effort as much as possible to achieve at least partial feasibility, as this allows more systems to be developed at a lower cost. This allows the provision of more assistive systems even for niche domains, to address a wider range of critical problems. Current software engineering research and practice proposes a number of activities targeting such a goal [17, 28, 98]. Many such activities possess at least one of the following features:

1. they are based on *using models at runtime* as a result of applying the principles of *model-based* and *model-driven software engineering paradigms* to the development of assistive systems, as working within these paradigms helps to reduce development time and resource require-

Communicated by Oystein Haugen.

✉ Judith Michael
michael@se-rwth.de

Volodymyr A. Shekhovtsov
volodymyr.shekhovtsov@imed.ac.at

[1] Software Engineering, RWTH Aachen University, Aachen, Germany

[2] Medical University of Innsbruck, Innsbruck, Austria

[3] University of Klagenfurt, Klagenfurt, Austria

Published online: 16 March 2024

ments without compromising software quality [1, 4, 17, 98].
2. they rely on the availability of established *reference architectures*, as making such architectures available to the developers reduces the effort for developing systems based on such architectures, allows for interoperability, and facilitates the design of concrete architectures [22, 32, 35, 60, 61].

The problem with the above features is that, while the techniques possessing either of them are widely studied and used, they are not combined together, i.e., *reference architectures do not specifically target assistive systems using models at runtime*. When referring to the usage of models at runtime of assistive systems, we consider using, e.g., process or task models, context models, goal models, or models for events and actions. Using these models enables an assistive system to react to changes and provide means for adaptation. Current approaches realizing adaptivity, e.g., Monitor-Analyze-Plan-Execute over a shared Knowledge (MAPE-K) [5, 87], or context-aware systems [27], do not provide all concepts needed for providing assistance for human users. Thus, defining a reference architecture that integrates common practices for handling context information and models at runtime would improve the state of the art. In this paper, we address this research gap by proposing a reference architecture for model-based assistive systems, which answers the research question "which architectural components are needed and how they should be connected to create an assistive system which uses models at runtime." We consider this architecture as a further step toward reducing the effort for developing assistive systems, making them more accessible to people in need.

Our approach. We follow the process of defining reference architectures from [75] and aligning our reference architecture with the RAModel reference model for reference architectures [76]. We start from an information source investigation step, then we continue with an architectural analysis and synthesis and conclude with an architectural evaluation.

In the first step, we investigate possible sources of functional requirements for assistive systems which use models at runtime; this is followed by collecting and analyzing requirements from these sources. In a second step, based on these functional requirements, we define a set of necessary components and their connections that form a proposed reference architecture. Finally, we analyze the presence of these components in two existing assistive systems from different domains, namely a system to support elderly people in their domestic environment [72] and a system to handle digital twins of production systems [26]. The first system relies on models at runtime in all its components; the second system uses generative aspects to create large parts of the application as well as models at runtime. In detail, our main research contributions are:

- An analysis of what aspects are the foundation for assistive systems using models at runtime and a set of functional requirements for such systems
- A reference architecture with 16 components derived from these requirements
- Two different ways of using the reference architecture, namely (1) for analyzing existing architectures which we show in two case studies, and (2) for creating new architectures, which we show in one example
- Guidelines and best practices for applying the proposed reference architecture and the methods in real-world scenarios

Outline. Section 2 lies down the foundations for our approach. Section 3 analyzes functional requirements for assistive systems, which use models at runtime. Section 4 presents the reference architecture, which includes components related to these requirements. Section 5 analyzes two existing assistive systems in terms of compliance with the proposed reference architecture. Section 6 discusses related work, and Sect. 7 discussed guidelines for using the reference architecture, limitations of our approach, and relevant non-functional requirements. The last section concludes this paper.

2 Foundation

This section describes the foundations: (1) for the effort-reducing features forming the presented approach, namely reference architectures and using models at runtime, (2) for the target systems, namely assistive and context-aware systems.

2.1 Reference architectures

There are several definitions of reference architecture in the existing body of literature. Bass et al. [12] provided a definition that is based on the concept of a reference model specified as "a division of functionality together with data flow between the pieces [...] a standard decomposition of a known problem into parts that cooperatively solve the problem.". A *reference architecture* is then defined as "a reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them". We use the above definition while concretizing software elements as *components* and *interfaces*.

Nakagawa et al. [76] established a reference model for reference architectures (RAModel) defining the following

sets of elements to be implemented by such architectures: (1) *domain*: the elements of the problem domain (concepts, constraints, quality attributes, etc.) reflected in the reference architecture; (2) *application*: the scope and the functional requirements of the applications to be built based on this architecture; (3) *infrastructure*: software elements that could be used to build applications based on the reference architecture; (4) *crosscutting elements* such as architectural decisions or rules that are spread across the other elements.

The existing body of literature also contains several definitions for the process of building reference architectures. We follow the ProSA-RA process by Nakagawa et al. [75] containing the steps of information source investigation, architectural analysis and synthesis, and architectural evaluation.

We align our proposed reference architecture with RAModel and ProSA-RA process as follows: (1) we define the domain concepts related to assistive systems and context-awareness (*domain* set) further in this section (information source investigation step); (2) we specify the requirements to the assistive systems (*application* set) in the next section (architectural analysis step); (3) we specify the implementation components, interfaces, and their connections (*infrastructure* set) in Sect. 4 (architectural synthesis step); (4) we do not consider crosscutting elements; (5) we validate our reference architecture in Sect. 5 (architectural evaluation step).

2.2 Using models at runtime

Model-based software engineering (MBSE) considers using models and model transformations as the fundamental elements of software development. Here we restrict ourselves to consider the class of such systems that use *models at runtime* (models@run.time [17]), which means that the models expressed in some structured format are loaded into the running instances of the applications within the system to drive their execution. Note that introducing such a restriction means that we are not going to address MBSE activities related to other phases of the software development lifecycle such as dealing with models at design or development time. Models at runtime approaches allow for change of these models during system execution. This property allows us to cope with the change of a system and context information captured as data. Moreover, it allows for dynamic adaption as systems can modify their behavior based on information provided by models.

Figure 1 shows a minimal set of components (a fragment of a reference architecture) for a system that handles models at runtime. In this, we follow Mayr et al. [63, 64] which introduced a set of common components (positioned as architectural patterns) for such systems. Relevant components include (a) a *model storage* that permanently stores the models, (b) a *model handler* that parses and interprets the

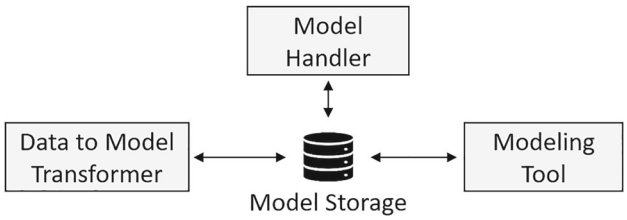

Fig. 1 Basic reference architecture for model-based systems

models obtained from storage prior to using them to control system functions at runtime, (c) a *modeling tool* that allows the modeler to create and manage models using some visual notation and transfer them to the running system, (d) (optional) *a data to model transformer*, which derives models from data.

In practice, model-based systems that handle models at runtime would include even more components, e.g., Aßmann et al. [8] propose a more complex reference architecture for the systems that use models at runtime including an analyzer, reasoner, monitor, and concrete modeling languages, e.g., to express goals as data flow in the reasoner. They define a *managed system* to be monitored and controlled, whereas this could be a human, a system, or a cyber-physical system (CPS) for assistive systems.

Assistive systems widely use models at runtime to provide assistance, such systems can be positioned as model-based assistive systems. Bencomo, Götz, and Song [14] have identified types of models for assistive systems using models at runtime. This includes, e.g., structure models for the system constituents and their states, behavior models defining what the system can or will do based on its current state, or goal models used to identify a current state and if certain goals are fulfilled or not.

Models at runtime can be integrated into assistive systems in various ways. Szvetits and Zdun [93] provide a comprehensive overview of which models can be used during runtime and for which purposes. For assistive systems, we can distinguish between the main perspective one has for defining these models: Are they describing humans in the real world (which then reflect in functionality of a system), or are the models describing mainly the system and its properties. To give some examples:

1. *Process models* could be used to describe the behavior of humans to be supported [67]. These descriptions could be used during the runtime of the assistive system to describe the support steps a system should provide. To use behavior models as runtime models is especially interesting, as human behavior might change over time and we require an assistive system to adapt to this behavior.
2. *Goal models* could be used to describe wished system states one might want to reach. This is similar to agent-

based systems which use goals to define what an agent should pursue [97]. Within assistive systems, such goals can be shown to human users as options they have in using the system [70]. When selecting a certain goal, an assistive system could use reasoning functionalities to detect how to reach that goal and provide human users with support.

3. *Event languages* allow, e.g., to describe events that occur in a system together with actions on how to react to an event [15]. Such approaches could also be applied to assistive systems during runtime where events refer to observable human behavior and the actions describe how an assistive system should react to them.
4. *Context models* include contextual information, e.g., about the environment of a person in assistive systems, spatial relations, or personal and social details [71]. By having this information explicitly modeled, an assistive system can monitor context changes and simplify adaptations.
5. Using *Graphical User Interface (GUI) models* during runtime allows for flexibility and adaption [16]. This property is interesting for assistive systems that require changing the workflow of a system dependent on user interactions.
6. Some *Architecture Description Languages (ADLs)* allow integrating behavior descriptions of specific system components, e.g., automata or state charts [83]. Those behavior models can be used to check if the running assistive system really refers to the planned system design, e.g., by monitoring and translating observed events in the running systems into an architecture model reflecting the running system [36] and compare this with the planned one. Moreover, developers could analyze problems in the assistive system by creating log trace models based on the information received or sent from architectural components [49].

Clearly, what types of models are integrated during runtime have an influence on what specific components and subcomponents are needed. Thus, approaches using models at runtime might differ in their concrete implementation but can share common constructs (see Fig. 1).

2.3 Context-awareness

Context-aware systems [27] use context to provide relevant information and/or services to the user. This relevancy depends on the user's task. Such context-aware systems have to provide means to handle different kinds of context in relation to human tasks. Following [81], there exist four phases in a context life-cycle for context-aware systems:

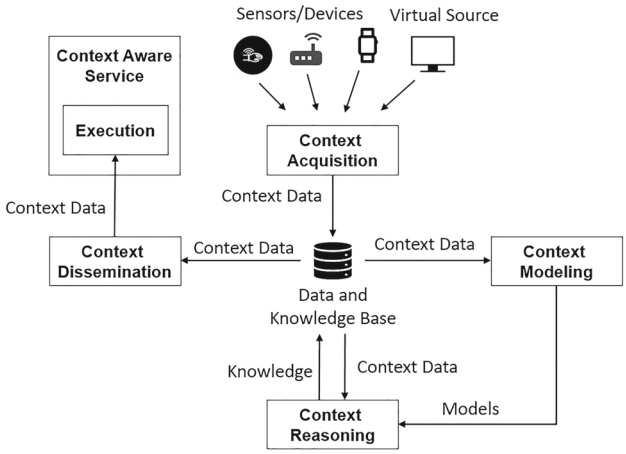

Fig. 2 Reference architecture for context-aware systems

1. *Context Acquisition:* Sensing and capturing heterogeneous context information provided by physical sensors/devices and virtual sources;
2. *Context Modeling:* Extracting and maintaining the context of interest as models and classifying context entities and relationships between these entities;
3. *Context Reasoning:* Deducing new knowledge based on available context;
4. *Context Dissemination:* Distributing context information to the consuming context-aware services and triggering actions based on the context.

Figure 2 represents the main components, which are needed to allow these phases. We consider a data and knowledge base as the main component for the storage of related data. Another option would be not to use a database but pass on the data between acquisition and modeling directly, pass on models and context data from the context modeling to the context reasoning component, and pass on the enriched context data to the context dissemination component. This context dissemination could deliver context information by allowing the context-aware service to query for relevant information or using a publish-subscribe mechanism.

Model-based assistive systems need to be context-aware, as changes in the context might lead to different support solutions in reasoning processes [2]. MBSE for context-aware systems needs approaches for context modeling [71] to include all relevant information in addition to human behavior. Such context information provides additional information used for support, e.g., in which room a person is, handicaps, or functionalities a device provides.

2.4 Assistive systems

An assistive system is a software system that "(1) provides situational support for human behavior, (2) based on

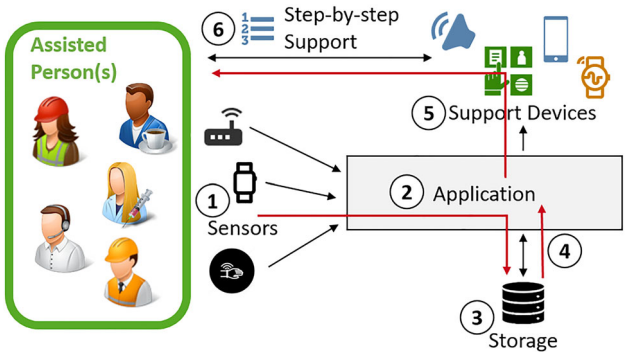

Fig. 3 Reference architecture for assistive systems

information from previously stored and real-time monitored structural context and behavior data, (3) at the time the person needs or asks for it" [45]. Situational support requires knowing the relevant current state of the person and its' context. Providing good support requires some adaptability of the provided support to fit specific user needs. These user needs are either known by the system and previously stored [71] or the user has to interact with the system and provide this information. This interaction can also be used to tell the assistive system that support is needed. Providing support at the time the person needs it is more complex as it requires knowledge about when to support a person based on her or his behavior or context changes.

Assistive systems should not be confused with the term "assistive technologies" [43], which describes devices for people with disabilities, which can be physical devices as well as software or cyber-physical devices. They are provided to specific user groups with mental, physical, or age-related disabilities. In contrast, assistive systems provide support functionalities to any user who is interested in being assisted.

Figure 3 shows the generic reference architecture for such domain-independent assistive systems. In this picture, red arrows reflect the sequence of steps in a process of assisting the users by means of the system, black arrows indicate the direction of the data exchange between the components, and the steps themselves are indicated by the numbers in circles.

In the first step, the behavior is monitored (1) using sensors [58] and the context of assisted persons [71]. This behavior is then handled (2) in the main application of an assistive system. In particular, the main application stores (3) both behavior and data. The storage could be either a single database or a set of e.g., relational databases, time-series databases for sensor data, or triple stores for ontologies. In the next step, (4) the application searches for known behavior and data in the storage and transforms it into knowledge about possible assistive actions. This can involve reasoning mechanisms [2]. When a person asks for assistance, the application, based on the possessed knowledge, uses (5) multimodal support and smart digital assistance devices [24] to provide (6) step-by-step support.

We use the reference architectures from Figs. 1, 2 and 3 as fundamentals for defining requirements and a reference architecture for model-based context-aware assistive systems.

3 Analyzing requirements for assistive systems

To be able to create an assistive system that uses models at runtime, we need to identify the main functional requirements for assistive, context-aware, and model-based systems. For a discussion of relevant non-functional requirements, we refer the reader to Sect. 7.4.

3.1 Functional requirements

To derive the functional requirements, we start with the definition of assistive systems from [45], presented in the previous section. Based on the three parts of this definition (situational support part, structure/behavior context part, and on-time part), we could identify the first set of requirements (see Table 1). An assistive system should enable system-to-person communication (R1), e.g., by providing step-by-step information, managing stored context and behavior data (R2), monitoring the person and her context (R3), e.g., via sensors and sensor middleware, enabling the interactive person-to-system communication (R4) and detecting an assistive need (R5), e.g., by identifying disorienting behavior or context changes.

Based on the above definition of assistive systems, it becomes clear that such a system needs to be aware of its context and could be, thus, called a context-aware system. Following [27], there exist three important context-awareness activities: (1) presentation of information and services to a user, (2) automatic execution of a service and (3) tagging of context to information for later retrieval. These three aspects result in further requirements (see Table 2). Regarding these activities, an assistive system should provide ways to present information (R6), e.g., in a GUI, give acoustic support, by vibrations, present services (R7), e.g., to choose between different support modes, allows the automatic execution of services or on devices (R8), e.g., to open the blinds or switch off a device, and to handle the previous context used for later support (R9). As we consider context as a part of our data, the concrete technique of we are tagging context to information (activity 3) is not needed. However, an assistive system should be able to handle the previous context.

As stated in Sect. 2, there exist four phases of a context life-cycle for context-aware systems [81], which are reflected in several requirements (see Table 3): For (1) *Context Acqui-*

Table 1 Requirements based on the definition of an assistive system (see [45])

Req. num.	Requirement definition	Assistive system definition
R1	Enable system-to-person communication	Situation. support (1)
R2	Manage stored context & Behavior data	Structure/behavior context (2)
R3	Monitor the person and her context	
R4	Enable person-to-system interaction	On-time (3)
R5	Detect the assistive need	

Table 2 Requirements based on activities of context-aware systems

Req. num.	Requirement definition	Activity
R6	Present information	Presentation (1)
R7	Present services	
R8	Allow automatic execution	Automation (2)
R9	Handle previous context	Tagging (3)

sition, we have identified R3 to monitor the person and her context via sensors, sensor middleware, devices, and virtual sources, e.g., monitor if somebody clicks on a button. For (2) *Context Modeling*, we have identified the need to create new models (R10), to merge new context data with existing information (R11), and to handle runtime models (R12), e.g., by workflow-, process-, or own engines or interpreters. For (3) *Context Reasoning*, we have identified the need to reason about models and make predictions (R13) which includes context preprocessing, sensor data fusion, and context inference. For (4) *Context Dissemination*, we have identified the need for interfaces between components storing data and services using context information (R14), the automatic execution of some components or services (R8), and the ability to control actuators (R15).

Aßmann et al. [8] propose a reference architecture for systems using models at runtime. This is reflected in the following requirements (see Table 4): The base layer includes models of the managed system which requires the ability to create, read, update, and delete models (R16), to have facilities to store and load models (R17), and to transform data from the monitored system into models (R18). The configuration management layer includes a reasoner (R13) to create a reconfiguration plan, needs to analyze runtime models and compare them with existing goals (R19), and a learner who updates models (R16) based on context data, and learns models by analyzing reasoning plans (R20). The goal management layer requires the handling of goals (R21).

3.2 Mapping of functional requirements

In the next step, we have taken the functional requirements and grouped them into logical entities. Together with our own knowledge of the engineering of assistive systems, we have defined a set of general components that are able to handle the needed functionality. Table 5 presents an overview of the different requirements in column one. Column two indicates where the requirement originated from (A for the assistive system in Sect. 2.4, C for context-awareness in Sect. 2.3, and M for models at runtime in Sect. 2.2). The next columns show from 1 to 16 in which component the requirement should be handled.

As we rely on multiple sources while deriving the requirements, some of the requirements are overlapping. In particular, R1 and R4 overlap with R6 and R7, R10 overlap with R16, and R12 overlap with R13 and R19. However, we do not consider this as a problem as they are treated individually when mapping them to components.

To improve this categorization, we have revised our initial mapping after (1) completing the detailed description of each component and (2) a basic implementation of the components for an example project. The changes are highlighted in red (deleted mapping) and green (added mapping) in Table 5.

4 Components of assistive systems

We have identified 16 important components for assistive systems that can be used (1) to evaluate an existing architecture for possible missing aspects and (2) to implement a new assistive system. Figure 4 shows the detailed version of an abstract and domain-independent assistive system architecture, which includes all 16 components. The reason why some of them are optional is discussed in Sect. 4.1. In Sect. 4.2, we discuss a method of how these components can be used in practice to support the engineering of assistive systems.

The components and their communication are modeled as a component-connector model using a variant of the architecture description language MontiArc [42]. The boxes represent *components*, and each component can include several subcomponents of the same type in the technical realization of the architecture. The possible behavior of each component can be found in the detailed explanation of the functionalities of each component. Each arrow represents *connectors* between components leading from an input to an output port. On each connector, possible port types are

Table 3 Requirements reflecting the context lifecycle phases

Req. num.	Requirement definition	Lifecycle phase
R3	Monitor the person and her context	Context acquisition (1)
R10	Create new models	Context modeling (2)
R11	Merge new and existing context data	
R12	Handle runtime models	
R13	Reason about models	Context reasoning (3)
R14	Provide interfaces between context storage and usage	Context dissemination (4)
R8	Allow automatic execution	
R15	Control actuators	

Table 4 Requirements reflecting needs for using models at runtime

Req. num.	Requirement definition	Layer
R16	Create, read, update, delete models	Base layer
R17	Store and load models	
R18	Transform data into models	
R13	Reason about models	Configuration management layer
R19	Analyze runtime models	
R20	Learn models	
R21	Handle goals	Goal management layer

Table 5 Mapping from functional requirements to components showing the origin of the requirement (A for assistive system, C for context-awareness, M for models at runtime)

Requirement	Origin	Component															
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
R1 enable system-to-person communication	A								X								
R2 manage stored context & behavior data	A			X													
R3 monitor the person and her context	A, C	X															X
R4 enable person-to-system interaction	A											X					
R5 detect the assistive need	A							X									
R6 present information	C									X							
R7 present services	C										X					X	
R8 allow automatic execution	C												X			X	
R9 handle previous context	C		A				D										
R10 create new models	C					X									X		
R11 merge new and existing context data	C		X			D											
R12 handle runtime models	C						X										
R13 reason about models	C, M		D					X									
R14 provide interfaces between context storage and usage	C				X												
R15 control actuators	C													X			
R16 create, read, update, delete models	M														X		
R17 store and load models	M			X	A												
R18 transform data into models	M					X											
R19 analyze runtime models	M						X										
R20 learn models	M							X									D
R21 handle goals	M							X								X	

described; however, on this reference architecture level, they are underspecified and need to be refined in the concrete technical realization, e.g., when the concrete language for models at runtime was chosen, or concrete sensors, actuators or support devices were specified. For better readability, we have chosen a combined representation for connectors in both directions between two components. Moreover, we additionally represent the connection between the users/context and the system with arrows, as we consider this as important. As this connection happens outside the software system border, the data flow between the human/context and the assistive system is modeled using different types of arrows.

We use the following structure to discuss each component in detail:

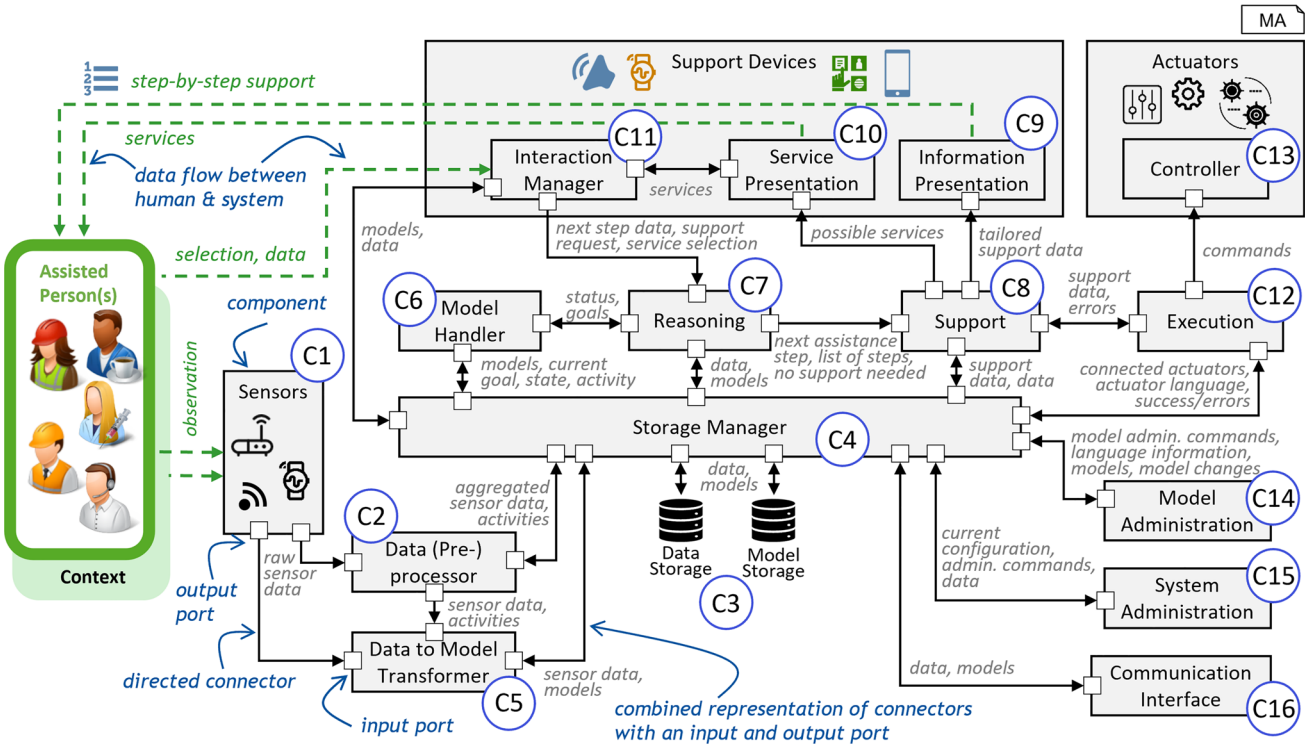

Fig. 4 Assistive system reference architecture represented with the MontiArc component-connector architecture description language

1. **Requirements** for assistive systems: which requirements are related to this component and why.
2. **Input:** which data should be handled as input to this component coming from which other components (incoming arrows for a component in Fig. 4).
3. **Functionalities:** what are the methods this component should provide (describing the component behavior).
4. **Output:** which data results from this component and is handed over to which other components (outgoing arrows for a component in Fig. 4).
5. **(optional) Example:** comparison of the functionality with existing systems or architectural concepts.

C1 - Monitoring component(s)

Requirements: Monitoring components fulfill a requirement to provide functionality for monitoring a person (R3) to ensure that all relevant data is included in the assistive system.

Input: These components receive status changes or more complex behavior from a human, machine, or software.

Functionalities: They are used to monitor the person and her context. Typically, there exist several monitoring components with different functionalities on different devices. The purpose of these devices is to sense the human-in-the-loop, relevant objects, and the environment.

Output: The output is raw sensor data, which is stored after pre-processing (C2) or transformed into models (C5).

Example: Typical examples for monitoring are sensors, sensor middleware, devices, and virtual sources such as screen tracking on a PC, and behavior tracking on smartwatches or smartphones. Other examples are, e.g., systems to detect human steps [54]. Another option is video-based monitoring systems, which are technically possible but not really wanted by observed persons, especially in their private homes [7].

C2 - Data (pre-)processing component(s) (opt.)

Requirements: The data processing and preprocessing component is needed to handle data from the monitoring devices and prepare it in the needed quality. It fulfills the requirements to merge new and existing context data (R11) and to handle the previous context (R9).

Input: It gets input from sensors, which is a (raw) sensor data point or sensor data stream or it gets its input from an external "black box" activity recognition system that has already preprocessed the data to activities and (filtered) sensor data. Note that C2 itself can be implemented as a ("white box") activity recognition system, see the Example section.

Functionalities: Data (pre-)processing component(s) aim either to pre-process raw data from the monitoring component(s) (C1) or to prepare data for the model handler to get more complex representations and aggregated data. It stores aggregated data via the storage manager (C4).

Output: The output is aggregated sensor data and a set of activities getting stored in the storage (C3) via the storage

manager (C4). If a data-to-model transformer (C5) exists, it could also send it this set of activities and its sensor data.

Example: An example of the preprocessing system (C2) is an activity recognition system, which transforms raw sensor data (events) into more complex activities [23]. In this case, it is supposed to be implemented as a part of the solution (i.e., as a "white box" system). Another example could be a system that takes camera pictures as input from a camera (C1), it detects falls of patients [103], and provides the information if a fall happened or not to the storage manager (C4).

C3 - Storage

Requirements: This component fulfills requirements to manage stored context and behavior data (instances) (R2) and to store and load models (R17).

Input: Both input and output of this component are served by the storage manager (C4) which hides the specifics of storage implementation from the rest of the system. As input for the storage component, the storage manager provides the structural context and behavior data, in particular, the data from sensors, together with models or documents for visualization of assistive steps, all this data can be stored by the storage component. The storage manager also provides commands to retrieve the data.

Functionalities: The storage needs to provide CRUD functionality via interfaces. For the different types of data that are relevant as input, it might be reasonable to have different types of database systems, when necessary even in parallel.

Output: As mentioned, the storage component is accessed via the storage manager (C4), which provides commands to retrieve the data and models from the storage. As a response, the storage component provides the requested data as its output to the storage manager.

Example: We can use, e.g., relational databases to store structural data, document stores for, e.g., models or JSON files, or triple-stores to handle the data as subject-predicate-object data entities.

C4 - Storage Manager

Requirements: The storage manager fulfills requirements to provide interfaces between context storage and usage (R14) and to provide functionality to store and load models (R17).

Input: As the storage manager is the central component for creating, accessing, updating, and deleting data, it can get context information, process information, and model data from different components using data and models (C2, C5-C8, C12, C14-C16) to send to the storage component (C3). It also gets requests to retrieve data from the above components and gets the requested information back from the storage component (C3).

Functionalities: If various databases are used in an assistive system, an abstraction between the data and the application is needed. This storage manager is an abstraction layer between requests from components using data and models (C2, C5-C8, C14-C16) and components storing data and models (C3). For components using data and models exists only one component to contact. Their requests are then transformed into specific requests for the connected database(s).

Output: The storage manager provides interfaces to access data and model information covering structural context and process information. By means of these interfaces, it provides the context information, process information, and model data to store in the storage component (C3). It also sends requests to retrieve data to the storage component and provides the requested information to the different components using data and models (C2, C5-C8, C12, C14-C16).

Example: If we are working with technologies such as Hibernate, the storage manager includes, e.g., various data access object classes and the infrastructure to access the database.

C5 - Data-to-Model Transformer (opt.)

Requirements: The data-to-model transformer aims to synthesize models from existing data. It fulfills the requirements to create new models (R10) and to transform data into models (R18).

Input: This component can receive different kinds of input: (1) (Raw) sensor data points or a sensor data stream from one or more sensors (C1), (2) activities and sensor data from activity recognition systems (C2), (3) event logs from third-party systems via the communication interface (C16) to the storage manager (C4), and (4) one or more existing models via the storage manager (C4).

Functionalities: In a learning assistive system, structure, and behavior models could be automatically extracted from data. This component can also use data to add variants in existing models, e.g., process models, or update existing models.

Output: Models and the data used to create these models for reproducibility to be stored in the storage (C3) via the storage manager (C4).

Example: Process discovery algorithms from process mining could take raw data described in event logs and transform it into process models, e.g., [18] shows an approach to use such algorithms in a (semi-) automated transformation chain to derive digital twin cockpits from sensor data.

C6 - Model Handler

Requirements: The model handler satisfies the requirements to handle runtime models (R12) and to analyze runtime models (R19), e.g., by comparing a currently detected activity with existing processes.

Input: In order to use models at runtime, components such as the model handler are needed to process these models. These could be models from different modeling languages and directions, e.g., workflows, events, states, or goals. Fur-

ther input from the storage manager (C4) is certain aspects of these models, e.g., the current goal, state, or activity, and from the reasoning component (C7) updates on models and data based on reasoning results.

Functionalities: The main functionality of the model handler is to orchestrate and manage instances from our models at runtime, e.g., run goal or process instances, add a current action to a model instance, or create a new model instance if a new model at runtime has started, and compare a current goal, state, or activity of the object/subject to be observed with existing model information. The result of this comparison can be passed on to the reasoning component (C7) or the component can already provide new goals to reach for reasoning if anomalies were detected in the comparison.

Output: The model handler provides either status information or new goals to reach the reasoning component (C7).

Example: A part of the model handler component could be, e.g., process engines, model interpreters, analyzers, or evaluators.

C7 - Reasoning component(s)

Requirements: The reasoning components form the core intelligence of assistive systems and, thus, fulfill requirements to detect the assistive need (R5), to reason about models (R13), to learn models (R20) and to handle goals (R21).

Input: Such a component gets either (a) a goal, or (b) a problem from the model handler (C6), (a) high-level information about the next steps, (b) triggering support, (c) selections of services from the interaction manager (C11), or data and models via the storage manager (C4).

Functionalities: The reasoning component gets (a) a goal and finds a solution to reach that goal, or gets (b) a problem and finds the best solution for this problem. In both versions, it is possible to deduce new knowledge, e.g., add new cases to a case base. Depending on the level of support, finding the solution can be a simpler approach or a more intelligent one where old solutions were taken into account such as case-based reasoning. Moreover, this component should cover the detection of the assistive need. This requires reason about context information, e.g., about the current stress level or preferences of users when they want to be supported, and provides the decision to the support component.

Output: The outputs are (1) a possible next step for assistance, (2) several steps, or (3) the information that no support is currently needed. This information is handed over to the support component (C8). The interaction manager (C11) might influence the reasoning process by allowing the assisted person to select support solutions if several are provided. Moreover, the reasoning component can request updates on the current goals from the model handler (C6) and provide information about possible next steps. Reasoning data and models can be stored via the storage manager (C4).

Example: The reasoning components may provide the next step for support after they have checked if all resources needed to do that step are present in the context information.

C8 - Support components

Requirements: The support components fulfill a requirement to enable system-to-person communication (R1) by providing relevant support information.

Input: As input, they get the results from the reasoning component (C7), and additional support information via the storage manager (C4) from the database.

Functionalities: Support components collect all relevant information needed to provide human behavior support in a way that users will be able to understand. They provide the needed support information to the information representation component (C9) of the support devices, which requires tailoring the information for different devices, e.g., specific screen sizes or acoustic vs. textual representations, and providing default presentations. Moreover, these components can prepare information for diverse users and their specific needs. Additionally, they provide services to the service presentation component (C10). This requires them to know which services are available for end users and how they can react to them.

Output: They hand on different kinds of information packages for individual support steps tailored for specific devices (C9) or possible services (C10) the user could choose from. For automated support, support information is sent to the automatic execution components (C12). All this data can be stored via the storage manager (C4) in the database to keep the support history.

Example: In a client–server architecture, the support components would be implemented as server-side components which, e.g., add additional pictures or audio files to support steps or create full support sentences from given phrases and used resources. Moreover, they can provide services to support adaptability to meet diverse end-user needs [40].

C9 - Information Presentation

Requirements: These components are implemented by different kinds of devices and fulfill a requirement to present information (R6).

Input: They get support information tailored for specific devices from the support component (C8).

Functionalities: They display tailored support information for the users to be assisted. This presentation could be, e.g., a graphical user interface, an acoustic way of information presentation, or a multi-modal representation of support information.

Output: Information for step-by-step-support is presented to the supported user via different user interfaces.

Example: In a production process, support information could be displayed on a tablet as text and pictures, in a smart glass as short text, or in more silent areas of a factory as acoustic information.

C10 - Service Presentation (opt.)

Requirements: These components fulfill a requirement to present services (R7).

Input: Information about what services to present to the user is provided by the support component (C8). This list of services might be updated by the interaction manager (C11).

Functionalities: If the user should be able to trigger certain functionalities via the assistive system, it needs to show her a set of services, e.g., callable methods or execution functionalities, and allow for selection. Another provided functionality could be to enable the users to add additional information that was not detected by the monitoring system or provide corrections if something was wrongly detected.

Output: A list or more detailed representation of possible services is provided for the supported person and the interaction manager (C11).

Example: Such services could, e.g., control the blinds in a smart home, change some machine settings in a production environment, or control certain settings of a car entertainment system.

C11 - Interaction Manager

Requirements: The interaction manager fulfills the requirement to enable person-to-system interaction (R4).

Input: This component allows communication from the assisted person to the system. Thus, the input can be (a) information about the behavior such as selections of several next possible steps provided, or (b) information from services a user provides the system in a direct way, e.g., model changes, or preferences. Moreover, it is informed about the list of services by the service presentation component (C10).

Functionalities: The interaction manager takes the user input and redirects it to the relevant places, e.g., the reasoning (C7) or the storage manager component (C4). Adjusted models need to be communicated back to the storage manager (C4). Choosing the next steps or triggering support is important information for the reasoning component (C7). The service presentation component (C10) needs to know about services that are not selectable after a certain decision was made by the user, already selected options might not be provided again or need to be changed e.g., if the service "close the blind" was selected, the available services have to change to "open the blind". However, this information is not directly provided to C10 or C8 but indirectly via the reasoning component (C7) which decides regularly again which services to show or which options a user could select from in the next steps. The returned selection to the interaction manager (C11) could also affect possible next support steps as the assistive system might provide the service to learn new preferences or selections from the user which affects the decisions in the reasoning component (C7).

Output: (a) high-level information about the next steps to (C7), (b) triggering support in (C7) via support requests, (c) selections in services to the reasoning component (C7) and the service presentation (C10), and (d) model changes and data provided by users to the storage manager (C4).

Example: The supported person can choose between different options for the next steps, adjust models used for reasoning, or she can trigger support.

C12 - Automatic execution components (opt.)

Requirements: These components fulfill a requirement to allow automatic execution of commands (R8) on actuators.

Input: Possible input is (a) knowledge about connected actuators, (b) knowledge about the language by means of which the actuators could be controlled from the storage manager (C4), and (c) support information from the reasoning component (C7) via the support component (C8).

Functionalities: In assistive systems with full automation, the support component (C8) sends support information that triggers actions for actuation in this component. The automatic execution components create the right commands in the correct language to control each relevant actuator individually. Moreover, they communicate possible errors back to the assistive system and user via the support (C8) and information (C9) components.

Output: *On success:* a set of commands for controlling actuators (C13) and their success to the storage manager (C4). *On error:* error information to be sent to the support (C8), information presentation component (C9), and the storage manager (C4) for keeping historic information.

Example: In a smart home, we can send commands via MQTT to open or close the blinds. More complex use cases and IoT devices might require more complex components and connections, e.g., via generated connections [50] and IoT app stores [20].

C13 - Controller (opt.)

Requirements: The controller components fulfills a requirement to control actuators (R15).

Input: They get the commands from the execution component (C12) as input.

Functionalities: In assistive systems which should directly execute the support on actuators, these components perform specific tasks to control actuators, based on commands from the execution component (C12).

Output: Results of task execution, e.g., switching something on and off, moving something, or changing values. The effects of these changes might be sensed by monitoring components (C1).

Example: Actuators are cyber-physical objects that can be controlled by running commands on their software parts,

e.g., to switch a smart light bulb on or off, to open a window blind, or to set an alarm.

C14 - Model Administration

Requirements: The model administration component fulfills requirements to create new models (R10) and to create, read, update, and delete models (R16).

Input: Possible user inputs are (a) language commands for model administration (creating, updating, and deleting of models), and (b) information about the necessary language - for a modeling tool based on this language. If the model administration component is a low-code tool enabling modeling on a higher level of abstraction, it could get graphical user input and commands for model administration. It also gets input from the storage manager (C4) consisting of (c) the information about the already deployed models and (d) model changes.

Functionalities: It supports model administration of an assistive system (not only during the runtime of a model), namely creating, updating, and deleting of models by means of (a) a modeling tool based on language, either with a graphical or textual interface, (b) a low-code tool which allows creating, updating, and deleting model information without providing too many details of the used modeling languages. This tool exists on a higher level of abstraction, being feasible for domain experts without formal software engineering training. Moreover, it includes (c) a component to handle model updates. Such a model update could be provided as a service via the service presentation component (C10) to the user who replies with what he or she wants to change in a model. This model change is then provided via (C11) to the storage manager (C4), and provided to the model update component of (C14) which makes a change in the model. In this case, no explicit modeling tool would be needed.

Output: Its output is the result of performing model administration tasks such as changed or created models communicated back to the storage manager (C4).

Example: The HBMS Modeling tool [72] is used to manage models of human behavior specified in the HCM-L language. Another option is to provide models on a more abstract level, e.g., using form-based interfaces in low-code platforms [25].

C15 - System Administration Component

Requirements: This component fulfills requirements to present services (R7), to allow automatic execution (R8), and to handle goals (R21).

Input: Possible inputs are: (a) language commands for system administration (changing the configuration of the system), and the information about the necessary language for specifying the administration tasks—for a system administration component based on language, (b) low-coding user input for system administration—for a low-coding administration component, and (c) information about the current configuration.

Functionalities: This component allows us to configure the assistive system. This could include, e.g., which services to provide for users, which degree of support should be provided, or other technical administration for the components, models, and data of the assistive system.

Output: Results of performing system administration tasks such as the set of changes in the configuration of the assistive system.

Example: An example is the HBMS Administration component [72] which is used to specify the configuration of the HBMS system for the specific environment, e.g., regarding preferences for different support devices.

C16 - Communication Interface Components (opt.)

Requirements: These components form an interface to external knowledge sources and, thus, fulfill a requirement to monitor the person and her context (R3).

Input: Possible input is all external knowledge, e.g., from ontologies or third-party systems such as information systems or physical systems, which is relevant for human behavior support.

Functionalities: Each interface allows the acquisition of external knowledge needed for the system. This knowledge has to be transformed into the data structures to be handled by the storage manager component (C4).

Output: The updated state of the system via the storage manager component (C4), taking into account the acquired knowledge. This can be further handled, e.g., by the reasoning component (C7) to learn models.

Example: A component that takes external domain ontologies specified by means of OWL2 ontology description language and adds the knowledge from these ontologies to the local knowledge base.

4.1 Assistive system properties affecting decisions to implement the optional components

Some of the proposed components are optional. Figure 5 provides an overview. Which components are used depends on the main properties of the assistive system. We have identified four different system properties, which influence which combination of components should be chosen: the way of data handling, the degree of support, the learning abilities of the assistive system, and the degree of customizability by users.

Data handling. If we handle sensor data, this will create massive amounts of data which will need to be pre-processed. However, dependent on the abilities of the monitoring components (C1), this step might not be necessary if already only pre-processes data from activity recognition systems is stored.

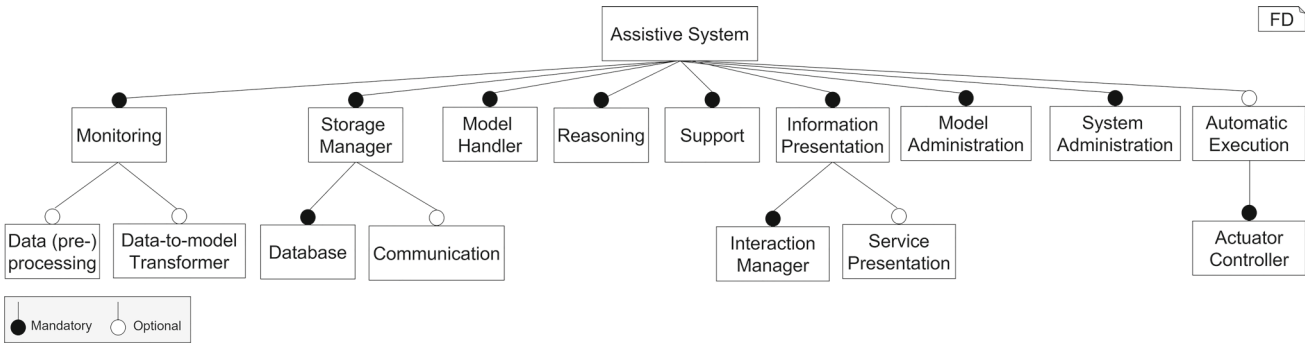

Fig. 5 Feature diagram (FD) representing mandatory and optional components

Degree of support. Assistive systems could provide step-by-step guidance a person has to follow or provide a higher degree of automation where changes within the environment of a person are also executed. Therefore, component(s) to control actuators (C11) and automatic execution component(s) (C12) are optional and only needed in the latter case.

Learning assistive systems. Assistive systems might be operated with learning and support phases [92]. If they start with an explicit learning phase, the system learns by creating models from data, and a component that automatically transforms data into models (C5) is needed. However, models might also be created and added manually which would not stop the assistive system from including learning abilities, e.g., by comparison of sensed data with information in models which is done in C6, the reasoning of new knowledge in C7 based on existing knowledge, and updating or adding new models using the model administration component (C14).

Degree of customizability. If the assistive system should be more customizable by the users, presentation of services (C10) would be needed. The selection of services would be handled by the interaction manager (C11). Clearly, an assistive system would also work without it if the selection possibilities were only restricted to, e.g., a list of possible next steps.

4.2 Application of the approach

We see two main areas of application for our reference architecture: analyzing existing architectures and using the reference architecture as a blueprint for creating a new architecture.

1. When *analyzing existing architectures* we aim at finding gaps in existing architectures to update them to an assistive system covering relevant requirements. Moreover, it is possible to identify problems in existing architectures, e.g., if functionalities are spread over too many components, if functionalities are missing, or if components are not connected. If components are missing, it means that certain requirements for assistive systems are not met. One can even trace back to the requirements which ones are not fulfilled. E.g., if the Data to Model Transformer (C5) is missing, one cannot fulfill the transformation of data into models (R18) and one is not able to fulfill the creation of new models (R10) if the Model Administration component (C14) is missing. Similarly, missing connections between components may indicate that the requirements that are addressed by these connections are not satisfied. Based on the above, we understand the compliance to our architecture as the combination of the structural compliance, i.e., the presence of all required components, and the communication compliance, i.e., the presence of the connections between components. Compliance is treated as a binary function, thus, either a component or connection is compliant with the reference architecture or not.
2. We can use the reference architecture as a *blueprint for creating a new architecture.* During the engineering process of an assistive system, we can use the reference architecture to discuss and reflect on the functionality of each component. By discussing which functional part an implemented component should cover, it is easier to reflect if it covers the basic requirements, and planned connections to other components could be added or deleted.

Within this paper, we are focusing on the analysis of existing architectures. We use the following method: For each component of the architecture under consideration, we have investigated the existing implementation and identified to which component in the reference architecture it corresponds. These findings were mapped to the graphical representations of the system architecture. For a reflection on this method and its limitations, we refer the reader to Sect. 7.

5 Analyzing existing assistive systems

To validate our reference architecture, we map it into real systems, expressing such systems in terms of compliance with

this architecture (as mentioned above, we understand compliance as the combination of the presence of components, and the presence of connections between components). In this section, we describe the results of such mapping for two existing systems from different domains, namely assistance for elderly people and assistance for operators in smart manufacturing.

5.1 Assistance for elderly people

Within the Human Behavior Monitoring and Support (HBMS) project [72], a group of researchers has developed an assistive system to support the elderly in their daily activities in their private homes.

System architecture. Figure 6 shows the main architecture of the developed system. This architecture is designed to make it possible for this system to learn the behavioral knowledge of the person when his/her cognitive abilities are intact and preserve this knowledge by means of domain-specific modeling language HCM-L [62], so it can be reused when these abilities begin to decline. The system provides support in problematic situations (e.g., when the person forgets the next steps or makes a wrong step in an activity of daily life) by suggesting the course of actions based on the previous behavior.

To express the architecture of HBMS in terms of compliance with our reference architecture, for each of its described components, we show the code of the corresponding component of the reference architecture in parentheses; in Fig. 6 they are shown within circles on top of the mapped HBMS components. For each of the described connections between components, we show the codes of the components at the ends of the connection in parenthesis and separate them with a dash: e.g., "(1)–(5)".

The HBMS architecture consists of the following components:

1. *HBMS Modeling Tool* which is used for creating and maintaining HCM-L models **(C14)**. The communication between this tool and the HBMS kernel (to be introduced later) is performed by means of the *HBMS model transfer interface* **(C5)**. The following connection is present: (C14)–(C5).
2. *HBMS knowledge base* that holds both HCM-L models and behavioral data **(C3)**. After coming from the model transfer interface, HCM-L models are converted into the knowledge base format using the *HCM-L-OWL Converter* **(C6)**. The communication between the knowledge base and the kernel is performed through the *HBMS Data Management Subsystem* **(C4)**; within such subsystem, an HCM Storage management component specifically handles storing and extracting models **(C6)**.
3. *HBMS Observation Interface HBMS-OI* obtains the observed behavioral data from heterogeneous external *human activity recognition systems* (HARS) **(C2)**, which, in turn, collect raw data from *sensors*, e.g., installed as part of smart home environments **(C1)**.
4. *HBMS Kernel* which provides the core functionality of the system. By itself, it cannot be mapped to our reference architecture, instead, it contains sub-components for which such a mapping could be established, namely:
   (a) *HBMS Observation Engine* which obtains the recognition data from the observation interface HBMS-OI and converts it into the representation suitable for further processing **(C2)**, *jointly with HBMS-OI and HAR systems*).
   (b) *HBMS Behavior Engine* which matches the arrived behavior data against the current HCM to indicate the position in the model corresponding to the currently performed action, and decide if this action is correct **(C6)**, *jointly with HCM-L-OWL converter and HCM Model storage*); it is also able to predict further actions based on ontological reasoning [57] **(C7)**.
   (c) *HBMS Support Engine* which controls the activities of the assisted users through the support client UI **(C8)**. It converts the match and prediction information obtained from the Behavior Engine into support client UI commands.
5. Multimodal *support client UI*, which translates the commands obtained from the Support Engine into visual or audio cues for the supported person **(C9)**.
6. *Integration client UI*, which collects behavior data from persons when their cognitive abilities are intact; it makes it possible to start and stop collecting activities, to check if the collected sequence of steps is correct, etc. **(C11)**.
7. *Administrator* and *caregiver UI*, which allows the respective users to issue administrative commands and monitor the current status of the system **(C15)**.
8. *Knowledge acquisition interface* connects the system to the external knowledge sources, e.g., it allows the acquisition of domain ontologies to handle concepts related to specific domains or specific categories of users **(C16)**.

Compliance with the proposed reference architecture. Our main findings regarding the compliance of the HBMS architecture to our proposed reference architecture are as follows.

1. All 10 mandatory and 3 out of 6 optional components from the proposed reference architecture are present in the HBMS architecture. The absence of optional components (C10), (C12), and (C13) does not, in our opinion, disqualify the HBMS system from compliance with the reference architecture. This absence is related to the

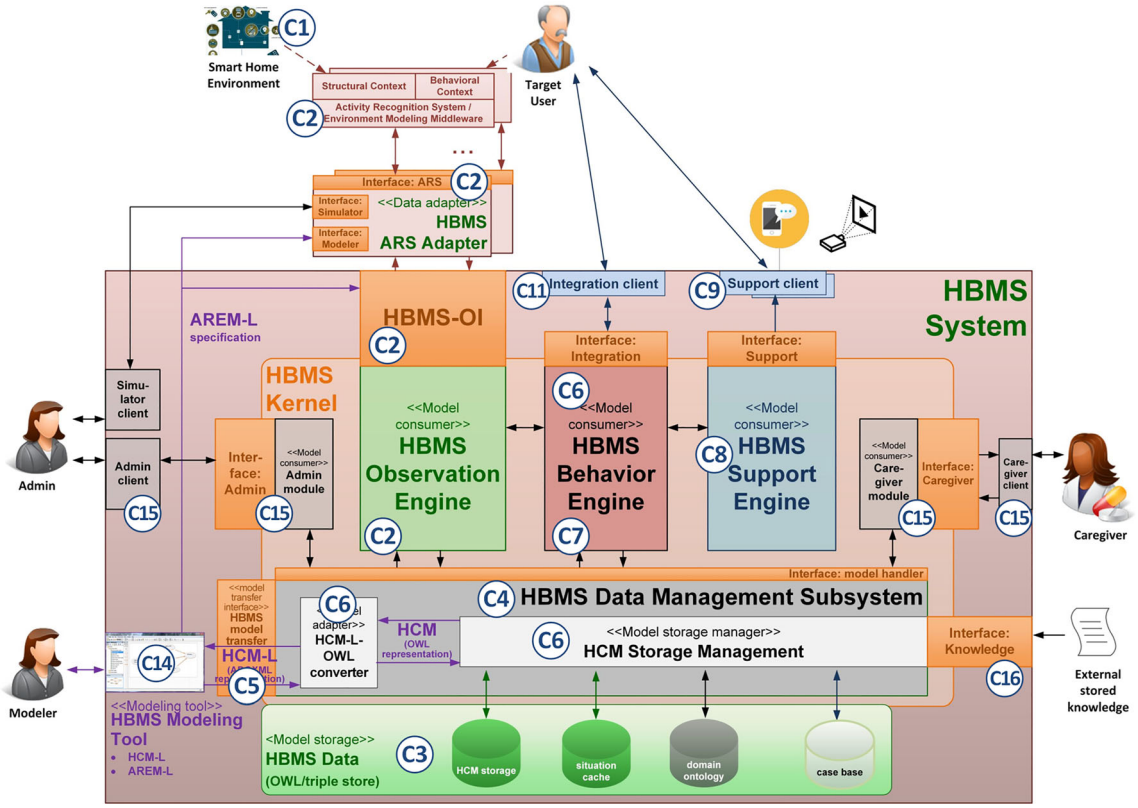

Fig. 6 Mapping of the proposed reference architecture into the HBMS system architecture (adapted from [72])

specifics of the support proposed by HBMS (e.g., this support is not designed to use actuators or automated tasks).

2. The mapping of the remaining components is not always one-to-one:
   - the data (pre-)processing component **(C2)** is mapped into the external HAR system, its adapter, HBMS-OI interface, and the HBMS Observation Engine.
   - The model-handling component **(C6)** is mapped into the HCM-L-OWL converter, the model storage component of the Data Management Subsystem, and into the subset of the HBMS Behavior Engine responsible for handling behavioral models.
   - the administrator client component **(C15)** is mapped into admin and caregiver clients, and the corresponding interfaces;
   - both model-handling **(C6)** and reasoning **(C7)** components are mapped into the HBMS Behavior Engine.

   Discovering such mappings could help in indicating possible design problems. For example, one-to-many mapping of C6 could indicate the lack of unified model management implementation in HBMS. On the other hand, one-to-many mapping of C2 could be justified as the HBMS system was designed to get the observation data from heterogeneous HAR systems, so introducing separated HBMS-OI interface with HAR-specific adapters allowed for extra flexibility while dealing with such systems [90].

Communication Compliance. We have checked each connection between the components in the reference architecture and their occurrence in the architecture within Fig. 6. First of all, as several optional components (C10, C12, C13) are not present, all connections referring to them and from them are not present as well. Similarly to missing optional components, this, in our opinion, does not disqualify HBMS from being compliant. On the other hand, some connections involving non-optional components, namely C1 to C5, C2 to C5, C4 to and from C8, C4 to and from C14, C6 to and from C7, C11 to C4 and C11 to C7, are not represented in the architecture directly. Still, most of these connections can be followed via other components of the system. For example, the connection between C4 (implemented by HBMS Data Management Subsystem) and C14 (implemented by the HBMS Modeling Tool) can be followed through C5 (implemented by HCM-L Model Transfer Interface) and C6 (HCM-L-OWL converter). The exception here is the connection from C1 to C5 which is missing from the HBMS architecture, as the HBMS Modeling Tool does not obtain

the input from the Smart Home Environment. This indicates the possible way for the improvement of the architecture.

To sum it up, the analysis of the HBMS system has shown that all mandatory components and all except one mandatory connection defined in the proposed reference architecture occur directly in this system or can be followed through intermediate components. Thus, we conclude that the real-life HBMS system complies almost fully with the proposed reference architecture. This fact of compliance, together with the fact, that the perceived mapping inconsistencies helped to identify real design problems in the HBMS system, contribute toward the validation of the proposed architecture.

5.2 Assistance for operators in smart manufacturing

Dalibor et al. [26] use a Model-Driven Software Engineering (MDSE) approach to create digital twins and their cockpits. They define a digital twin as "a set of models of the system, a set of digital shadows and their aggregation and abstraction collected from a system, and a set of services that allow using the data and models purposefully with respect to the original system" [26]. Those software systems include models, data, and services to interact with a Cyber-Physical Production System (CPPS) for a specific purpose. Such digital twins might include assistive services to provide support for operators in production processes, e.g., on the shop floor.

System architecture. Figure 7 shows a reference architecture for such digital twins, which realizes the MAPE-K feedback loops [5] for self-adaptive systems. The researchers use a MDSE approach to generate the adaptive look from UML Class Diagrams (CDs) and MontiArc models [26] and the visualization backend and frontend from CDs and GUI models [37]. The architecture consists of the following components:

1. The *CPPS* includes sensors **(C1)** which detect specific property changes of the monitored system, e.g., temperature and pressure sensors in various steps of an injection molding process [15].
2. The *Data Lake* **(C3)** includes a set of databases that could be used as information sources for the assistive system as well as the visualization.
3. The *Data Processor* uses data from the data lake and aggregates the data about the current state of a CPPS **(C2)** into a digital shadow, which is "a set of contextual data traces and/or their aggregation and abstraction collected concerning a system for a specific purpose with respect to the original system [15]". This step might also include the *transformation from data into models* **(C5)**.
4. The *Evaluator* **(C6)** uses this aggregated data and models to monitor the CPPS state and check if the CPPS operates as intended. If any anomalies are detected, it creates goals that specify the intended CPPS state.
5. The *Reasoner* **(C7)** finds a solution on how to change the current state to reach the intended state by using the given goal.
6. The solution is handed over to the *Data Distributor* **(C4)**, which stores relevant data in the *Data Lake* **(C3)** and provides relevant information to the *Data Connector* of the visualization part of the digital twin which shows the solution to an operator so that he can perform it.
7. The solution is also handed over to the *Executor* **(C12)**, if a change in the *CPPS* could be performed automatically. The *Executor* translates the provided solution into concrete CPPS settings and performs these using the components to control actuators **(C13)** of the CPPS.
8. The solution to be shown to the user is passed on from the *Data Connector* **(C4)** to the *Logic Processor* **(C8)**, which requests additional support information from the *Data Lake* based on the solution from the reasoner.
9. The *Data Aggregator* sends commands to the *Logic Processor* and can return aggregated view models which fit different support devices.
10. The *Data Aggregator* sends view models to the *Frontend* and presents **(C9)** them to the user. If data is sent from the *Frontend* to be stored in the *Data Lake* the commands are passed on by the *Data Aggregator* to the *Logic Processor* and *Data Connector* sends the according queries.

Compliance with the proposed reference architecture. This mapping has shown which components of the digital twin can be used to provide assistive functionalities within a digital twin of a CPPS. However, there are some aspects missing from the reference architecture to provide full support.

- The architecture within Fig. 7 provides no detailed information about the *Frontend* components. Whereas the presented GUI shows selectable services **(C10)**, e.g., to stop the machine, and ways to interact **(C11)**, there are no details about these functionalities within the architecture model. It has to be further defined to describe them.
- The architecture model shows no *Model Administration* **(C14)** or *System Administration* **(C15)** component. By now, administering the models is only possible at design time and not at runtime. From a software engineering point of view, those two components would be especially helpful for the maintenance of the running application and should be added.
- Providing *Interfaces to other data and knowledge sources* **(C16)** is also relevant for digital twins. However, the proposed architecture does not describe them explicitly but implicitly, as those knowledge sources are a part of the

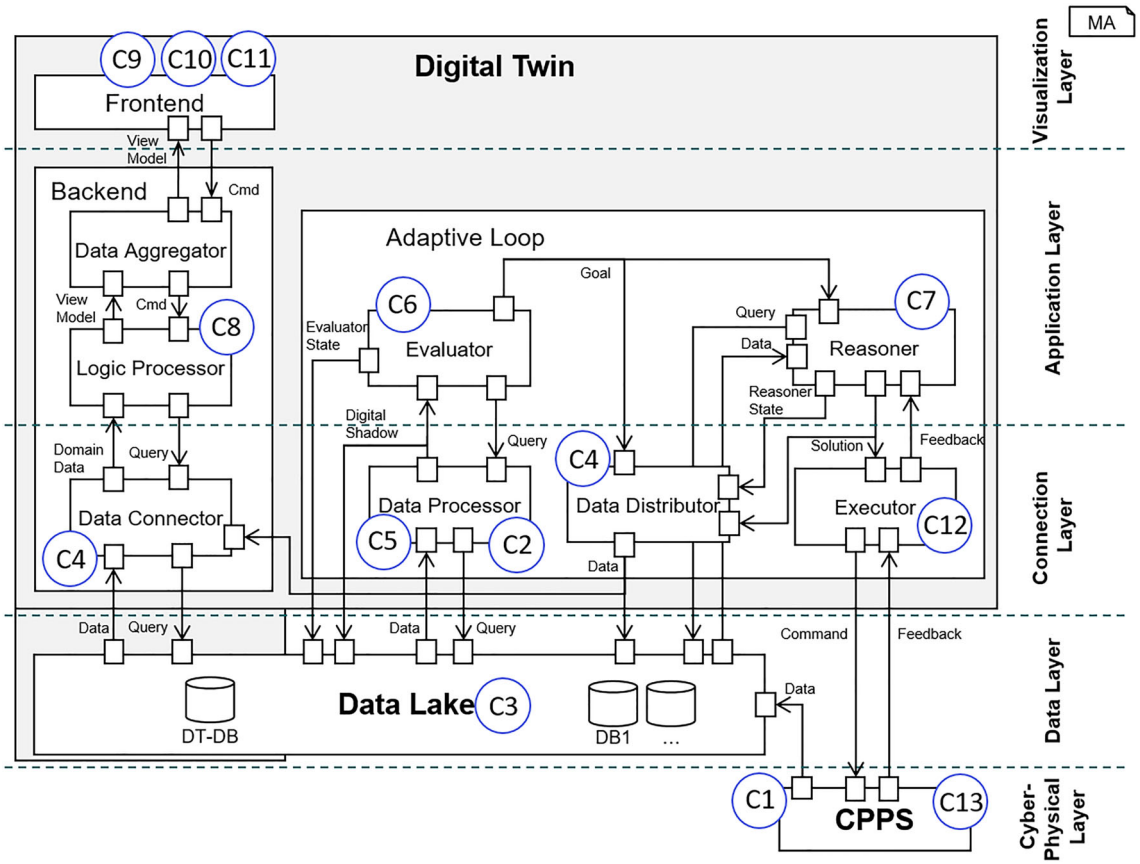

Fig. 7 Mapping of the proposed reference architecture into the digital twin system architecture represented as MontiArc (MA) model (adapted from [26])

Data Lake. It would be better to describe those interfaces explicitly in the architecture.

Communication Compliance. We have checked each connection between the components in the reference architecture and their occurrence in the architecture within Fig. 7. As several components do not exist, all connections referring to them and from them do not exist. Moreover, the *Data Distributor and Data Connector* (C4) lacks several connections to other components, and several connections are not modeled directly but indirectly through other components, e.g., C8 to C9, C11 to C4, C5 to C4, or C2 to C4. This shows that the current approach stores less information that might be relevant for assistive systems and is more restrictive regarding which components interact. To add assistive functionalities within this architecture according to our reference architecture, several connections need to be added.

To sum up, the analysis of this system has shown that several of the components defined in Sect. 4 occur in this system. As it is now, it does not provide all the functionalities needed for an assistive system. However, it became clear what components and data flows are missing and need to be added to be able to provide assistance for operators.

6 Related work

As we stated in the introduction, we know of no prior research specifically targeting reference architectures for assistive context-aware systems which use models at runtime, so we consider here

1. the publications on reference architectures for assistive systems not explicitly using models at runtime;
2. the publications dealing with the properties of target systems, in particular
   (a) the publications on reference architectures for context-aware systems;
   (b) the publications on reference architectures for self-adaptive systems.

We compared reference architectures described in the literature to our architecture, the results of this comparison are shown in Table 6. For brevity, we limited ourselves to comparing only components:

– if the architecture contains the specific component, the corresponding cell is marked with a plus sign;
– if the component is defined partially (e.g., it offers a part of the functionality or it is a generic component that has to be instantiated as a more specific component to match our architecture), the cell is marked as "±",
– if the component is not present in the compared architecture, the cell is marked with a minus sign.

The results of the comparisons are grouped by the category of the compared system (assistive systems, context-aware systems, self-adaptive systems).

Below we elaborate on the systems which were used in comparison.

6.1 Reference architectures for assistive systems

Surveys of reference architectures for assistive systems not necessarily addressing using models at runtime are available in [29, 33]. Below we consider some specific works.

Liu et al. [56] presented a reference architecture of assistive hardware devices and services aiming at improving the quality of life of older people. Compared to our architecture, it is more hardware-oriented, provides fewer functions (no support for actuators, no reasoning mechanism, etc.), and does not explicitly include model-handling components (it, however, mentions dealing with "models, rules, and policies" without elaborating on that).

El Murabet et al. [30] proposed RAFAALS reference architecture for AAL systems as an architectural model based on the "platform as a service" (PaaS) principle. It uses BPMN models to drive the execution of services, though it does not provide means for creating or maintaining such models, also, it does not define interaction management components.

Oestreich et al. [78] proposed the adaptive workflow architecture for digital assistance systems (mostly aiming at supporting instructional systems, i.e., not limited to AAL). It is based on runtime BPMN models, including such components as Modeler, Editor, and Canvas to support the creation of such models, and Process Engine for their execution. The other component groups include Adaptive Assistance (with Companion component), Process Monitoring and Analytics (with Dashboard component), and Adaptation (including, besides Process Engine, the Adaptation Component and plugins). It does not include actuators, does not use any external knowledge, and is not explicitly positioned as a reference architecture.

Augusto et al. [10] proposed the system architecture for smart environments; their treatment of the system architecture can be adapted to provide a reference architecture for the specific domain. Most of its components are also defined by our architecture, still, it lacks an explicit treatment of model support or dealing with external knowledge.

Personal Connected Health Alliance (PCHA) published Continua Design Guidelines that provide a reference architecture (Continua End-to-End Reference Architecture) for interoperable healthcare ecosystems [100]. This architecture includes the concepts of Personal Health Device, Personal Health Gateway, Health, and Fitness Service, and Health Information Service, the communication between implementations of these concepts is performed by means of Personal Health Devices Interface, Service Interface, and HIS Interface. It introduces a set of models: Domain Information Model, Communication Model, and Service Model, but the correspondence between these models and the models@run.time paradigm is not shown.

ECHONET consortium proposed the ECHONET Lite reference architecture to support interoperability of home appliances [51]. It includes the concepts of Home Appliance, Home Gateway (connected by means of ECHONET Lite protocol), Cloud Server, and Application (connected by means of ECHONET Lite Web API). One of the implementations of this architecture is introduced in [74]. Pham et al. [82] described a concept that supports the interoperation of the implementations of both PCHA and ECHONET Lite reference architectures. As for PHCA, ECHONET Lite also does not rely on models@run.time; in particular, the above implementations do not provide any model handling or interaction support.

Garcés Rodríguez, Zanin Vicente and Nakagawa [34] present a software architecture for healthcare systems to assist patients with diabetes mellitus. This architecture is generalized in [85] as HomecARe: a reference architecture for chronic disease management at home. These architectures include additional services for, e.g., quality checks and describe various needed databases as information sources in detail. They also include a business process model for activities regarding the patient, though the model administration component is not described in detail. In addition, they lack details on the support devices, include no actuators or automatic execution, and are very specific to their application domain. Still, HomecARe includes the richest set of components among all architectures we reviewed, though the way of presenting it (as a connection-less overview diagram supplemented by a set of smaller-scope diagrams that describe component interaction) makes it difficult to check the communication compliance.

Among less recent reference architectures targeting AAL domain are Feelgood [44], MPOWER SOA [73], OpenAAL [101], and PERSONA [94]. Here, the richest set of components is provided by PERSONA, which, however, completely lacks support for administrative components and does not define actuators.

To sum it up, most of the above approaches, not targeting models@run.time directly, do not support model-related components (C5, C6, and C14) in full. Many of them, how-

Table 6 Component definition in state-of-the-art reference architectures

References	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15	C16
Reference architectures for assistive systems																
[56]	+	+	+	±	-	±	−	−	−	−	−	+	+	−	−	−
[30]	+	±	±	±	±	±	±	±	−	−	−	−	+	−	−	−
[78]	−	−	±	+	−	±	±	+	+	−	+	−	−	+	+	−
[10]	+	+	±	±	+	±	+	±	+	−	+	−	+	−	−	−
[100]	+	±	+	±	±	±	±	−	+	−	−	−	−	−	−	−
[74]	+	+	±	±	±	−	−	+	−	−	+	+	+	−	−	−
[82]	+	+	+	±	±	−	−	−	−	−	−	−	−	−	−	−
[85]	+	+	±	+	+	+	±	±	+	±	+	+	−	±	+	+
[44]	±	±	+	−	−	−	±	+	+	−	−	−	−	−	−	+
[73]	+	±	+	+	−	±	+	−	+	−	−	−	−	+	−	±
[101]	+	+	−	+	−	±	−	−	−	−	−	+	+	−	−	+
[94]	+	+	±	+	±	±	+	±	+	+	+	+	−	−	−	−
Reference architectures for context-aware systems																
[46]	+	+	±	±	+	+	−	−	−	−	−	−	−	±	−	+
[55]	+	+	+	+	+	+	±	±	±	−	±	−	−	−	−	±
[11]	+	+	−	−	+	−	+	−	−	−	±	−	−	−	−	+
[88]	±	±	−	−	+	±	±	−	−	−	−	−	−	−	−	±
[6]	+	±	±	±	±	+	+	±	−	−	−	+	−	+	−	+
[3]	+	±	+	±	±	±	±	+	−	−	−	+	−	−	−	−
Reference architectures for self-adaptive systems																
[39]	±	±	−	−	±	+	−	±	+	−	+	−	−	+	±	±
[13]	±	±	−	−	±	±	±	±	−	−	±	−	−	−	−	±
[79]	+	+	+	+	±	+	±	±	+	±	±	±	+	+	±	−
[91]	±	+	+	+	±	±	+	±	−	−	−	+	−	−	−	−
[102]	−	±	−	−	±	+	−	+	−	−	+	±	−	−	−	−

ever, support dealing with models and rules partially, usually by mentioning that such structures can be used to drive the support, without discussing these issues in detail. The level of support for other kinds of components varies by solution, we can state that getting the data from sensors (C1 and C2) is supported much more widely than providing interaction support (C11) or dealing with actuators (C13).

6.2 Reference architectures for context-aware systems

Following Hu, Indulska, and Robinson [46], there are three main approaches to building context-aware applications: (1) without application-level context model, (2) with an implicit context model (3) with an explicit context model. The latter is the most relevant for our reference architecture. Roda et al. [84] provide a study on software architectures for context-aware systems. It is possible to implement their guidelines for modeling, retrieving, and managing context with our reference architecture. A specific reference architecture for context-aware systems is presented by Lewis et al. [55], the architectural components of such systems are also addressed in detail by the CASA architecture by Augusto et al. [11] and by a context-aware architecture for personal healthcare smart gateways by Santos et al. [88].

In addition to the more generic context-aware approaches listed above, the architectural approaches for context-aware Internet of Things (IoT) systems (reviewed by Perera et al. in [81] and represented by a context- and self-awareness architecture for the IoT by Arnaiz et al. [6]) and the approach to model and execute context-sensitive production processes by Alexopoulos et al. [3] would also fit into our proposed reference architecture.

Among the approaches to handle context-awareness, the reference architecture by Lewis et al., and, to lesser degree, the IoT architecture by Arnaiz et al. offer the richest set of components, coming closest to our approach. The problem with these and other approaches of this kind is that they lack assistive services for the supported human (not including, or partially including C9 and C11), and do not support actuators.

6.3 Reference architectures for self-adaptive systems

A broad variety of self-adaptive systems implement the MAPE-K feedback loop [5, 87]. Addressing four MAPE-K steps is a goal of a typical set of components that are found in assistive systems. In particular, Grua et al. [39] propose a reference architecture targeting the development of personalized and self-adaptive e-health apps for smartphone-centered environments which includes a specific set of components supporting self-adaptation by implementing MAPE-K loops (e.g., user-driven adaptation manager). The problem with such steps is that they typically do not focus on the human-in-the-loop to be assisted but on the automatic change and execution of changing goals.

Further specific examples of reference architectures targeting self-adaptive systems are RA4Self-CPS [79], a reference architecture for resilient behaviors control by Bemthuis et al. [13], HAFLoop [102], and self-adaptive digital twin reference architecture by Splettstosser et al. [91]. Among these approaches, RA4Self-CPS includes the richest set of components, coming close to our architecture; it, however, lacks external knowledge support and does not fully support interaction support or actuators, whereas self-adaptive digital twin architecture by Splettstosser et al. does not address human support at all (not implementing C9 and C11), and does not include administrative components, actuators, and external knowledge support. Finally, HaFLoop, being a reference architecture specifically targeting the support for adaptive feedback loops, is not supposed to be used to design standalone architectures, so it does not include any explicit support for sensors, storage, or administrative components.

6.4 Final remarks on the related work

From the comparison in Table 6, it can be seen that no existing architecture includes all the components proposed by our reference architecture. The reference architectures closest to our approach, such as HomecARe, PERSONA, the context-aware architecture by Lewis et al., or RA4Self-CPS, still lack the support of important components, such as actuators, explicit model handlers, or interaction support components. In our opinion, this further emphasizes the novelty of our approach.

We also observed that the supported subset of components depends on the type of the system: for example, context-aware systems put more effort into handling models (C5 and C6), whereas self-adaptive systems are more likely to implement support (C8) and interaction management (C11) components.

One further observation is related to the components which are mentioned the least. Among them, model and system administration (C14 and C15) are rarely presented as, in our opinion, academic solutions rarely reach a high technology readiness level and therefore do not consider aspects such as the long-term maintenance of systems. Service presentation (C10) is rarely mentioned explicitly, as, in our opinion, architecture authors often do not feel the need to separate it from the rest of the user interface, though its functionality may be still present. Also, the support for actuators (C13) is limited, which may diminish the quality of system-to-user interaction.

7 Discussion

To date, we have not found in the literature any descriptions of a domain-independent reference architecture for complex assistive systems that use models at runtime. Within this paper, we have introduced such an architecture and compared it with two concrete implementations. Clearly, we have made some assumptions in this process, our approach has limitations and it could be further extended.

7.1 Guidelines and best practices

Our guidelines for analyzing and developing systems are based on the blueprint already presented in Sect. 4.2. In particular, for *analyzing systems*, we propose to establish a checklist (e.g., in a form of the Excel spreadsheet), which can be followed against the current system architecture. Such a checklist may include the list of components, together with the indication of components providing the input and output of each component (taken together, such input/output specifications actually define the relationships between components). While filling the checklist, the analyst may specify the name of the system component which implements the particular component of the reference architecture (if it exists), and the name of the component which serves as an input or output for the specific system component (again if it exists).

If deviations are detected in real-world scenarios, e.g., components or connections are missing, the development team can use the reference architecture to discuss (1) how the requirement related to a component is related to their concrete system requirements and (2) how adding each of the relationships from the reference architecture would influence the data flows within their implementation. Moreover, we suggest that the team evaluating the architecture includes developers who are very familiar with the implementation to make these discussions easier.

For *developing systems*, we suggest the developers use the reference architecture as a blueprint and add additional functional requirements, e.g., on concrete modeling languages to use during runtime, from the application domain, or technical details. Moreover, non-functional requirements need to be explored for the specific use case. These additional

requirements then have to be reflected if additions are needed and which concretizations have to be done in the technical architecture of the system. Moreover, the developers have to explore if the newly developed system has to be connected to systems that already exist.

Resource Reduction and Development Effort. We have developed a case study for supporting the cooking process in a smart kitchen [69]. It took one student app. 4 months to come up with the first implementation of the whole assistive system. This included (a) setting up the system architecture, (b) the creation of two new Domain-Specific Languages (DSLs) to be handled as models at runtime, a task and a context modeling language, (c) a survey to get feedback on the design of the system, (d) creating models, pictures, audio support, and making hand-written additions to the partly generated system, and (e) performing an evaluation with end users to get additional feedback. Thus, we can state that the development effort was relatively low to realize such a system. The proposed reference architecture helped to reduce this effort as follows. (1) As the detailed *descriptions of each system component*, its in-, and output as well as the functionalities provided a more systematic way of *structuring the code* in the first place, this enabled the creation of a *basic structure* fast. (2) The described *requirements* were used as a starting point to further refine the requirements for the domain-specific application. (3) As some *components are marked as optional and serve specific system properties* (see Sect. 4.1), this helped the student to focus on the implementation of the most relevant components and connections for the particular application domain and its use cases. (4) In cases where it was *unclear in which system component a certain functionality should be included* or where specific data and models should be handled, the descriptions helped to make decisions faster. This is mainly because our reference architecture follows the separation-of-concerns principle and suggests a high cohesion within each component. (5) When extending the implementation, this separation of concerns helped to identify which components needed additional subcomponents to realize additional functionality. (6) When *refactoring* the code during the development process, again the functionality descriptions and comparing the actual input and output of components with the suggestions in the reference architecture helped to identify if certain implementation details should be moved to other components.

A quantitative comparison of the effort needed for an implementation without the reference architecture is, however, a common challenge we are facing in software engineering research: Developers have different skills and learn when doing the same task more than one time, the realized domains and software characteristics differ in size and complexity, and requirements change during implementation. Thus, we are currently applying the reference architecture to several applications and case studies to be able to quantify the development effort. However, as we are also using model-driven engineering approaches for the development of such systems, this could also have a strong influence on the development effort.

The presented reference architecture describes an assistive system on a higher level of abstraction. If one wants to further explore the benefits of reference architectures, such as the reuse of common functionalities and configurations in the generation of systems, reduced risk through the reuse of proven architectural elements, or better quality [61], one needs to break the reference architecture down to concrete domains and provide concrete reusable implementations for the components of the reference architecture.

7.2 Assumptions and limitations

Assumptions. Clearly, only the graphical representation of concrete system architecture (boxes and arrows) alone is not enough to analyze a system. Thus, we assume (1) that architecture models are detailed enough to identify missing components and data flows, and (2) that developers which use our reference architecture are knowledgeable enough about their concrete implementation to be able to evaluate missing aspects. Thus, our approach allows developers to identify ways to extend their implementation toward providing more functionality for assistive systems.

Application to different domains. We have shown the applicability of our approach for examples from two non-related domains. Thus, we assume that our approach is generalizable and can be used to analyze architecture models from assistive systems of different domains. Clearly, each domain has to be further detailed, as different monitoring components, actuators, and support services are needed.

Author involvement in the development of the concrete implementations. The authors developed the first analyzed system (HBMS) together and one author was a member of the team developing the second system (digital twin). However, the second system was initially developed without assistive functionalities in mind as the research focus was on self-adaptive systems and on understanding how MDSE could be applied to digital twins. This involvement might influence the analysis process; however, it also enabled a deeper analysis of the system architectures as the authors know more details in comparison with what is written in research papers about one specific architecture.

Limitations of the communication structure. The proposed reference architecture has no limitation regarding the communication structure of the assistive system. It would work for monolithic, distributed, agent-based, or client–server architectures. Monitoring components (C1) and actuators (C13) are clearly independent components in any kind of architecture, which just need any kind of connection to the related components. We assume that C9-C11 are a part of the

support devices. However, there is also no restriction on the kind of communication with those components.

Restrictions on the size or architectural style. This reference architecture provides no restriction on the number of lines of the code of the concrete implementation, on how many sensors are needed, or how many activities have to be supported to call an application an assistive system. Moreover, we do not restrict the reference architecture to a specific architectural style, as the concrete implementations could differ in how distributed components are.

Limitation of expressiveness. Defining systems on a component level using textual descriptions and analyzing them based only on their architectural models has only limited means to express if a component really provides what is needed for a specific assistive use case. For example, if we have an *Information Presentation* component in the architectural description, this does not ensure that there exists an acoustic way of information presentation for an operator in a noisy environment. Such aspects have to be further analyzed regarding the specific requirements from the domain.

Reference architecture validation vs. formal compliance checking. Clearly, how to represent reference architectures and how to check the compliance of reference architectures to concrete implementations are commonly known problems [19]. Approaches such as described by Bucaioni et al. [19] require component connector models of the reference architecture and the concrete implementations. They describe an approach combining model transformation and weaving techniques allowing for the automatic conformance checking of concrete architectures. This is based on various types of conformance, e.g., if an architecture conforms to constraints, rules, and characteristics of the reference architecture, the architecture language grammar, architectural styles, and additional constraints. These approaches have other goals in comparison with our approach: If the goal is to identify misalignment between reference and concrete architectures as one of the main causes of architectural technical debts or to define Software Product Lines (SPLs) where products must conform to the architecture of the related family, it is important to strictly follow a reference architecture. In more general, [52] suggests a more formal approach to define conformance between two models without explicitly referring to reference architectures. Following [52], a conformance relation is a binary, reflexive, transitive relation that describes whether the concrete model is a concretization of a reference model. In addition to this, concrete conformance rules need to be defined for each language, e.g., architecture description languages.

Deeper exploration of different kinds of runtime models. Systems that use different types of models at runtime might require additional subcomponents handling the specific model types within the Model Handler (C6). We have already explored how to use behavior models [2], task and context models [69] as well as goal models [70] within assistive systems. However, Szvetits and Zdun [93] list significantly more kinds of models existing for models at runtime approaches, e.g., observation models, feature models, aspect models, Abstract Syntax Trees (ASTs), or safety models. Thus, an exploration of these different kinds of modeling approaches could be an interesting research topic.

Requirements elicitation. As mentioned in Sect. 2.1, we followed the ProSA-RA process while designing our reference architecture. While following this process in general, and, in particular, conducting its requirements elicitation step, we did not elaborate on the requirements elicitation process in detail, limiting ourselves to obtaining requirements based on the domain-specific definitions found in the state-of-the-art literature. The reason is that such a process should rely on the generic standards for collecting requirements for the assistive systems which rely on using models at runtime, but we are not aware of all standards of every possible application domain in the necessary detail.

Requirements quality. As a result of following the process based on the existing definitions of reference architecture and assistive systems, we got a set of generic requirements. We argue that such requirements correspond to our goal better as we are proposing a generic architecture, not tied to the specific technology or application domain. In particular, we argue that, e.g., by including the storage manager as a technology-independent interface to the storage component, we eliminated the need to specify the requirements for the architecture to ask for the specific type of storage or the specific storage technology to be implemented. Checking the requirements with experts would be one possibility. This requires finding experts who are familiar with different implementations of and domains for assistive systems.

Vague requirements. The presence of vague requirements (to some extent) hinders the external validation and replication of the study. E.g., the requirement to create new models without mentioning which types of models these are, or to enable person-to-system interaction without mentioning how and with which technologies. In our opinion, this vagueness is inevitable, as some of the requirements must be kept general to allow for heterogeneous implementations in different application domains and with different technologies. They will become more concrete if a domain-specific implementation is considered based on the reference architecture and the requirements are further detailed with the specific end users of the assistance system. To mitigate this risk, we have added a comparison with components in existing reference architectures covering some of the proposed aspects in Sect. 6. However, this does not completely resolve this limitation.

Validating the architecture by asking the experts. While following the ProSA-RA process in most of its activities, we deviated from this process by omitting the use of the FERA checklist [89] for evaluating the resulting architecture as we

found it overly detailed (consisting of 93 questions grouped into several categories) and too relying on the specific knowledge possessed by the experts. As our architecture is based on the knowledge belonging to multiple domains, we expect difficulties with recruiting unbiased experts which possess the knowledge belonging to all these domains. We also expect difficulties in persuading experts from practice to invest their time for the whole set of questions belonging to the checklist. Still, in future, we plan to organize the evaluation based on the modified version of the checklist taking into account these considerations.

7.3 Architectural styles

Architectural styles, such as Client/Server, Microkernel, or Microservices are often defined as parts of specific reference architectures [76]. We, however, decided to make our architecture style-independent, as the fundamental property of the systems compliant with it is that they have to be based on runtime models, and enforcing this property does not depend on selecting a specific architectural style. As a result, we limit ourselves to checking component and communication compliance, where required components and their connections can be implemented by means of applying a specific architectural style.

In fact, we consider applying such a style to be a part of the process of creating an *implementation architecture* of the system, which is a blueprint for its implementation by means of the specific set of technologies. Different such architectures may be further derived from the compliant model-based architecture, however, this process is outside the scope of our work.

7.4 Non-functional requirements

To create the reference architecture, we have only considered functional requirements, since we treat non-functional requirements as application- and technology-specific. Those are addressed by deriving the specific implementation architecture of a domain-specific system from the compliant model-based architecture, and, in particular, by applying a specific architectural style, which we consider as outside the scope of our work.

Still, non-functional requirements as quality attributes significantly impact assistive systems. Thus, we describe their relationship with assistive systems and provide some examples.

Security. Assistive systems should be secure, i.e., they have to protect their data against attacks or unauthorized access. Depending on the technical realization and the use of standardized techniques for ensuring security, it might reflect differently in concrete system architectures and, thus, has to be seen as a cross-cutting topic.

Privacy. As assistive systems are human-centric systems [41, 47], they partially need the private information of their users. Using this information enables assistive systems to be adaptable to specific user needs. To support the privacy-aware handling of data, we can rely on privacy-by-design principles [21] in the engineering of assistive systems. Privacy cannot be centered within one specific component in the software architecture but is again a cross-cutting topic. We can introduce privacy checkpoints [59, 66, 68] within the software architecture in each point where data passes happen. E.g., from monitoring components (C1) to the data (pre)processor (C2), and from any component to and from the storage manager (C4) and the data and model storages (C3). Thus, such non-functional privacy requirements have more influence on the communication between components than the concrete components of the reference architecture. To further elaborate on this idea, further studies are needed considering different privacy checkpoints in a concrete assistive system architecture.

Usability and Adaptation. Depending on the relevant user groups, an assistive system has specific requirements for the user interface and interaction, e.g., elderly users might need larger fonts, more contrast, or louder voice output. Moreover, depending on the application domain different user interfaces might be relevant, e.g., smartphones, tablets, AR glasses, speakers. This heterogeneity in user interfaces leads to a variety of different non-functional requirements on how they could be adapted and what would be well usable for their users. This will influence what information is collected by the support component (C8), what and how information is handled by the information (C9) and service presentation component (C10), and what interaction is possible (C11). Thus, when applying the reference architecture to specific use cases, one has to check whether this is realized within the functionality of these components.

Performance and Scalability. For assistive systems, it is important to provide support in a reasonable time. Whether support should be provided immediately or if there is more time to react depends again on the concrete use case. Scalability requirements describe what workloads an assistive system should be able to handle to meet performance requirements. To scale an assistive system (horizontal and vertical) depends again on concrete use cases and what amount of interaction to expect from them. Whether these non-functional performance requirements could be met depends on the concrete technological realization and not on the reference architecture.

Compatibility. Whereas there might exist assistive systems that could work as a stand-alone application, in practice, they will often coexist with other systems in the same environment. This aspect again relies on the concrete technological realization.

Maintainability. [61] report on reduced maintenance costs due to the use of reference architectures and [80] on improving the maintenance of software with the use of model-driven approaches. Having a clear reference architecture supports the separation of concerns by avoiding redundancy. However, non-functional maintainability requirements are again not explicitly reflected in a reference architecture.

To sum up, different kinds of non-functional requirements will be added to the list of requirements for a specific realization of an assistive system. These might translate to functional requirements, e.g., which encryption method to use, with which firewall or authentication system an assistive system has to be compatible, or how many servers are needed to be able to handle the workload. Thus, they have to be considered e.g., in the implementation architecture of a system.

8 Conclusion

Within this article, we have shown how to develop a reference architecture for model-based assistive systems from requirements. We have introduced relevant components and data flows in detail and shown how these components could be identified in architectural models of existing assistive systems.

This reference architecture with its components allows us to (a) evaluate if a system is an assistive system and (b) make suggestions on which components have to be added to an existing architecture if it does not yet provide these assistive functionalities. Our reference architecture reduces the effort for development and helps to make high-level design decisions about which components should be developed.

In the future, it might be interesting to see how MDSE [98] with models at design time and automation with code generation, might influence models at runtime [14] for assistive systems. Moreover, we need to analyze the useability of different modeling languages, such as goal modeling [70] for their usefulness in assistive systems and if they require a change in the system architecture.

Acknowledgements Funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2023 Internet of Production - 390621612. Website: https://www.iop.rwth-aachen.de

Funding Open Access funding enabled and organized by Projekt DEAL.

References

1. Abrahão, S., Bourdeleau, F., Cheng, B., Kokaly, S., Paige, R., Stöerrle, H., Whittle, J.: User experience for model-driven engineering: challenges and future directions. In: ACM/IEEE 20th Int. Conf. on Model Driven Engineering Languages and Systems (MODELS), pp. 229–236 (2017). https://doi.org/10.1109/MODELS.2017.5
2. Al Machot, F., Mayr, H.C., Michael, J.: Behavior modeling and reasoning for ambient support: HCM-L Modeler. In: Proceedings of the International Conference on Industrial, Engineering & Other Applications of Applied Intelligent Systems (IEA-AIE 2014), Lecture Notes in Artificial Intelligence (2014)
3. Alexopoulos, K., Makris, S., Xanthakis, V., Sipsas, K., Chryssolouris, G.: A concept for context-aware computing in manufacturing: the white goods case. Int. J. Comput. Integr. Manuf. **29**(8), 839–849 (2016)
4. Alvarez, M.L., Sarachaga, I., Burgos, A., Estévez, E., Marcos, M.: A methodological approach to model-driven design and development of automation systems. IEEE Trans. Autom. Sci. Eng. **15**(1), 67–79 (2018). https://doi.org/10.1109/TASE.2016.2574644
5. Arcaini, P., Riccobene, E., Scandurra, P.: Modeling and analyzing mape-k feedback loops for self-adaptation. In: IEEE/ACM 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems 2015, pp. 13–23 (2015). https://doi.org/10.1109/SEAMS.2015.10
6. Arnaiz, D., Vila, M., Alarcón, E., Moll, F., Sancho, M.R., Teniente, E.: Relating context and self awareness in the internet of things. In: International Conference on Cooperative Information Systems, pp. 384–402. Springer (2023)
7. Arning, K., Ziefle, M.: "Get that Camera Out of My House!" Conjoint Measurement of Preferences for Video-Based Healthcare Monitoring Systems in Private and Public Places. In: Geissbühler, A., Demongeot, J., Mokhtari, M., Abdulrazak, B., Aloulou, H. (eds.) Inclusive smart cities and e-health, *LNCS*, vol. 9102, pp. 152–164. Springer (2015). https://doi.org/10.1007/978-3-319-19312-0_13
8. Aßmann, U., Götz, S., Jézéquel, J.M., Morin, B., Trapp, M.: A reference architecture and roadmap for models@run.time systems. In: Bencomo, N., France, R., Cheng, B.H.C., Aßmann, U. (eds.) Models@run.time, Lecture Notes in Computer Science, vol. 8378, pp. 1–18. Springer International Publishing, Cham (2014). https://doi.org/10.1007/978-3-319-08915-7_1
9. Aufrère, R., Chapuis, R., Chausse, F.: A model-driven approach for real-time road recognition. Mach. Vis. Appl. **13**(2), 95–107 (2001). https://doi.org/10.1007/PL00013275
10. Augusto, J., Giménez-Manuel, J., Quinde, M., Oguego, C., Ali, M., James-Reynolds, C.: A smart environments architecture (search). Appl. Artif. Intell. **34**(2), 155–186 (2020)
11. Augusto, J.C., Quinde, M.J., Oguego, C.L., Giménez Manuel, J.: Context-aware systems architecture (casa). Cybernet. Syst. **53**, 1–27 (2021)
12. Bass, L., Clements, P., Kazman, R.: Software architecture in practice, 2nd edn. Addison-Wesley, Reading, MA (2003)
13. Bemthuis, R., Iacob, M.E., Havinga, P.: A design of the resilient enterprise: a reference architecture for emergent behaviors control. Sensors **20**(22), 6672 (2020)

14. Bencomo, N., Götz, S., Song, H.: Models@runtime: a guided tour of the state of the art and research challenges. Softw. Syst. Model. **18**(5), 3049–3082 (2019)
15. Bibow, P., Dalibor, M., Hopmann, C., Mainz, B., Rumpe, B., Schmalzing, D., Schmitz, M., Wortmann, A.: Model-Driven Development of a Digital Twin for Injection Molding. In: Dustdar, S., Yu, E., Salinesi, C., Rieu, D., Pant, V. (eds.) International Conference on Advanced Information Systems Engineering (CAiSE'20), Lecture Notes in Computer Science, vol. 12127, pp. 85–100. Springer International Publishing (2020)
16. Blumendorf, M., Lehmann, G., Albayrak, S.: Bridging models and systems at runtime to build adaptive user interfaces. In: 2nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems, EICS '10, pp. 9–18. ACM, USA (2010). https://doi.org/10.1145/1822018.1822022
17. Brambilla, M., Cabot, J., Wimmer, M.: Model-driven software engineering in practice: Second edition. Synthesis Lectures on Software Engineering **3**(1), 1–207 (2017). https://doi.org/10.2200/S00751ED2V01Y201701SWE004
18. Brockhoff, T., Heithoff, M., Koren, I., Michael, J., Pfeiffer, J., Rumpe, B., Uysal, M.S., van der Aalst, W.M.P., Wortmann, A.: Process Prediction with Digital Twins. In: Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 182–187. ACM/IEEE (2021)
19. Bucaioni, A., Di Salle, A., Iovino, L., Malavolta, I., Pelliccione, P.: Reference architectures modelling and compliance checking. Softw. Syst. Model. (2022). https://doi.org/10.1007/s10270-022-01022-z
20. Butting, A., Kirchhof, J., Kleiss, A., Michael, J., Orlov, R., Rumpe, B.: Model-Driven IoT App Stores: Deploying Customizable Software Products to Heterogeneous Devices. In: 21th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 22). ACM (2022)
21. Cavoukian, A.: Privacy by design: The 7 foundational principles (2009). Inf. Privacy Commissioner of Ontario, Toronto, ON, Canada
22. Chamari, L., Pauwels, P., Petrova, E.: Reference architecture for smart buildings (2023)
23. Chen, L., Hoey, J., Nugent, C.D., Cook, D.J., Yu, Z.: Sensor-based activity recognition. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews) **42**(6), 790–808 (2012). https://doi.org/10.1109/TSMCC.2012.2198883
24. Czerniak, J.N., Schierhorst, N., Brandl, C., Mertens, A., Nitsch, V.: Smart Digital Assistance Devices for the Support of Machine Operation Processes at Future Production Workplaces. In: Ahram, T.Z., Falcão, C. (eds.) Advances in usability, user experience, wearable and assistive technology, *Advances in Intelligent Systems and Computing*, vol. 1217, pp. 491–497. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51828-8_64
25. Dalibor, M., Heithoff, M., Michael, J., Netz, L., Pfeiffer, J., Rumpe, B., Varga, S., Wortmann, A.: Generating customized low-code development platforms for digital twins. J. Comput. Lang. (COLA) (2022). https://doi.org/10.1016/j.cola.2022.101117
26. Dalibor, M., Michael, J., Rumpe, B., Varga, S., Wortmann, A.: Towards a Model-Driven Architecture for Interactive Digital Twin Cockpits. In: Dobbie, G., Frank, U., Kappel, G., Liddle, S.W., Mayr, H.C. (eds.) Conceptual Modeling, pp. 377–387. Springer International Publishing, Berlin (2020)
27. Dey, A.K., Abowd, G.D.: Towards a better understanding of context and context awareness. Tech. rep. (1999)
28. Di Ruscio, D., Kolovos, D., de Lara, J., Pierantonio, A., Tisi, M., Wimmer, M.: Low-code development and model-driven engineering: Two sides of the same coin? Softw. Syst. Model. (2022). https://doi.org/10.1007/s10270-021-00970-2
29. Elmurabet, A., Abtoy, A., Touhafi, A., Tahiri, A.: Ambient assisted living system's models and architectures: a survey of the state of the art. J. King Saud. Univ. Comput. Inform. Sci. **32**(1), 1–10 (2020). https://doi.org/10.1016/j.jksuci.2018.04.009
30. El murabet, A., Anouar, A., Touhafi, A., Tahiri, A.: Towards an SOA architectural model for AAL-Paas design and implimentation challenges. Int. J. Adv. Comput. Sci. Appl. (2017). https://doi.org/10.14569/IJACSA.2017.080708
31. Ernst, J.M., Ebrecht, L., Schmerwitz, S.: Virtual cockpit instruments displayed on head-worn displays - capabilities for future cockpit design. In: IEEE/AIAA 38th Digital Avionics Systems Conference (DASC'19), pp. 1–10 (2019). https://doi.org/10.1109/DASC43569.2019.9081733
32. Galster, M., Avgeriou, P.: Empirically-grounded reference architectures: A proposal. In: Proceedings of the Joint ACM SIGSOFT Conference – QoSA and ACM SIGSOFT Symposium – ISARCS on Quality of Software Architectures – QoSA and Architecting Critical Systems – ISARCS, QoSA-ISARCS '11, p. 153-158. ACM, USA (2011). https://doi.org/10.1145/2000259.2000285
33. Garcés Rodríguez, L.M., Ampatzoglou, A., Avgeriou, P., Nakagawa, E.Y.: A comparative analysis of reference architectures for healthcare in the ambient assisted living domain. In: 2015 IEEE 28th Int. Symposium on Computer-Based Medical Systems, pp. 270–275. IEEE (2015)
34. Garcés Rodríguez, L.M., Zanin Vicente, I., Nakagawa, E.Y.: Software Architecture for Health Care Supportive Home Systems to Assist Patients with Diabetes Mellitus. In: 2019 IEEE 32nd Int. Symposium on Computer-Based Medical Systems (CBMS), pp. 249–252 (2019). https://doi.org/10.1109/CBMS.2019.00060
35. Garcés, L., Martínez-Fernández, S., Oliveira, L., Valle, P., Ayala, C., Franch, X., Nakagawa, E.Y.: Three decades of software reference architectures: A systematic mapping study. Journal of Systems and Software **179**, 111,004 (2021). https://doi.org/10.1016/j.jss.2021.111004
36. Garlan, D., Schmerl, B.: Using architectural models at runtime: Research challenges. In: Oquendo, F., Warboys, B.C., Morrison, R. (eds.) Software Architecture, pp. 200–205. Springer, Berlin Heidelberg, Berlin, Heidelberg (2004)
37. Gerasimov, A., Michael, J., Netz, L., Rumpe, B., Varga, S.: Continuous Transition from Model-Driven Prototype to Full-Size Real-World Enterprise Information Systems. In: Anderson, B., Thatcher, J., Meservy, R. (eds.) 25th Americas Conference on Information Systems (AMCIS 2020), AISeL, pp. 1–10. AIS (2020)
38. Golan, M., Cohen, Y., Singer, G.: A framework for operator - workstation interaction in industry 40. Int. J. Prod. Res. **58**(8), 2421–2432 (2020). https://doi.org/10.1080/00207543.2019.1639842
39. Grua, E.M., De Sanctis, M., Lago, P.: A reference architecture for personalized and self-adaptive e-health apps. In: European Conference on Software Architecture, pp. 195–209. Springer (2020)
40. Grundy, J., Khalajzadeh, H., McIntosh, J., Kanij, T., Mueller, I.: HumaniSE: approaches to achieve more human-centric software engineering. In: Evaluation of Novel Approaches to Software Engineering, pp. 444–468. Springer (2021)
41. Grundy, J., Khalajzadeh, H., McIntosh, J., Kanij, T., Mueller, I.: Humanise: Approaches to achieve more human-centric software engineering. In: Ali, R., Kaindl, H., Maciaszek, L.A. (eds.) Evaluation of Novel Approaches to Software Engineering, pp. 444–468. Springer International Publishing, Cham (2021)
42. Haber, A., Ringert, J.O., Rumpe, B.: MontiArc - Architectural Modeling of Interactive Distributed and Cyber-Physical Systems. Technical Report AIB-2012-03, RWTH Aachen University (2012)
43. Hersh, M.A., Johnson, M.A.: On modelling assistive technology systems - part i: Modelling framework. Technol. Disabil. **20**(3), 193–215 (2008). https://doi.org/10.3233/TAD-2008-20303

44. Hietala, H., Ikonen, V., Korhonen, I., Lahteenmaki, K., Maksimainen, A., Pakarinen, V., Parkka, J., Saranummi, N.: Feelgood-ecosystem of phr based products and services. Research report VTT-R-07000–09, VTT Technical Research Centre of Finland., Tampere, Finland (2009)
45. Hölldobler, K., Michael, J., Ringert, J.O., Rumpe, B., Wortmann, A.: Innovations in model-based software and systems engineering. J. Object Technol. **18**(1), 1–60 (2019). https://doi.org/10.5381/jot.2019.18.1.r1
46. Hu, P., Indulska, J., Robinson, R.: An Autonomic Context Management System for Pervasive Computing. In: 6th Int. Conf. on Pervasive Computing and Communications (PerCom'08), pp. 213–223. IEEE (2008). https://doi.org/10.1109/PERCOM.2008.56
47. Jim, A., Shim, H., Wang, J., Wijaya, L., Xu, R., Khalajzadeh, H., Grundy, J., Kanij, T.: Improving the Modelling of Human-centric Aspects of Software Systems: A Case Study of Modelling End User Age in Wirefame Designs. In: 16th Int. Conf. on Evaluation of Novel Approaches to Software Engineering (ENASE 2021), pp. 68–79. SCITEPRESS (2021). https://doi.org/10.5220/0010403000680079
48. Jin, Z., Cui, S., Guo, S., Gotz, D., Sun, J., Cao, N.: CarePre: an intelligent clinical decision assistance system. ACM Trans. Comput. Healthcare (2020). https://doi.org/10.1145/3344258
49. Kirchhof, J.C., Malcher, L., Michael, J., Rumpe, B., Wortmann, A.: Web-based tracing for model-driven applications. In: 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 374–381. IEEE (2022)
50. Kirchhof, J.C., Michael, J., Rumpe, B., Varga, S., Wortmann, A.: Model-driven digital twin construction: synthesizing the integration of cyber-physical systems with their information systems. In: 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, pp. 90–101. ACM (2020)
51. Kodama, H.: The ECHONET Lite specifications and the work of the ECHONET consortium. New Breeze-Q. ITU Assoc. Jpn. **27**(2), 4–7 (2015)
52. Konersmann, M., Michael, J., Rumpe, B.: Towards reference models with conformance relations for structure. In: Strecker, S., Jung, J. (Hrsg.): Informing Possible Future Worlds. Logos Verlag Berlin (2024)
53. Kostavelis, I., Giakoumis, D., Malasiotis, S., Tzovaras, D.: Ramcip: Towards a robotic assistant to support elderly with mild cognitive impairments at home. In: Serino, S., Matic, A., Giakoumis, D., Lopez, G., Cipresso, P. (eds.) Pervasive Computing Paradigms for Mental Health, pp. 186–195. Springer International Publishing, Cham (2016)
54. Leusmann, P., Möllering, C., Klack, L., Kasugai, K., Rumpe, B., Ziefle, M.: Your Floor Knows Where You Are: Sensing and Acquisition of Movement Data. In: A. Zaslavsky, P.K. Chrysanthis, D.L. Lee, D. Chakraborty, V. Kalogeraki, M.F. Mokbel, C.Y. Chow (eds.) 12th IEEE Int. Conf. on Mobile Data Management (Volume 2), pp. 61–66. IEEE (2011)
55. Lewis, G., Novakouski, M., Sánchez, E.: A reference architecture for group-context-aware mobile applications. In: International Conference on Mobile Computing, Applications, and Services, pp. 44–63. Springer (2012)
56. Liu, J.W., Wang, B., Liao, H., Huang, C., Shih, C., Kuo, T., Pang, A.: Reference architecture of intelligent appliances for the elderly. In: 18th International Conference on Systems Engineering (ICSEng'05), pp. 447–455. IEEE (2005)
57. Lunardi, G.M., Al Machot, F., Shekhovtsov, V.A., Maran, V., Machado, G.M., Machado, A., Mayr, H.C., de Oliveira, J.P.M.: Iot-based human action prediction and support. Internet Things **3**, 52–68 (2018)
58. Machot, F.A., Mayr, H.C., Ranasinghe, S.: A windowing approach for activity recognition in sensor data streams. In: Eighth International Conference on Ubiquitous and Future Networks, ICUFN 2016, Vienna, Austria, July 5-8, 2016, pp. 951–953. IEEE (2016). https://doi.org/10.1109/ICUFN.2016.7536937
59. Mannhardt, F., Petersen, S.A., Oliveira, M.F.: Privacy challenges for process mining in human-centered industrial environments. In: 14th International Conference on Intelligent Environments (IE), pp. 64–71 (2018). https://doi.org/10.1109/IE.2018.00017
60. Martínez-Fernández, S., Ayala, C.P., Franch, X., Marques, H.M., Ameller, D.: Towards guidelines for building a business case and gathering evidence of software reference architectures in industry. J. Softw. Eng. Res. Dev. **2**(1), 7 (2014). https://doi.org/10.1186/s40411-014-0007-5
61. Martínez-Fernández, S., Ayala, C.P., Franch, X., Martins Marques, H.: Benefits and drawbacks of reference architectures. In: Drira, K. (ed.) Software Architecture, pp. 307–310. Springer, Berlin Heidelberg, Berlin, Heidelberg (2013)
62. Mayr, H.C., Al Machot, F., Michael, J., Morak, G., Ranasinghe, S., Shekhovtsov, V., Steinberger, C.: HCM-L: Domain-specific modeling for active and assisted living. In: Karagiannis, D., Mayr, H.C., Mylopoulos, J. (eds.) Domain-specific conceptual modeling, pp. 527–552. Springer, Berlin (2016)
63. Mayr, H.C., Michael, J., Ranasinghe, S., Shekhovtsov, V.A., Steinberger, C.: Model centered architecture, pp. 85–104. Springer International Publishing (2017)
64. Mayr, H.C., Michael, J., Shekhovtsov, V.A., Ranasinghe, S., Steinberger, C.: A Model centered perspective on software-intensive systems. In: Enterprise Modeling and Information Systems Architectures (EMISA'18), *CEUR Workshop Proceedings*, vol. 2097, pp. 58–64. CEUR-WS.org (2018)
65. Meliones, A., Maidonis, S.: DALÍ: A digital assistant for the elderly and visually impaired using alexa speech interaction and TV display. In: 13th ACM Int. Conf. on PErvasive Technologies Related to Assistive Env., PETRA '20. ACM (2020)
66. Michael, J., Koschmider, A., Mannhardt, F., Baracaldo, N., Rumpe, B.: User-Centered and Privacy-Driven Process Mining System Design for IoT. In: C. Cappiello, M. Ruiz (eds.) Proceedings of CAiSE Forum 2019: Information Systems Engineering in Responsible Information Systems, pp. 194–206. Springer (2019)
67. Michael, J., Mayr, H.C.: Conceptual modeling for ambient assistance. In: Conceptual Modeling - ER 2013, *LNCS*, vol. 8217, pp. 403–413. Springer (2013)
68. Michael, J., Netz, L., Rumpe, B., Varga, S.: Towards privacy-preserving IoT systems using model driven engineering. In: N. Ferry, A. Cicchetti, F. Ciccozzi, A. Solberg, M. Wimmer, A. Wortmann (eds.) Proceedings of MODELS 2019. Workshop MDE4IoT, pp. 595–614. CEUR Workshop Proceedings (2019)
69. Michael, J., Rumpe, B.: Software Languages for Smart Assistance. SSRN (2023). https://doi.org/10.2139/ssrn.4423849
70. Michael, J., Rumpe, B., Zimmermann, L.T.: Goal modeling and MDSE for behavior assistance. In: Int. Conf. on Model Driven Engineering Languages and Systems Companion (MODELS-C), pp. 370–379. ACM/IEEE (2021)
71. Michael, J., Steinberger, C.: Context modeling for active assistance. In: C. Cabanillas, S. España, S. Farshidi (eds.) Proc. of the ER Forum 2017 and the ER 2017 Demo Track co-located with the 36th Int. Conference on Conceptual Modelling (ER 2017), pp. 221–234 (2017)
72. Michael, J., Steinberger, C., Shekhovtsov, V.A., Al Machot, F., Ranasinghe, S., Morak, G.: The HBMS story - past and future of an active assistance approach. Enterp. Modell. Inform. Syst. Archit. Int. J. Concept. Model. **13**, 345–370 (2018)
73. Mikalsen, M., Hanke, S., Fuxreiter, T., Walderhaug, S., Wienhofen, L.: Interoperability services in the mpower ambient assisted living platform. In: Medical Informatics in a United and Healthy Europe, pp. 366–370. IOS Press (2009)

74. Moriya, K., Nakagawa, E., Fujimoto, M., Suwa, H., Arakawa, Y., Kimura, A., Miki, S., Yasumoto, K.: Daily living activity recognition with ECHONET Lite appliances and motion sensors. In: 2017 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops), pp. 437–442. IEEE (2017)
75. Nakagawa, E.Y., Guessi, M., Maldonado, J.C., Feitosa, D., Oquendo, F.: Consolidating a process for the design, representation, and evaluation of reference architectures. In: 2014 IEEE/IFIP Conference on Software Architecture, pp. 143–152. IEEE (2014)
76. Nakagawa, E.Y., Oquendo, F., Becker, M.: RAModel: a reference model for reference architectures. In: 2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture, pp. 297–301. IEEE (2012)
77. Nee, A., Ong, S.: Virtual and augmented reality applications in manufacturing. IFAC Proceedings Volumes **46**(9), 15–26 (2013). https://doi.org/10.3182/20130619-3-RU-3018.00637. 7th IFAC Conference on Manufacturing Modelling, Management, and Control
78. Oestreich, H., da Silva Bröker, Y., Wrede, S.: An adaptive workflow architecturefor digital assistance systems. In: The 14th PErvasive Technologies Related to Assistive Environments Conference, pp. 177–184 (2021)
79. de Oliveira Camargo, M.P., dos Santos Pereira, G., Almeida, D., Bento, L.A., Dorante, W.F., Affonso, F.J.: Ra4self-cps: a reference architecture for self-adaptive cyber-physical systems. IEEE Lat. Am. Trans. **22**(2), 113–125 (2024)
80. Palyart, M., Lugato, D., Ober, I., Bruel, J.M.: Improving scalability and maintenance of software for high-performance scientific computing by combining mde and frameworks. In: Whittle, J., Clark, T., Kühne, T. (eds.) Model Driven Engineering Languages and Systems, pp. 213–227. Springer, Berlin Heidelberg, Berlin, Heidelberg (2011)
81. Perera, C., Zaslavsky, A., Christen, P., Georgakopoulos, D.: Context aware computing for the internet of things: a survey. IEEE Commun. Surv. Tutor. **16**(1), 414–454 (2014). https://doi.org/10.1109/SURV.2013.042313.00197
82. Pham, V.C., Xin, T., Sioutis, M., Lim, Y., Tan, Y.: Toward cooperation of personal health devices and smart appliances in japan smart homes. In: 2021 IEEE 3rd Global Conference on Life Sciences and Technologies (LifeTech), pp. 386–389. IEEE (2021)
83. Ringert, J.O., Rumpe, B., Wortmann, A.: MontiArcAutomaton: modeling architecture and behavior of robotic systems. In: Conference on Robotics and Automation (ICRA'13), pp. 10–12. IEEE (2013)
84. Roda, C., Navarro, E., Zdun, U., López-Jaquero, V., Simhandl, G.: Past and future of software architectures for context-aware systems: a systematic mapping study. J. Syst. Softw. **146**, 310–355 (2018). https://doi.org/10.1016/j.jss.2018.09.074
85. Rodriguez, L.M.G.: A reference architecture for healthcare supportive home systems from a systems-of-systems perspective. Ph.D. thesis, Université de Bretagne Sud; Universidade de São Paulo (Brésil) (2018)
86. Rüther, S., Hermann, T., Mracek, M., Kopp, S., Steil, J.: An assistance system for guiding workers in central sterilization supply departments. In: 6th Int. Conf. on PErvasive Technologies Related to Assistive Env., PETRA '13. ACM (2013)
87. Samin, H., Bencomo, N., Sawyer, P.: Decision-making under uncertainty: be aware of your priorities. Softw. Syst. Model. **21**, 2213–2242 (2022). https://doi.org/10.1007/s10270-021-00956-0
88. Santos, D.F., Gorgônio, K.C., Perkusich, A., Almeida, H.O.: A standard-based and context-aware architecture for personal healthcare smart gateways. J. Med. Syst. **40**, 1–14 (2016)
89. Santos, J.F.M., Guessi, M., Galster, M., Feitosa, D., Nakagawa, E.Y.: A checklist for evaluation of reference architectures of embedded systems (s)
90. Shekhovtsov, V.A., Ranasinghe, S., Mayr, H.C., Michael, J.: Domain specific models as system links. In: International Conference on Conceptual Modeling, pp. 330–340. Springer (2018)
91. Splettstößer, A.K., Ellwein, C., Wortmann, A.: Self-adaptive digital twin reference architecture to improve process quality. Procedia CIRP **119**, 867–872 (2023)
92. Steinberger, C., Michael, J.: Using semantic markup to boost context awareness for assistive systems. In: Smart Assisted Living: Toward An Open Smart-Home Infrastructure, Computer Communications and Networks, pp. 227–246. Springer (2020)
93. Szvetits, M., Zdun, U.: Systematic literature review of the objectives, techniques, kinds, and architectures of models at runtime. Softw. Syst. Model. **15**(1), 31–69 (2016). https://doi.org/10.1007/s10270-013-0394-9
94. Tazari, M.R., Furfari, F., Ramos, J.P.L., Ferro, E.: The persona service platform for aal spaces. In: Handbook of Ambient Intelligence and Smart Environments, pp. 1171–1199. Springer, Berlin (2010)
95. Uhlmann, E., Franke, D., Hohwieler, E.: Smart maintenance - dynamic model-based instructions for service operations. Procedia CIRP **81**, 1417–1422 (2019). https://doi.org/10.1016/j.procir.2019.04.327. (**52nd CIRP Conference on Manufacturing Systems (CMS)**)
96. Ullrich, C.: Rules for adaptive learning and assistance on the shop floor. In: Int. Conf. on Cognition and Exploratory Learning in the Digital Age (CELDA), pp. 261–268 (2016)
97. van Riemsdijk, M.B., Dastani, M., Winikoff, M.: Goals in agent systems: a unifying framework. In: 7th Int. Joint Conf. on Autonomous Agents and Multiagent Systems (AAMAS '08), p. 713-720 (2008)
98. Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K.: Model-Driven Software Development: Technology, Engineering, Management. Wiley Software Patterns Series. Wiley (2013)
99. Wagner, M., Zöbel, D., Meroth, A.: Model-driven development of soa-based driver assistance systems. ACM SIGBED Rev. **10**(1), 37–42 (2013). https://doi.org/10.1145/2492385.2492392
100. Wartena, F., Muskens, J., Schmitt, L., Petković, M.: Continua: the reference architecture of a personal telehealth ecosystem. In: The 12th IEEE International Conference on e-Health Networking, Applications and Services, pp. 1–6. IEEE (2010)
101. Wolf, P., Schmidt, A., Otte, J.P., Klein, M., Rollwage, S., König-Ries, B., Dettborn, T., Gabdulkhakova, A.: openaal-the open source middleware for ambient-assisted living (aal). In: AALIANCE conference, Malaga, Spain, pp. 1–5 (2010)
102. Zavala, E., Franch, X., Marco, J., Berger, C.: Hafloop: an architecture for supporting highly adaptive feedback loops in self-adaptive systems. Futur. Gener. Comput. Syst. **105**, 607–630 (2020)
103. Zhang, Z., Conly, C., Athitsos, V.: A survey on vision-based fall detection. In: Proceedings of the 8th ACM International Conference on PErvasive Technologies Related to Assistive Environments, PETRA '15. ACM, New York, NY, USA (2015). https://doi.org/10.1145/2769493.2769540

Judith Michael is PostDoc and team leader at the Software Engineering Chair of RWTH Aachen University, Germany, and the speaker of the modeling community (QFAM) within the German Informatics Society (GI). Her research focuses on model-driven software engineering, the engineering of digital twins, and software language engineering for domains such as production, ambient assisted living, controlling and finances, smart homes, health, or IoT. Her Ph.D. thesis at Alpen-Adria-Universität Klagenfurt was about cognitive modeling for assistive systems. For more information, please visit https://www.se-rwth.de/staff/Judith.Michael/.

Volodymyr A. Shekhovtsov is a PostDoc researcher at the Medical University of Innsbruck, Austria, and a lecturer at the University of Klagenfurt, Austria. His research focuses on IT support for biobanks, data and metadata quality in healthcare, model-driven engineering of complex systems, domain-specific languages, assistive systems and behavior support. His PhD thesis at National Technical University, Kharkiv, Ukraine was devoted to model, methods, and information technologies in distributed production systems.