IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 28, 2020, accepted July 8, 2020, date of publication July 29, 2020, date of current version August 13, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3012824

RoSA: A Framework for Modeling Self-Awareness in Cyber-Physical Systems

MAXIMILIAN GÖTZINGER[1,2], (Member, IEEE), DÁVID JUHÁSZ[2,5], NIMA TAHERINEJAD[2], (Member, IEEE), EDWIN WILLEGGER[2], BENEDIKT TUTZER[2], PASI LILJEBERG[1], (Member, IEEE), AXEL JANTSCH[2], (Senior Member, IEEE), AND AMIR M. RAHMANI[3,4], (Senior Member, IEEE)

[1]Department of Future Technologies, University of Turku, 20014 Turku, Finland
[2]Institute of Computer Technology, TU Wien, 1040 Vienna, Austria
[3]Department of Computer Science, University of California at Irvine, Irvine, CA 92697, USA
[4]School of Nursing, University of California at Irvine, Irvine, CA 92697, USA
[5]Imsys AB, 194 61 Stockholm, Sweden

Corresponding authors: Maximilian Götzinger (maxgot@utu.fi) and Dávid Juhász (david.juhasz@tuwien.ac.at)

*The first two authors contributed equally to the paper.

This work was supported in part by Federal Ministry Republic of Austria for Climate Action, Environment, Energy, Mobility, Innovation and Technology (BMVIT)/Austrian Research Promotion Agency (FFG) under the program Production of the Future in the project SAVE under Grant FFG 864883; in part by the European Union's Horizon 2020 Framework Programme for Research and Innovation under Grant 674875 (oCPS Marie Curie Network); and in part by the Tekniikan Edistämissäätiö (Finnish Foundation for Technology Promotion).

ABSTRACT The role of smart and autonomous systems is becoming vital in many areas of industry and society. Expectations from such systems continuously rise and become more ambitious: long lifetime, high reliability, high performance, energy efficiency, and adaptability, particularly in the presence of changing environments. Computational self-awareness promises a comprehensive assessment of the system state for sensible and well-informed actions and resource management. Computational self-awareness concepts can be used in many applications such as automated manufacturing plants, telecommunication systems, autonomous driving, traffic control, smart grids, and wearable health monitoring systems. Developing self-aware systems from scratch for each application is the most common practice currently, but this is highly redundant, inefficient, and uneconomic. Hence, we propose a framework that supports modeling and evaluation of various self-aware concepts in hierarchical agent systems, where agents are made up of self-aware functionalities. This paper presents the Research on Self-Awareness (RoSA) framework and its design principles. In addition, self-aware functionalities abstraction, data reliability, and confidence, which are currently provided by RoSA, are described. Potential use cases of RoSA are discussed. Capabilities of the proposed framework are showcased by case studies from the fields of healthcare and industrial monitoring. We believe that RoSA is capable of serving as a common framework for self-aware modeling and applications and thus helps researchers and engineers in exploring the vast design space of hierarchical agent-based systems with computational self-awareness.

INDEX TERMS Computational self-awareness, framework, agent-based, hierarchical, modeling, development, monitoring, observe-decide-act.

I. INTRODUCTION

The number of Cyber-Physical Systems (CPSs) with embedded sensors and actuators is growing exponentially [1], [2]. These systems enable a wide range of applications like automated manufacturing plants [3], telecommunication systems [4], [5], autonomous driving [6], traffic control [7], smart grids [8], and mobile health monitoring systems [9] — just to name a few examples. No matter the actual application, CPSs connect their physical environment (the real world) with the digital (i.e., cyber) space under ever-increasing expectations and requirements [10]. Some system properties that are needed for meeting application requirements are adaptivity, autonomy, reliability, robustness, long lifetime, high performance, and energy efficiency. A controlled balance among those sometimes contradictory properties is a must as well [11].

Because of these requirements, a complex interaction between a CPS and its environment is necessary. The system needs to know how its environment behaves and how its

The associate editor coordinating the review of this manuscript and approving it for publication was Mark Kok Yew Ng.

own actions may affect the environment. Besides, a CPS may have limited resources and need to consider system properties like the growing process variability, thermal limitations, and wear-out effects of System on Chip (SoC) solutions. These could lead to an unbalanced lifetime, overheating, hotspots, rapid aging, and under-utilization [12]–[14], which requires sophisticated resource management. Thus, a comprehensive assessment of the system's state and that of its environment is needed and allows prediction of future events, better planning of actions, and hence optimized operation [15].

Computational Self-Awareness (CSA) has been studied in a wide range of applications [16]–[18], and proved to be a key enabler of efficient resource management in different domains (e.g., sensor networks [19] and health monitoring systems [20]). It has also been proposed to tackle the challenges of comprehensive assessment in different CPSs [18], [21]–[24]. However, the community researching on self-awareness is fractioned, and research proceeds rather slow. To the best of our knowledge, so far, there is no satisfactory common tool to speed up research on self-awareness. We propose a software framework, *Research on Self-Awareness (RoSA)*,[1] for modeling self-awareness concepts and applications. RoSA is based on a hierarchical agent-based model and provides facilities to implement, adapt, customize, and evaluate self-aware applications. The framework itself is a three-fold software engineering exemplar [25]: it can be used in the engineering process to model applications as well as it serves as a testbed and library (i.e. infrastructure for conducting research and a set of reusable models or code, respectively). We hope that RoSA can serve as a common framework for the community to explore uncharted aspects of self-awareness and speed up development in the field.

This paper provides an overview of the framework, how it works and how it can be used. The applicability and flexibility of RoSA are demonstrated by two case studies from the fields of human health monitoring [26]–[28] and industrial machine monitoring [29]–[31]. The main contributions of the paper are:

1) we propose a framework, RoSA, which facilitates the modeling and evaluation of self-awareness concepts by means of modeling self-aware applications as *hierarchical agent systems* and modeling agents based on self-aware *functionalities*;
2) we provide an initial set of self-aware *functionalities*, namely abstraction, data reliability, and confidence, implemented in RoSA; and
3) we describe use cases for RoSA-based modeling, whose utility has been proven by our case studies.

The rest of the paper is organized as follows. Section II highlights the motivation and research challenges of this study. Section III then summarizes the state of the art in the field of CSA with interest in modeling and implementation frameworks. Section IV introduces terminology as well as the architecture and implementation of RoSA. Possible use cases of the framework are discussed in Section V. Self-aware functionalities that are currently available in RoSA are described in Section VI, whereas Section VII presents case studies, which use those functionalities and general RoSA facilities. Section VIII discusses which lessons have been learned while developing this framework, and finally, Section IX concludes the paper. We include a list of abbreviations at the end of the paper for the reader's reference.

II. MOTIVATION AND RESEARCH CHALLENGES

CSA is a hot topic, and some approaches that make CPSs intelligent exist [32], [33]. Still, the field is widely unexplored, and many aspects of self-awareness are yet to be researched.

While studying open questions of self-awareness (case studies in Section VII), we made an effort to implement experiments in a sustainable modular fashion. It was possible to separate a runtime system from application code and identify reusable components by systematizing our experimental codebases. A retrospective realization showed that much work could have been saved if it were for a framework that provided the application-agnostic parts of our custom code.

We also realized that lacking a reusable framework is not a specific issue for us but must be a general one. Despite being a hot topic, techniques and methods around self-awareness are developed at a moderate pace and lack convergence. A major obstacle that is to be overcome is the high cost (mostly development time) of implementing self-aware systems. Lacking a common framework makes each system to be developed from scratch. This results in a considerable amount of work being done redundantly, inefficiently, and uneconomically. Using a common framework would enable cooperation and synergy among researchers and practitioners from a diverse spectrum of expertise.

So we set off to make a framework based on our experience and considering the following goals:

- use a compositional application model,
- provide reusable features and facilitate customization,
- support both simulation and deployment of applications,
- have a low-footprint realization to enable the framework in resource-constrained Embedded Systems (ESs),
- make a future-proof and sustainable framework (e.g., standard and stable technology, platform independence, low overhead, open architecture).

Selecting a proper architecture and implementation fitting our goals was a fundamental question. We concluded with a hierarchical agent-based architecture, whose details are discussed in Section IV-B. As none of the available agent-based frameworks can fully cover our goals (detailed evaluation in Section III-F), we implemented RoSA as a new framework.

Identifying and implementing self-aware functionalities so that they can be reused in different applications is a

[1]Open-source implementation is available at `https://phabricator.ict.tuwien.ac.at/source/SoC_Rosa_repo.git`.

storehouse of challenges. We spent the most time with functionalities abstraction, data reliability, and confidence, whose reusable implementations are featured in RoSA.

III. BACKGROUND AND RELATED WORK

Our work is motivated by the ever-growing importance and relevance of self-awareness in CPSs and SoCs. In Section III-A, we give a short inside of Autonomic Computing (AC), while we throw a bridge to Self-Awareness in Section III-B. Since we propose a modeling framework for self-aware systems to facilitate collaboration in the field and go beyond the state of the art, we discuss existing self-aware architectures in Section III-C and review frameworks implementing self-aware systems in Section III-D. For the technical background of our proposed implementation, decentralized architectures are reviewed in Section III-E and available implementations of our choice of architecture, agent-based frameworks, in Section III-F.

A. AUTONOMIC COMPUTING

Smart systems require high degrees of automation and autonomy [34]. The word autonomy originates from ancient Greece and means to be self-governing, in other words, to have own laws [35]. In the context of computer systems, the concept of autonomy came up in the 1990s and was inspired by biological systems [36]. Both academia and industry started some initiatives at that time [35].

As often, very early attempts were made in the military field. Defense Advanced Research Projects Agency (DARPA) had a project in which they developed a communication and location device for soldiers [37], [38]. Soldiers could give information about the situation of themselves and their environment. Together with locating and sensing abilities of the device, relevant details on the battlefield were spread between the soldiers.

Besides, in the 1990s, the National Aeronautics and Space Administration (NASA) started projects such as Mars Path Finder and Deep Space 1. The goal of these projects was that space crafts should become more autonomous to operate, navigate, and manage deep-space probes with less intervention of humans [39]. The fact of becoming more autonomous was important because remote control of these space crafts is associated with a clearly noticeable delay and therefore was highly impractical.

The complexity and dynamic changing environments call for autonomic systems [35]. In 2001, the International Business Machines Corporation (IBM) declared that the complexity of Information Technology (IT) systems would be one of the biggest challenges for the progress of the industry in the coming decades [40]. To make computer systems autonomous and having less need for human interventions, IBM started the AC initiative and introduced five levels of maturity: basic, managed, predictive, adaptive, and autonomic [41]–[44]. The lowest level (basic) describes a system that is managed by highly skilled staff which monitor these systems and manually modify them based on the gathered information [37]. In contrast, the highest level describes fully autonomic systems (or applications) that totally manage themselves in order to fulfill high-level goals which could be given by humans [36]. In other words, an AC system manages itself according to high-level objectives given by humans [45].

Furthermore, IBM introduced in [42] the four self-* properties of AC (often referred to as "self-chop" [10], [37]):

- self-configuration (autonomous configuration, such as adjusting parameters or changing software, in order to fulfill high-level goals),
- self-healing (autonomous detection and diagnostic for discovering problems and trying to fix them autonomously),
- self-optimization (autonomous resource usage optimization), and
- self-protection (autonomous protection against malicious attacks and unintentional misapplication by the system's user).

These self-* properties (in details described in [44], [46]) are the most cited ones in the AC domain, but the number of them has continuously grown; for the most prominent examples, we refer to [35], [36].

B. SELF-AWARENESS

Self-awareness, which is one of the self-* properties, was proposed originally in the IBM initiative on autonomic computing [44], [47]. Computational reflection and self-awareness are very close to each other. Computational Reflection is the ability of a system to reason about its capabilities, limitations and resources [45]. A self-aware system observes itself as well as its environment and changes its behaviour according to the observations it has made. Thus, self-awareness could also be called computational reflection [48], [49]. A self-aware computer system needs sensors to sense the internal as well as the external environment and actuators to self-adapt to the changing environment [36]. In an effort to improve flexibility and adaptivity of systems, the self-* properties are organized into a hierarchy with self-awareness and context-awareness at the base (Figure 1). In other words, a system has to be self-aware to be self-adaptive (or autonomous). A correlation between the usage of self-* properties and the quality of complex software systems has been shown in [50].

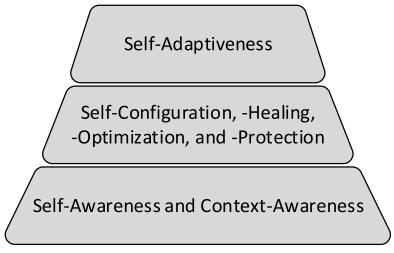


FIGURE 1. Pyramid of self-* properties.

Self-awareness has recently moved up in prominence. Initially, it was at the bottom of the self-* pyramid (Figure 1) as a supporting feature for more advanced adaptive behavior. Recently, self-awareness is used quite often to encompass all relevant self-* properties, including self-adaptiveness. The pyramid has been turned upside down because of the realization: self-awareness is not just a collection of state variables. It must include the goals of the system and properly reflect the effects of actions on itself and the environment. However, in contrast to other self-* solutions or AC, a fully self-aware system operates not only *reactively* but *proactively*. This means that such a system needs to be able to learn, make conclusions, and act accordingly [51].

For example, in the recent past, self-awareness showed to be a key enabler to tackle many challenges SoCs face such as growing process variability, thermal limitations, and wear-out effects [17], [18], [21]–[24]. CSA has been applied to both software [44] and hardware [52]. Following applications have benefited from CSA concepts (some of them under other terms such as adaptivity, autonomy, and goal-oriented systems): mobile applications [53], object tracking with smart cameras [24], [54], artificial intelligence [55], cloud computing [56], networks [57], operating systems [58], web [59], adaptive and dynamic compilation environment [60], Multi-Processor System-on-Chip (MPSoC) resource management [61], [62], (cyber-physical) SoC [52], mobile robots [63], industrial systems [64], [65], health monitoring [22] as well as single and multi-user active music environments [66].

The different aspects of self-awareness — like *self-monitoring*, *situation-awareness*, and *attention* — have been shown to be essential for efficient embedded CPSs [15], [52], [67]–[71]. Self-monitoring is the activity of sampling system properties (e.g., chip temperature [71]) as well as transforming and filtering sampled data in a system-specific way (see the self-aware functionality *abstraction* in Section VI-A). Situation-awareness assesses the observations and gives significance to data. On the other side, attention balances the competing tasks of data collection, processing, and responses under tight resource constraints by dynamically prioritizing goals and tasks. The overall system performance is monitored in a dynamically changing environment by means of self-awareness.

It has already been shown that self-awareness can help solve many problems of CPSs and SoCs. Furthermore, different aspects of self-awareness are used to make CPSs smarter [32], [33], [72]. However, the development of self-aware systems and related methods is still a difficult and tedious process. Moreover, efforts are fragmented among different communities because of the lack of a common framework to explore self-awareness and its properties. We propose a framework, RoSA, to overcome that obstacle. RoSA is based on principles and methods that have been published in literature but have not been combined before. The next few paragraphs overview various works that are related to RoSA.

C. REFERENCE ARCHITECTURES FOR SELF-AWARENESS

There exist several reference architectures which concern systems related to CSA [51]. One of them is the MAPE-K loop (an autonomic control loop coming from the AC field [42], [44])), which stands for *Monitor*, *Analyze*, *Plan*, *Execute*, and *Knowledge*. Information is collected from sensors in the monitor phase, and the gathered information is analyzed in the analyze phase. Subsequently, the plan- and execute phases are about planning and executing actions in order to fulfill goals or solve problems [51]. All these four phases share one common aspect: knowledge about the context, the execution environment and the hardware infrastructure. The MAPE-K loop is very similar to the Observe-Decide-Act (ODA) loop we have implemented (Section IV-B3).

The *Learn, Reason, Act* architecture is a model-based learning and reasoning loop (LRA-M loop) [73]. The architecture describes a self-aware computing system that is driven by its goals and its observations collected as empirical data. The collected data are used in an ongoing learning process that abstracts observations into models. The learned models provide a basis for the reasoning process, which might trigger actions affecting the system itself and possibly its environment. The LRA-M loop is a model-based formulation of the ODA loop.

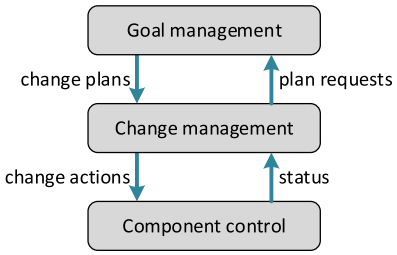


FIGURE 2. The reference architecture for self-managed systems from [74].

Another related architecture is the *Reference Architecture for Self-managed Systems* from Kramer and Magee [74]. Figure 2 shows this architecture which consists of three different layers with different tasks. The *Goal management* (the top layer of the architecture) is there for the planning. This is where plans are initiated to meet the requirements of the applications and to achieve their goals. Such plans may be required by new goals from the user or by requirements of the layer below. The *Goal management layer usually has some awareness models to be able to reflect on the layer below and address it properly [51].* This underlying layer contains *Change management*. This is where the various plans are stored, which shall be processed. The best plan for the respective current situation is selected in order to adapt the layer below. The *Change management* layer is also reflective and has typically an awareness model of the layer below; the lowest layer [51]. This layer, the layer on the bottom of the architecture, is called *Component control*. Here, the actual functionalities of the application are implemented

and accordingly adjusted by the instructions (based on various plans) from the layer above (*Change management* layer). The *Component control* layer is pre-reflective, and it sends up status reports to the layer above. If the *Component control* layer reports an inability to meet the given application goals, the *Change management* layer adapts it in a way it can achieve them in the current (environmental) situation [51]. Besides the usage of various awareness models, the hierarchical structure of this approach matches the RoSA architecture IV-B2.

The *Reference Architecture for Models@run.time Systems* is proposed in [75], and its main characteristic is that there is an explicit distinction between two systems often called managing system and managed system, where the first one manages the second one [51]. The managed system can be divided again into the (actual) managed system and its environment. The managing system often has three layers accordingly to the above-mentioned *Reference Architecture for Self-managed Systems*, where the lowest layer has an interface to the managed system. While the top layer is very similar to the previous architecture, the bottom two layers are formulated much more precisely. The bottom layer contains *configuration models* (reflecting the current state of the managed system), *plan models* (controlling the managed system), *capability models* (covering the managed system's capabilities) and *context models* (focusing on the managed system's environment). The middle layer consists of a *learner* synchronizing all models of the lowest level with the managed system, a *reasoner* making decisions based on the models of the lowest level, and an *analyzer* abstracts the information provided by models of the base layer in order to enable a hierarchical decomposition.

The ''reference architecture for self-awareness'' from Lewis *et al.* [24] describes a psychology-inspired conceptual framework of self-awareness. The architecture defines a number of different units that can be used to describe a system with self-aware and self-expressive capabilities. The components are sensor and actuator units, self-expression unit, self-awareness unit, and meta-self-awareness unit. The meta-self-awareness unit assesses the desirability of maintaining a level of awareness. The self-awareness unit consists of several subsystems for certain types of awareness:

- **stimulus awareness** is the knowledge about stimuli that act on the system and the ability to respond to them;
- **interaction awareness** is the knowledge about the interaction between the system and its environment;
- **time awareness** is the knowledge about past states and future phenomena;
- **goal awareness** is the knowledge about objectives, preferences, and constraints as well as the ability to reason about them or manipulate them;
- **meta-self-awareness** is the knowledge about possible levels of awareness and the way they are executed.

The recommended use of the reference architecture is described in a handbook [22]. A case study about implementing a service selection application in the reference architecture is available in [24].

Besides these reference architectures, a suitable modeling method, which is similar to our work, is proposed in [76]. However, that model uses a vague definition of agents as design abstraction, while RoSA provides facilities for the definition of agents based on self-aware functionalities.

D. FRAMEWORKS FOR SELF-AWARENESS

There are frameworks that focus on particular self-* properties. SAPERE [77] and ACOSO [78] are middlewares that support self-organization of autonomic nodes in distributed environments. Though they build on an agent-based model like RoSA, they are focused on self-organization (a self-awareness property that is not covered in RoSA yet) and so provide complementary features to the current set of self-aware functionalities of RoSA. The following examples provide complementary features as well. BIONETS [79] is based on similar concepts and supports self-adaptation of autonomic nodes in distributed environments. The *Collective Adaptive Systems* approach of the ALLOW Ensembles project [80] supports collaborative self-adaptation of agents within groups called *ensembles*. SEEC [61] is a framework for self-aware resource allocation based on the concept of application heartbeats, which allows monitoring and adjusting program performance. We did not base our work on any of these frameworks because (i) SAPERE and ACOSO are implemented on top of JADE, which does not fit most ESs (Section III-F); (ii) the BIONETS concepts are implemented only in simulation models, which limits its deployability in real systems; (iii) the ALLOW Ensembles approach is demonstrated by a case study in DeMOCAS [81], which is a simulation framework implemented in Java and hence has limited deployability; and (iv) the implementation of SEEC does not match the agent-based architecture, which we selected for flexibility and scalability (Section III-F).

Although these works offer more or less specific design proposals for various self-aware systems, they do not constitute a complete modeling framework.

E. DECENTRALIZED ARCHITECTURES

Decentralized architectures have already been proposed in the early days of Artificial Intelligence (AI) [51]. A decentralized system in this context consists of several agents (independent modules) which may interact with each other and work in parallel on their different tasks. According to [82], designing and building rational agents is fundamental for AI. Russell *et al.* further state that agents are rational entities that take the best possible action according to the information and capabilities they have at their disposal [82].

In [83], Wooldridge *et al.* define agents as software pieces that are autonomous (can autonomously operate without human intervention), social (can communicate with other agents or humans), reactive (can respond to changes in

TABLE 1. Multi-Agent modeling Systems.

Framework	License	Multi-Agent Simulation	Deployable Actor System	Implementation / Runtime	CPS / Embedded System	modeling Language	Native Interface
Akka [87]	Free	Possible**	Yes	Java/Scala	No	Java/Scala	JNI
CAF [88]	Free	Possible**	Yes	Native C++	Partially****	C++	C/C++
GAMA-Platform [89]	Free	Yes	No	Java	-	GUI / GAML*****	-
JADE [90]	Free	Possible**	Yes	Java	No	Java	JNI
Mobile-C [91]	Partially free*	Possible**	Yes	Native C / C++ / Ch***	Yes	Ch	Ch Binary Interface
Repast for HPC [92]	Free	Yes	No	Native C++	-	C++	-
Repast Simphony [93]	Free	Yes	No	Java	-	GUI / Groovy / Java	-
RoSA	Free	Yes	Yes	Native C++	Yes	C++	C/C++

* Proprietary dependency: Embedded Ch, free for non-commercial use on ARM-based systems.
** Not designed for simulation but might be configured for the purpose with considerable effort.
*** Ch is a scripting language with C/C++ syntax.
**** CAF poses a relatively large footprint because of its extensive non-configurable set of features.
***** GAML is the GAMA Modeling Language.

the environment), and pro-active (can take the initiative instead of just reacting). A Multi-agent System (MAS), in further consequence, is a system consisting of multiple agents that work together to fulfill one or more common goals [45].

An agent-based architecture (e.g., a MAS) implements the actor model [84], which is a programming paradigm known for scalable parallel and distributed computing. To better handle complex applications, it is usually advantageous to divide them into different tasks. Often these can be divided into different levels to cover the big picture as well as small details in particular. Accordingly, it can be helpful to have the possibility of a hierarchical structure. This is similar to the nature-inspired hierarchical system of the AC initiative from IBM [85]. Applying a *hierarchical agent-based* approach to self-aware systems has been studied in the literature [86]. An agent-based framework that facilitates self-awareness, however, has been an open issue.

F. AGENT-BASED FRAMEWORKS

Some existing self-aware frameworks are built on agents (see Section III-D). There are general agent-based frameworks, which are ignorant of the internal workings of agents. These are summarized in Table 1 and discussed in this section.

The two main use cases of the agent-based frameworks are *multi-agent simulation* and *deployable actor system*. Java-based frameworks have a high resource requirement beyond the typical capacity of ESs. The large-scale *multi-agent simulation* systems are not suitable for *ESs* for similar reasons, and they have limited capabilities for interfacing real hardware. *Deployable actor systems* with *native implementation* (Mobile-C and CAF) can support execution on ES hardware and are detailed as follows. *Mobile-C* is a small-footprint distributed actor system. However, it has a proprietary dependency and a custom native API, which limits its applicability.

CAF is an open-source distributed actor system with standard C++ implementation and with the aim of working on a wide spectrum of hardware platforms. Its extensive non-configurable feature set, however, makes it less suitable for ESs. A stripped-down version for resource-constrained systems remains a promise to date.

IV. THE RoSA FRAMEWORK

RoSA combines the agent-based actor model with self-aware properties in an ES-compatible fashion and is fully open-source. In this section, we discuss the general facilities of the RoSA framework, which are the agent-based architecture and details of its implementation. Actual *functionalities* are presented in Section VI, and the implementation of self-aware applications is showcased in Section VII by case studies.

A. TERMINOLOGY

Here, we define the following terms with the meaning we use in the context of RoSA and the rest of this paper.

1) **Agents** are design abstractions that help decompose a system into independent components. A classic definition of agents comes from the field of artificial intelligence [82]: "an agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through actuators." RoSA agents

comply with that definition. Section IV-B describes their inner workings and interactions.

2) **Data manipulation** is the processing activity that is done by any agent: observing its environment via input, maintaining its internal state, and optionally generating output to affect its environment. Individual RoSA agents may realize different ways of data manipulation, which is described in terms of *functionalities*.
3) **Functionalities** encapsulate self-awareness concepts in reusable components. They are the basic tools that can be put together to realize desired ways of data manipulation in agents. An agent is designed by a careful selection of functionalities for the required data manipulation. The self-aware functionalities that we have already implemented are elaborated in Section VI.

B. RoSA ARCHITECTURE

The architecture of the RoSA framework is outlined in this subsection, accompanied with a discussion on some design decisions.

1) SCOPE

The RoSA architecture supports modeling self-aware applications, whose relevance is motivated in Section I and Section II. The framework is intended to be a tool for modeling and evaluating novel ideas in self-aware applications. The application model (i.e., a hierarchical agent-based system with functionalities within agents) is flexible enough to incorporate variations in different aspects of design and implementation. Those aspects are mostly related to the functionalities: (i) what functionalities are there, (ii) how they are implemented and interconnected, and (iii) how applications are decomposed. The architecture provides a structured and modular way for defining self-aware applications: applications are decomposed into agents, which are defined by functionalities. Agents and functionalities are reusable components in RoSA.

2) HIERARCHICAL AGENT-BASED MODEL

An agent communicates with its environment (i.e., other agents of the application) by message passing via input and output channels. Semantics can be informally given as: the agent receives messages on its input channels; manipulates data (i.e., the received messages and its internal state), and may send messages on its output channels.

Agents are organized into a hierarchical structure (e.g., Figure 3). Agents on different levels of the hierarchy process data on different levels of abstraction: the system obtains fine- and coarse-grained knowledge according to hierarchy levels. Such a detailed representation of knowledge helps self-adaptive systems to operate more efficiently and meet their goals [32].

Connected agents are in a master-slave relation. An agent (e.g., Agent 2 in Figure 3) receives messages from its slaves (Agents 5 and 6) and sends messages to its master (Agent 1). That is, an agent acts as slave towards its only master and as master towards its potentially multiple slaves.

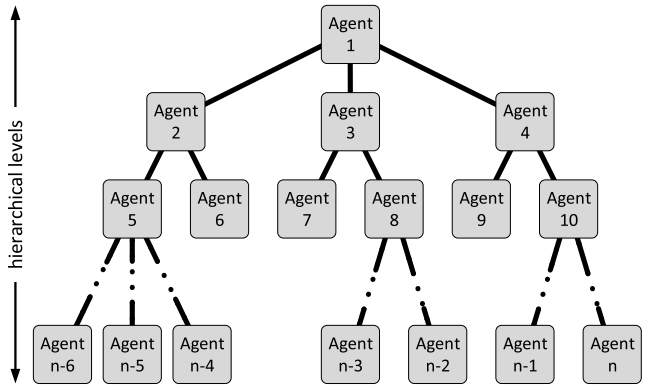


FIGURE 3. A hierarchical agent-based model.

A slave sends messages to its master regularly according to its configuration. A master may control the configuration of its slaves by sending control messages to them whenever appropriate.

A real-world application interacts with its environment via sensors and actuators, which are modeled as agents in RoSA. An agent that wraps a sensor is a data source (i.e., has no slaves) and sends sensor input to its master. Dually, an agent that wraps an actuator is a data sink (i.e., has no master). An actuator is activated (''controlled'') by slave-to-master data messages — rather than master-to-slave control messages.

3) OBSERVE-DECIDE-ACT LOOPS

AC systems consist of autonomic elements implementing a control loop [36]. Thus, self-aware applications in RoSA operate in an iterative manner implementing ODA loops [52]. ODA is our architecture of choice, however, other architectures could be chosen and implemented as well. An ODA loop (Figure 4) represents the way reactive systems operate: the system monitors the behavior of itself and/or its environment, decides about certain actions, and acts accordingly.

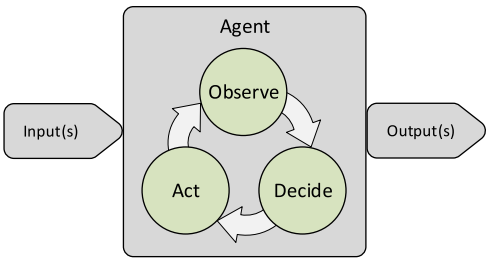


FIGURE 4. An agent implements an Observe-Decide-Act loop.

As shown in Figure 5, each RoSA agent operates in an ODA loop: receives input messages, does data manipulation, and optionally sends output messages. The composition of individual ODA loops results in a behavior that can be described as a compound ODA loop on the application level.

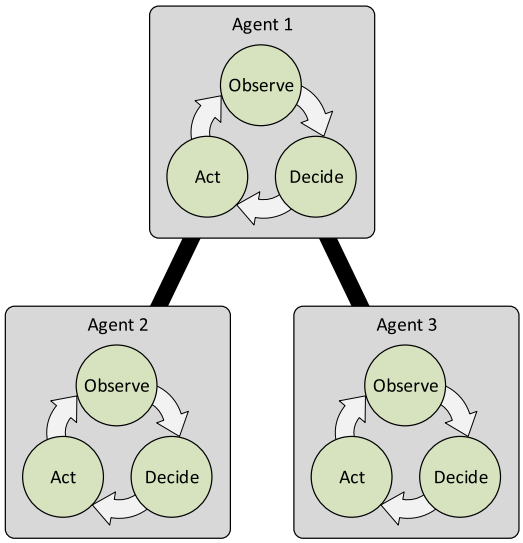


FIGURE 5. An agent system based on individual Observe-Decide-Act loops.

The RoSA architecture provides a way to implement ODA-loop-based applications that are decomposed into interacting ODA loops of lower complexity.

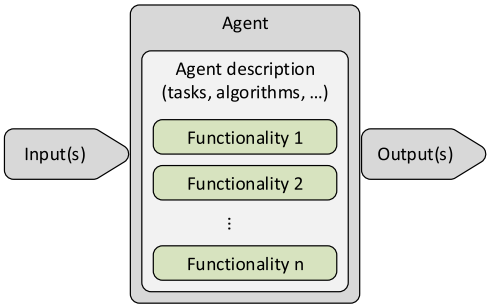


FIGURE 6. The behavior of an agent is defined by self-aware functionalities.

4) FUNCTIONALITIES

An agent is defined by functionalities (Figure 6). What functionalities an agent utilizes depends on its role in the application. RoSA provides a library of pre-defined functionalities (Section VI) and allows developers to implement new ones either based on existing ones or from scratch.

As shown earlier, RoSA agents conceptually operate in ODA loops. The functionalities that constitute an agent contribute to different characteristics of observation and decision making in the loop. For example, *abstraction* (Section VI-A) improves the outcome of observation, and *data reliability* (Section VI-B) helps decision making by providing meta-information. Our approach is inspired by the hierarchical agent-based model of Guang *et al.* [76]. While their model uses a vague definition of agents as design abstraction, RoSA agents are described as ODA loops that are based on functionalities.

C. SOFTWARE IMPLEMENTATION

We have implemented the RoSA architecture as a software framework. RoSA has a fully open implementation in standard C++ and can readily interface existing native software components. The main characteristics of the software implementation are (i) providing a high-level but safe modeling interface for application developers, (ii) allowing the same application code to be used for *simulation* and *deployment*, and (iii) realizing small-footprint software that can be deployed in resource-constrained ESs. We have done our case studies (Section VII) in simulation on a desktop computer. That is, input and output of sensor and actuator agents are fed to the system via stored files, and RoSA allows for other input-output interfaces as well. Runtime support for deploying on embedded devices requires further development to complete.

V. USE CASES OF THE FRAMEWORK

The section discusses how RoSA, the framework as a whole and its features separately, can be used in different scenarios.

A. MODELING A NEW APPLICATION IN RoSA

Modeling an application using the RoSA Architecture follows a general flow shown in Figure 7. That is,

Specify requirements: The most abstract description of an application defines input and output (sensors and actuators, respectively) and the data manipulation to be done. It can be seen as an extreme agent system with all sensors and actuators connected to the only agent that represents the entire application.

Model agent system: The monolithic application-agent is decomposed into a set of agents organized in a hierarchy. Agents enclose specific kinds of data manipulation and serve as a unit of reusability — within and between applications. Identifying agent patterns can help efficient decomposition.

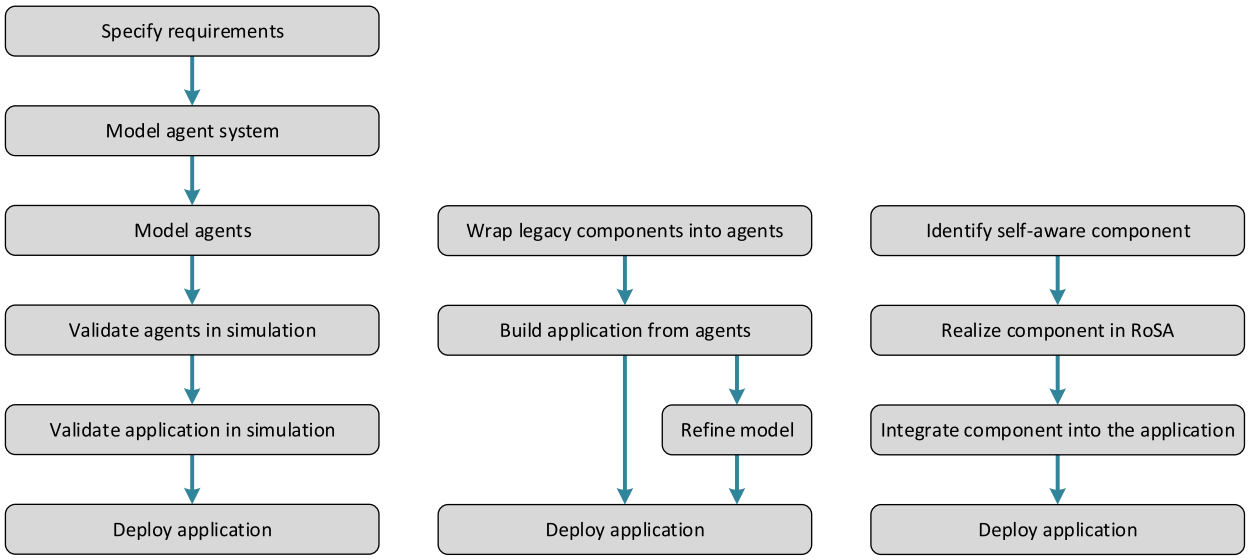
Model agents: Each agent is modeled, i.e., prescribed data manipulation is realized by available functionalities and custom code (Figure 8). Functionalities provide a level of reusability below agents. RoSA provides a set of functionalities, which is expected to grow over time.

Validate agents in simulation: *Unit testing* of agents is done by validating their input-output behavior in simulation mode: a single-agent system is evaluated with predefined input and expected output.

Validate application in simulation: Agents are put together according to the *system model*. *Integration testing* of the application is done by validating the input-output behavior of the system in simulation mode.

Deploy application: The application is deployed in an embedded device.

Though the RoSA methodology is presented as a sequential flow, the model of an application (i.e., the system model with corresponding agent models) may be refined in an iterative manner.



(a) Modeling a new application in RoSA (b) Migrating a legacy application to RoSA (c) Adding RoSA as component to a legacy application

FIGURE 7. Scenarios of using the RoSA framework.

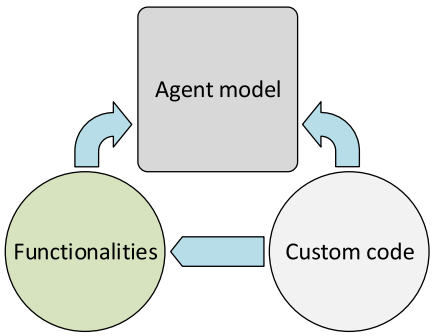


FIGURE 8. An agent is modeled based on available *functionalities* and *custom code*; reusable pieces of *custom code* are gradually promoted to *functionalities* in a generalized form.

B. MIGRATING AN APPLICATION TO RoSA

RoSA can also be used to add self-awareness to existing applications. An entire legacy application can be migrated to RoSA in a few steps shown in Figure 7(b). That is,

Wrap legacy components into agents: Each component, whose input-output behavior fits message-passing semantics, is wrapped into a RoSA agent. Legacy components may be grouped, if necessary. Existing legacy code implements data manipulation within agents.

Build application from agents: Agents are put together in an agent system according to the connections between corresponding components in the legacy system.

Refine model: The system and agent models may be refined iteratively, as in the general case.

Deploy application: The agent system is deployed as a RoSA application.

This approach turns a legacy system into a RoSA application and enables utilizing all RoSA features for further development.

C. ADDING RoSA AS A SELF-AWARE COMPONENT

It is also possible to add a RoSA agent system to an existing application as a self-aware component (Figure 7(c)). This scenario might be applied as a gradual migration path.

Identify self-aware component: The requirements are specified either as a new component of the application or based on an existing component to be replaced.

Realize component in RoSA: The component is realized as a RoSA agent system following the general RoSA methodology (Figure 7).

Integrate component into the application: The component is integrated into the existing application via input and output streams that are associated with its sensor and actuator agents, respectively.

Deploy application: The application is deployed with the RoSA system as one of its components.

This approach limits the development effort to one component of the application — in contrast to migrating the whole application. However, additional development and runtime complexity is posed by the need to integrate RoSA as a component of the existing application. Whether to take the full migration or the component approach depends on the size of the application and how much the application needs self-awareness and can benefit from RoSA.

D. USING SELF-AWARE FUNCTIONALITIES FROM RoSA

RoSA supports reusability on two levels: agents in the system model and functionalities in the agent model. The realizations

of these two levels are independent, and functionalities may be used without using agents. Should working with a RoSA agent system be uneconomic, functionalities that are defined in RoSA (Section VI) may be used in custom codes directly — without involving other parts of the RoSA framework.

E. IMPLEMENTING A GENERAL AGENT-BASED APPLICATION WITH RoSA

While the main aim of RoSA is to facilitate developing applications and concepts related to self-awareness, the applicability of the framework is not limited to that. The agent system that constitutes the base of the architecture can be used for any other application that may benefit from such an architecture (e.g., component-based systems). One can ignore self-aware functionalities and implement an agent-based application with all data processing in agents defined by custom application-specific code only.

VI. SELF-AWARE FUNCTIONALITIES

Each RoSA agent receives messages from its input channels and may send messages on its output channels. The data processing that the agent does to maintain its state based on input messages and generate output messages can be defined with full flexibility (i.e., custom application code). Nevertheless, RoSA provides predefined functionalities to be used as components when defining agents, with minimal glue code that connects them. It is also possible to mix functionalities and custom code freely within agents. The modularity enables application developers to define self-aware agents fast and efficiently by reusing existing functionalities and also customize data processing whenever needed.

The functionalities are based on self-aware properties [94], [95]. RoSA provides reference implementations of the functionalities that have been used in our case studies (Section VII): abstraction, data reliability, confidence, and history. We expect the set of self-aware properties and corresponding functionalities to grow as well as their implementation to improve — contributions from the community are welcome.

A. ABSTRACTION

Abstraction is ''an appropriate selection of the representation of the information in order to obtain compact knowledge relevant to a particular purpose'' [94]. It is a transformation of data from one domain to another. Raw input data may be abstracted into a semantic domain that the self-aware system understands [52], and the abstraction may be done at any level of a hierarchical system. It could also be done top-down instead of bottom-up [94]. An abstraction needs to be meaningful and efficient in the system's context and to have a well-defined structure.

1) ABSTRACTION FUNCTIONALITIES AVAILABLE IN ROSA

The broad definition of abstraction allows for a wide variety of approaches. RoSA currently provides the following abstraction functionalities:

1) **Lookup table** maps an input datum to a symbol (e.g., number, character, string).
2) **Overlapping lookup table** maps an input datum to potentially multiple symbols; in case the boundaries between symbols cannot be clearly defined (e.g., insufficient knowledge about the environment). Selecting one symbol in a later processing step may be a confidence-based decision (Section VI-C). In contrast, a standard lookup table maps an input value directly to one symbol.
3) **Threshold-based signal state detector** abstracts steady states from a signal waveform, that is a sequence of input values. In other words, it recognizes stable phases in a signal. These steady states of a signal are identified concerning a threshold of distance among the signal's sample values. A signal state is stored as an average value of all input samples belonging to it. A simple learning algorithm is utilized internally for state detection. Detailed discussion is available in [29], [30].
4) **Confidence-based signal state detector** also abstracts steady states from a signal waveform, that is a sequence of input values. These steady states of an input signal are identified concerning the relative distance among the signal's sample values. That is, in contrast to the *Threshold-based signal state detector*, the *Confidence-based signal state detector* makes all decisions based on a confidence assessment (Section VI-C). This assessment is not only based on a simple average, but on the most recent signal samples stored in a sliding window history. A detailed discussion of the learning algorithm behind this functionality is available in [31].
5) **System state detector** abstracts a system state from signals of an observed system. The current implementation works with stateless systems only (i.e., identifying states of a system whose output can be expressed as a function of its input).

B. DATA RELIABILITY

Data reliability is ''the extent to which a measuring procedure yields the same results on repeated trials'' [94]. The trustworthiness of data is determined by accuracy, precision, and truthfulness. The accuracy and precision are given by systematic and random error of measurement, respectively. Data can be accurate and precise but still not truthful [28], for instance, if a sensor is working outside of its operating conditions (e.g., a temperature sensor detached from the test object).

Data reliability is a piece of meta-data about the trustworthiness of the input data stream. Further actions may be taken

according to the reliability of data. The following measures may be used to assess trustworthiness:

1) **Plausibility** tells whether data is within its expected domain (i.e., range). If a variable exceeds the realistic limits of its represented quantity (e.g., human body temperature over 100°C), data might be unreliable.
2) **Consistency** tells whether data varies according to its expected variability (i.e., maximum difference between samples). If a variable changes too fast (e.g., position of a robotic arm), data might be not reliable. Checking consistency requires historical information (Section VI-D) about the input signal.
3) **Cross-validity** tells whether one piece of data correlates with other pieces as expected. If two dependent variables (e.g., two interdependent vital signs) do not follow each other, data might be not reliable.

1) DATA RELIABILITY FUNCTIONALITIES AVAILABLE IN ROSA

RoSA provides functionalities for assessing each of these three measures of trustworthiness either in a binary or in a fuzzy way (a total of 6 variants). Binary assessment makes a binary decision about reliability according to a threshold. Fuzzy assessment determines the level of data reliability in the [0, 1] range and can be configured with a custom function.

The individual assessments may be combined (e.g., considering both plausibility and consistency of a variable at the same time). RoSA provides a set of predefined methods (average and multiplication of fuzzy assessments; conjunction and disjunction for both binary and fuzzy assessments) for the combination, which may be done as custom application code as well.

C. CONFIDENCE

Confidence is "the extent to which a procedure may yield the same results on repeated trials" and has significant similarities to data reliability [94]. Confidence is a piece of meta-data about the trustworthiness of the data processing performed by a (sub-)system or function. It tells how well a calculated result corresponds to the expected output. Assessing confidence assumes error-free input — which may be assessed by data reliability (Section VI-B).

1) CONFIDENCE FUNCTIONALITIES AVAILABLE IN ROSA

RoSA defines an interface for assessing confidence, but the actual assessment logic needs to be provided as a custom function. The lack of predefined assessment functions is because no general confidence measures have been identified yet. The assessment of confidence varies much on a case-by-case basis in our experience.

Besides this interface, RoSA offers a confidence-based abstraction method, which is an *overlapping lookup table* (Section VI-A). This method is based on fuzzy membership functions [96], and Figure 9 shows an example of it. The input data is mapped to three symbols (A, B, and C) so that two symbols are associated for the overlapping ranges

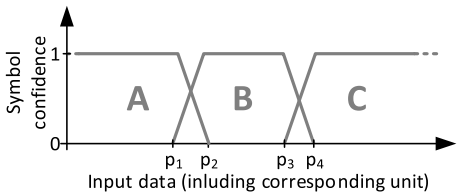


FIGURE 9. A confidence-based abstraction method to abstract data into one or more symbols with a corresponding confidence.

(i.e., (A, B) and (B, C) for $[p_1, p_2]$ and $[p_3, p_4]$, respectively). The abstracted symbols are assigned with a confidence value according to their corresponding fuzzy membership functions (i.e., *full* confidence outside of the overlapping ranges and lower confidences inside them). The membership functions can be adjusted dynamically via control feedback in the agent hierarchy whenever a higher level agent recognizes a systematic error.

Cross-validity confidence tells whether one piece of data correlates with other pieces. It is similar to cross-validity reliability in that respect. However, it calculates historical correlation information based on active monitoring, unlike the a priori expectations of cross-validity reliability. This assessment can be used to tune confidence-based abstraction in lower levels of the hierarchy.

Individual confidence assessments may be combined, similar to combining individual reliability assessments. RoSA provides predefined combination methods and the possibility of handling combination by custom application code. The reliability of the output of an agent can be assessed by combining the reliability assessment of its input and the confidence assessment of its data processing.

D. HISTORY

History is "recording and studying a series of past events connected to an entity" and enables extracting knowledge from the recorded time series [94]. Identifying trends in the past, understanding time-dependent aspects of the current state, and predicting future conditions [28] may all be supported by utilizing history.

1) HISTORY FUNCTIONALITIES AVAILABLE IN ROSA

RoSA includes limited support for history (only the features that we needed to implement other functionalities). A short description is still included because the history functionality can be used in the application code directly.

History functionality enables storing a sequence of data values. Its capacity can be configured for a balanced memory usage. History supports two strategies for redeeming memory once its capacity is reached: (i) stop strategy, when a full history does not accept further data values, and (ii) FIFO strategy, when history behaves like a sliding window. History functionality allows access to the individual stored values and also provides statistical properties (e.g., average) about the stored sequence.

VII. CASE STUDIES

Applications with different levels of self-awareness have been implemented in RoSA. We present two case studies in this section, which demonstrates how RoSA can be used for quick application development.

A. SELF-AWARE EARLY WARNING SCORE SYSTEM

The first case study is presented in detail for a smooth introduction of implementation details. The application — whose different variants are developed over the case study — is a health status assessment system.

1) BACKGROUND AND PROBLEM STATEMENT

Here, both, the calculation of the Early Warning Score (EWS) and the traditional EWS system are briefly described, before the next subsections deal with its extensions with various self-awareness properties.

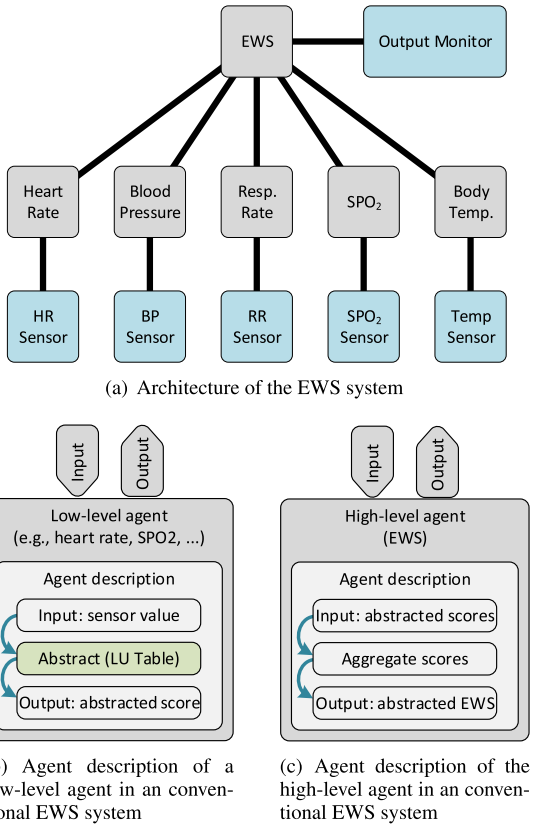
A patient's health status can be assessed based on their vital signs. Research on cardiac arrests shows that certain symptoms can be observed long before the situation turns into a case of emergency; symptoms may appear even 24 hours before actual health deterioration [97]. EWS is a standard manual tool for assessing patients' health status and predicting health deterioration. Healthcare professionals periodically monitor patients' vital signs (heart rate, respiratory rate, body temperature, blood pressure, and blood's oxygen saturation) and assess their health status by a criticality level defined as EWS [98]. For this reason, each vital value is assessed in the form of a score. A score of 0 indicates an ideal health condition of a vital sign, while score 3 corresponds to the worst. The EWS is the aggregate value of all the individual vital sign scores. The higher the score, the higher the criticality.

This manual procedure has been applied to hospitalized patients. A portable device that automates the procedure would allow high-risk patients to pursue their daily lives with a much higher chance of survival. Robustness and fault tolerance is of major importance for such a device. Autonomous monitoring of patients in a non-hospital environment needs to deal with faulty measurements: sensors can be attached incorrectly, become detached, or break down. Incorrect measurements result in incorrect EWS, which might lead to false positive or — even worse — false negative assessments.

We developed several self-aware variants of the EWS application, which are able to deal with different kinds of faults [26]–[28]. Those variants and their results are discussed in detail in the referred papers. Here we motivate their high-level design and present their implementation in RoSA.

2) THE CONVENTIONAL EWS SYSTEM

For starters, we implement an application that calculates the EWS in the conventional way (Figure 10). Five agents constitute the low level of the hierarchy, each connected to a sensor (modeled as a special agent) and assessing the corresponding vital sign. One agent in the higher level is connected to the low-level agents to make the aggregate assessment, whose result is recorded by a monitoring agent. The actual implementation is outlined in Listing 1. Even though RoSA and the applications are implemented in C++, the included listings use a C++-based pseudo-code for brevity. The sometimes verbose syntax of C++ is hidden, but the complexity of the application code is presented truly.



(a) Architecture of the EWS system

(b) Agent description of a low-level agent in an conventional EWS system

(c) Agent description of the high-level agent in an conventional EWS system

FIGURE 10. The conventional EWS system.

The implementation (Listing 1) starts with creating a RoSA *Application* (Line 1). Agents can be created and managed in the context of the Application.

For each vital sign (demonstrated by heart rate), a sensor (Line 2) and a low-level agent (Lines 4 to 21) are created. The low-level agent (Figure 10(b)) performs the EWS assessment by applying a lookup table abstraction (Section VI-A). The connection between the sensor and the low-level agent is established in a separate step (Line 23).

The high-level agent ((Figure 10(c)) aggregates vital sign assessments by summing them into a final EWS (Lines 26 to 31). Each low-level agent is connected to the high-level agent (Line 33).

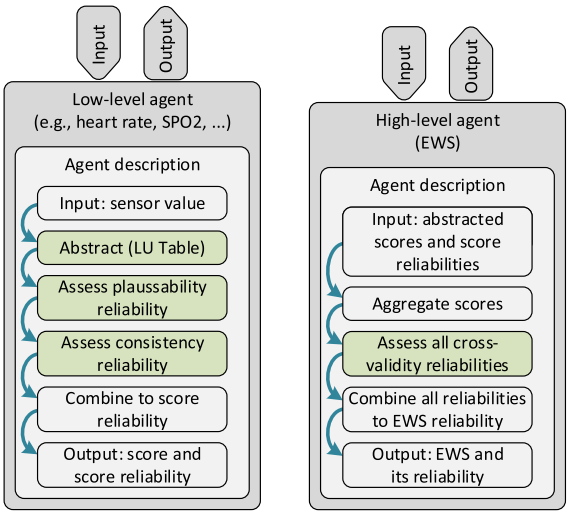
The final EWS is logged to the console by a dedicated agent (Lines 36 to 40), which is connected to the high-level EWS agent (Line 42).

3) A SELF-AWARE EWS SYSTEM WITH RELIABILITY FUNCTIONALITIES

The conventional EWS system does not tolerate faults. Thus we make the system more robust by utilizing additional

```
App = Application::create("EWS");
HRSensor = App->createSensor("HR Sensor");

HRAbstraction = RangeAbstraction(
  {
    // upper bounds are exclusive
    {{  0,  40}, Emergency},
    {{ 40,  51}, High},
    {{ 51,  60}, Low},
    {{ 60, 100}, No},
    {{100, 110}, Low},
    {{110, 129}, High},
    {{129,  40}, Emergency},
  }, Emergency);  //default

HRAgent = App->createAgent(
  fun (updated, value) HR → Optional(Score) {
    if(HR.updated)
      return HRAbstraction(HR.value);
    return None;
  });

App->connectSensor(HRAgent, HRSensor);
// repeat for all vital signs

BodyAgent = App->createAgent(
  fun {(updated, score)} Scores → EWS {
    if(all Scores.updated)
      return sum(all Scores.score);
    return None;
  });

App->connectAgents(BodyAgent, HRAgent);
// repeat for all vital signs

LoggerAgent = App->createAgent(
  fun (updated, value) Score → None {
    if(Score.updated)
       Write(Score);
  });

App->connectAgents(LoggerAgent, BodyAgent);
```

LISTING 1. RoSA implementation of a conventional EWS system.



(a) Description of a low-level EWS agent equipped with reliability assessment

(b) Description of the high-level EWS agent equipped with reliability assessment

FIGURE 11. Agent descriptions for the EWS system with reliability assessment.

```
HRAgent = App->createAgent(
  // process new sensor values
  fun (updated, value) HR → Optional(S, C) {
    if(HR.updated)
      return (HRAbstraction(HR.value),
          ↪ HRReliability(HR.value));
    return None;
  },


  // process control feedback from master
  fun (updated, values) Scores → None {
    if(Scores.updated)
      HRAbstraction.update(Scores.values);
  });
```

LISTING 2. RoSA implementation of the self-aware heart rate agent.

self-aware functionalities within the agents (Figure 11). The agent hierarchy remains unchanged (Figure 10). Setting up the RoSA application and agent hierarchy is done similarly to the conventional EWS system (Listing 1).

The low-level agents (Figure 11(a)) assess the reliability of the abstracted vital signs by checking plausibility and consistency in combination (Section VI-B). The agent implementation is adapted by specifying the output as a pair of values (i.e., abstracted value and reliability assessment) rather than a single value and by utilizing reliability functionality in data processing (Listing 2 Lines 3 to 7).

The high-level agent (Figure 11(b)) assesses cross-validity reliability ((Section VI-B)) of the vital signs and combines all reliability assessments for the final EWS. The agent implementation is adapted similarly to low-level agents (i.e., adjust input and output types and utilize reliability functionality).

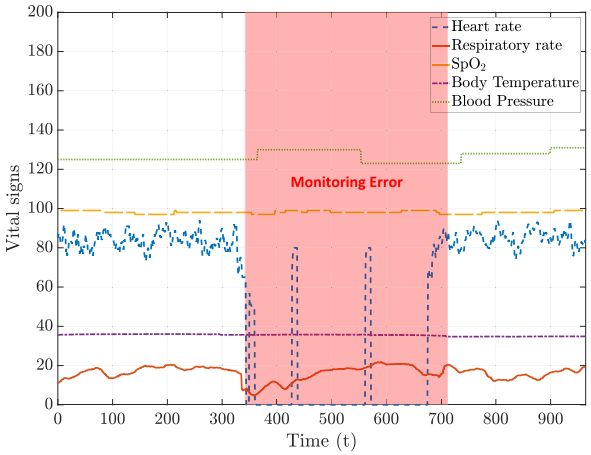
The reliability functionalities may be configured in different ways for the agents. Experiments have been performed both with binary [26] and with fuzzy [27] assessments.

Consider an experiment with the chest strap, which measures heartbeat, being loosely fastened. The measurement is not stable and provides unreliable readings during some time (e.g., 350s – 670s in Figure 12), while other sensors provide reliable data. Though the EWS is calculated according to the standard rules (i.e., results in false positives), the assessed reliability drops to 0 during measurement errors. The low reliability indicates an issue with the system's input(s).

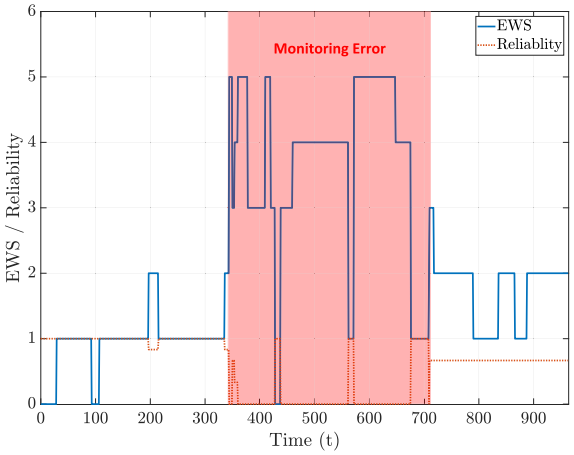
4) A SELF-AWARE EWS SYSTEM WITH CONFIDENCE FUNCTIONALITY

While the previous version calculated the EWS without any modifications, the final version adjusts the EWS in case of a low reliability [28].

The master agent (Figure 13(b)) additionally assesses cross-validity confidence of the vital signs (Section VI-C). The confidence assessment is based on personalized data and is — combined with the cross-validity reliability — used as control feedback for the low-level agents to adjust their score abstraction process. The implementation, using predefined functionalities, is still only a few lines (Listing 3).



(a) The monitored vital signs, where the heart rate signal often breaks down due to the faulty measurement



(b) The calculated EWS and its reliability

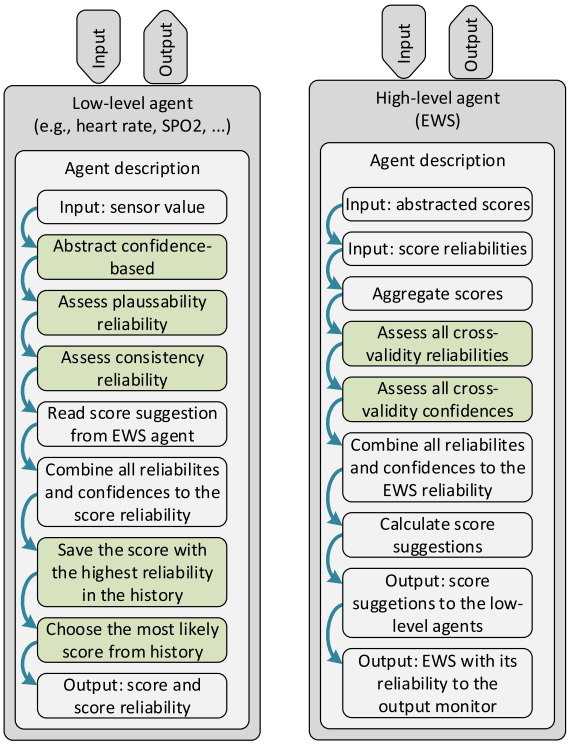
FIGURE 12. Results of an experiment in which the heartbeat sensor was not attached properly and therefore incorrect measurements were made.

```
1  BodyAgent = App->createAgent(
2    fun {(updated, score, rel)} Scores → ((EWS,
       ↪ rel), {vital-confidence}) {
3      if(all Scores.updated)
4        return (
5          // result
6          (sum(all Scores.score), EWSReliability
             ↪ (all Scores.rel))
7          // control feedback
8          {all EWSConfidence(all Scores.score)}
9        );
10     return None;
11   });
```

LISTING 3. RoSA implementation of the self-aware EWS master agent.

The agent generates its output as (i) a pair of calculated EWS and assessed reliability (Line 6) and (ii) a list of confidence feedback for each low-level agent (Line 8).

Low-level agents (Figure 13(a)) perform confidence-based abstraction (Section VI-C and Figure 14), which takes historical information into account (Section VI-D) about feedback from the high-level EWS agent. The calculated EWS is



(a) Agent description of a low-level agent in the EWS system equipped with reliability, confidence, and history

(b) Agent description of the high-level agent in the EWS system equipped with reliability, confidence, and history

FIGURE 13. Agent descriptions in the EWS system equipped with reliability, confidence, and history.

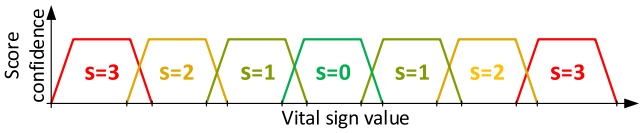
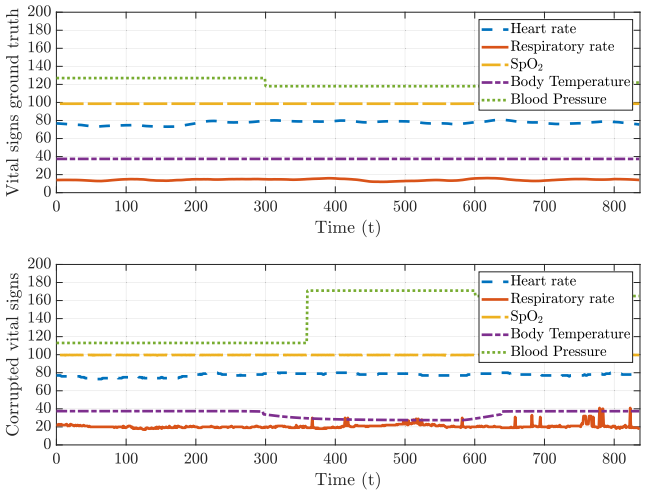


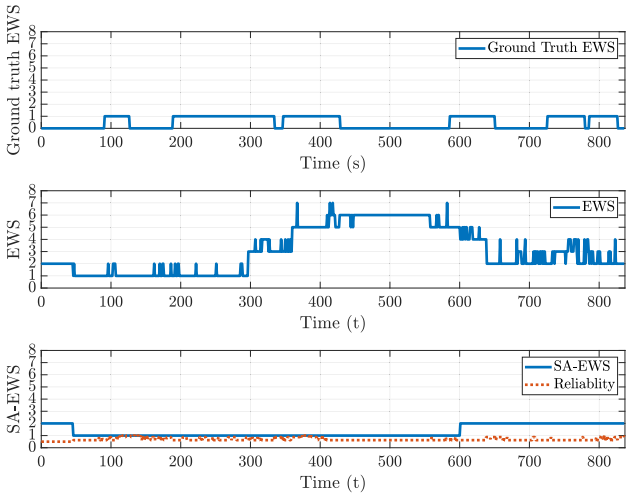
FIGURE 14. A confidence-based abstraction method to abstract a vital sign in one of four different scores (0 to 1).

adjusted in that way. The actual implementation (Listing 2) is divided into two functions: (i) one for processing input, like before, and (ii) one for processing control feedback. Sensory input is processed like before (Lines 3 to 7) except for abstraction being configured to operate based on confidence (Section VI-C). Processing control feedback (Lines 11 to 14) passes data from the high-level agent to the local confidence-based abstraction. The feedback is stored by the abstraction functionality internally with a history functionality and is utilized when processing future sensory input.

Consider an experiment when participants are monitored with both working and faulty sensors (upper and lower part of Figure 15(a), respectively). The experiment results (Figure 15(b)) show that our self-aware EWS application performs much better (even if not perfectly) than the conventional system in the presence of sensory errors. The self-aware EWS system has almost 80 times less false alarms than the conventional system in our experiments [28].



(a) The vital signs in the top are free from errors; those in the bottom were measured by faulty sensors and show significant errors



(b) The top diagram shows the correct EWS based on the actual vital signs; the two lower diagrams show EWS values calculated by the conventional and our self-aware EWS systems with faulty sensory input

FIGURE 15. Results of an experiment when participants were monitored with both working and faulty sets of sensors.

5) SUMMARY

The detailed case study followed the development of our EWS application through four versions. That process demonstrates that implementing an agent-based application with self-aware properties takes only a few steps in RoSA. Defining the agent system at the beginning was a lightweight task by using the existing agent interfaces of RoSA. Additionally, thanks to the modular design and reusable functionalities of RoSA, moving from one version to the next (i.e., including more sophisticated self-aware properties) needed only local modifications of agents and functionality configurations.

We were, of course, experimenting with different implementation alternatives during development. In the end, however, we packed the various functional components of data processing into functionalities, which are reusable modules. New applications can use those functionalities while they might also need to implement novel data processing approaches in custom application code. Those pieces of custom code, once matured, should be turned into functionalities for modularity and reusability. That is a way for sustainable development in the long run, and it is facilitated by RoSA.

B. CONTEXT-AWARE CONDITION MONITORING

This case study presents a monitoring system that assesses the working state and the health condition of another system or device; hereafter System under Observation (SuO). We limit the discussion to the modeling level (i.e., source-level implementation is ignored); the first case study provides insight into implementation.

1) BACKGROUND AND PROBLEM STATEMENT

Industry, particularly automated production plants, has an interest in reliable monitoring systems that are able to raise an alarm in case of malfunctions of the SuO [99]. Such a monitoring and warning system enables optimization of maintenance work and minimizes downtimes.

Implementing tailor-made monitoring systems for each SuO is an expensive endeavor. A reliable self-adaptive monitoring system can reduce cost and time. We present such a system, which is able to assess the health status of any SuO without detailed a priori knowledge but by observing its input and output. The system assumes that the SuO meets two requirements: (i) the SuO works as a bijective function between its input and output, and (ii) the SuO operates in steady states. Requirement (i) allows the monitoring system to uniquely identify input-output pairs of normal operation. Dissociation of input and output signals is then considered a symptom of fault. Requirement (ii) is a consequence of the fact that the monitoring system discards unstable and transient signals.

The monitoring system adapts to any SuO based on contextual information only (see context-awareness). We have performed experiments with two variants of the system: (i) Context-aware Health Monitoring (CAH) [29], [30] applies a threshold-based decision-making process and (ii) Confidence-based Context-Aware condition Monitoring (CCAM) [31] makes decisions based on confidence.

Compared to similar monitoring solutions (e.g., deep learning and data mining), our system has a considerably smaller runtime footprint and can be applied to resource-constrained applications.

2) MODELING THE MONITORING SYSTEM

The application has a hierarchical structure (Figure 16). The low-level agents are connected to the sensors and can perform pre-processing of sensory input if necessary as well as incorporate a signal state detector (Section VI-A).

The detected signal states are combined into a system state by a system state detector (Section VI-A) in the high-level agent. Its output (i.e., system state and health condition) is processed (e.g., logging or trigging a warning in case of malfunction) by a dedicated agent.

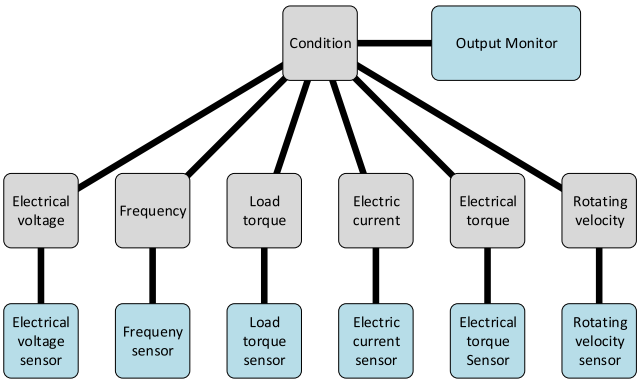


FIGURE 16. Architecture of CAH/CCAM for monitoring an AC motor.

The signal and system state detector functionalities keep historical information about their input to identify steady states as well as recognize state changes and drifting signals (see [29]–[31]). Each new signal sample is compared with historical information to detect if the signal changed state. The historical information consists of an average value in CAH and a sliding window history in CCAM. If the new sample is in close proximity to the saved data of a recorded state, it belongs to that state. Whether a signal is stable or drifting is extracted from the course of the historical data. The state of the system is then the composition of the individual signal states. For further details, we refer to our corresponding works.

The state and health condition of the observed signal/system are outputs of the functionalities. The difference between CAH and CCAM is in the configuration of signal and system state detection: they make binary threshold-based and confidence-based decisions, respectively. CCAM provides better results than CAH.

Adjusting CAH/CCAM for a different SuO takes only two simple steps: (i) defining a low-level agent with a signal state detector for each input and output signal of the SuO and (ii) connecting each low-level agent to the high-level one and associating each signal to the system state detector either as input signal or output signal. The system is easily scalable, but the system state detector could become a bottleneck in case of an extremely high number of signals. This potential scalability issue is not caused by RoSA but by the architecture of the implemented application. In the case of such a complex SuO with a massive number of signals, the problem could be scaled out by replacing the central system state detector with a corresponding hierarchy of those. In other words, the SuO would be split up in various subsystems that have their own system state detectors, which may be combined in a hierarchical structure. This approach shows the powerful implementation of RoSA and its self-aware functionalities. Furthermore, this approach would overcome not only the issue of a bottleneck but also enables highly systematic monitoring of the SuO.

3) SUMMARY

Context-aware detection of different signal and system states is now enabled in RoSA by corresponding functionalities. However, no state detector was implemented when we started to develop the application. In the experimental phase, we implemented state detection as custom code in combination with existing RoSA functionalities (abstraction, confidence, and history). Reusing functionalities in a modular way, facilitated our efforts to implement complex data processing for context-aware state detection. We turned the validated implementations of signal and system state detectors into functionalities, which can be reused and configured by application-specific rules.

VIII. DISCUSSION

Before concluding, we enumerate the lessons learned from developing RoSA and the open issues already identified. We organize the discussion in three themes: modeling self-awareness in Section VIII-A, the software implementation in Section VIII-B, and implementing on ES hardware in Section VIII-C.

A. MODELING SELF-AWARENESS

An important lesson was realizing how much application-dependent self-aware functionalities are. While a self-awareness property has some fundamental characteristic, a corresponding functionality may be implemented in different ways. For example, while confidence is a measure of how trustworthy the work of a task, part of the system, or the entire system is (Section VI-C), it may be calculated in many different ways [100] (see for example [31], [55], [101]). What interface to use for a self-awareness property depends on the actual usage. Therefore, each functionality must have a sophisticated interface to support modularity. This allows using functionalities directly or in combination with other functionalities to express more complex concepts. Whenever a new concept that cannot be built from the existing functionalities is to be developed, devising a modular interface for the new functionality is challenging but essential for reusability.

While the interfaces of functionalities are instrumental for reusability, details of their internal implementations can affect performance significantly. We provide a set of functionalities in RoSA; however, there might be better-working implementations. Hence, users of RoSA are not discouraged from adjusting and optimizing the implementations to their specific end-use. The modular design makes it possible to experiment with alternative implementations at will. In addition, users are encouraged to develop other functionalities whenever they have new ideas or specific needs.

We realize there is room for improving the modeling capabilities of RoSA. One limiting factor is the small set of implemented self-aware functionalities. Our research effort in self-awareness properties and functionality implementations will continue. We foresee exciting challenges in the area

TABLE 2. Static and dynamic characteristics of the case studies (Section VII) executed on different ES platforms with different ARM cores (Cortex-A7 and Cortex-A15 implement the 32-bit ARMv7-A architecture, Cortex-A53 implements the 64-bit ARMv8-A architecture) shows that each application can work in real-time on ES hardware; note that numbers of different applications are not to be compared as they implement independent algorithms.

Metrics	EWS			SA-EWS 1			SA-EWS 2			CAH/CCAM		
Cortex-	A7	A15	A53	A7	A15	A53	A7	A15	A53	A7	A15	A53
Executable Binary Size (kB)	798	798	817	911	911	933	1079	1079	1105	662	662	678
Maximum Allocated Memory* (kB)	3508	3508	3272	3584	3584	3440	3656	3656	3624	3348	3348	3404
Average Sample Processing Time** (ms)	0.46	0.22	0.39	0.74	0.35	0.64	1.64	0.77	1.50	7.41	4.37	7.21
Real-Time Sampling Period*** (ms)	1000	1000	1000	1000	1000	1000	1000	1000	1000	33	33	33

* Memory size includes code and all data during execution.
** The average is based on the total processing time including initialization of the application and reading sensor input files and writing output file, in single-threaded execution.
*** Maximum acceptable sample processing time for real-time execution based on application requirements.

and hope for the community's contribution in tackling them together to make RoSA a powerful common framework.

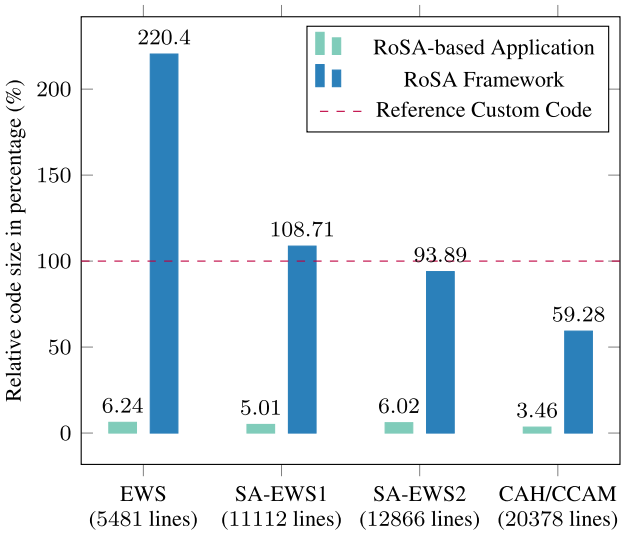


FIGURE 17. Ratio of non-comment codelines of RoSA-based application code (*Application*) and that of the RoSA framework itself (*Framework*) relative to the number of non-comment codelines of corresponding custom-written applications in our case studies. The base-line (actual number of codelines) for each application is indicated below the application names.

B. SOFTWARE IMPLEMENTATION

We made RoSA to accelerate and simplify research on self-awareness through a reusable software framework. Comparing the number of non-comment codelines (as an indicator of development effort) of our original custom-written applications and that of the RoSA-based implementation provides a quantitative measure of how much the development effort is simplified by using RoSA. This comparison for the presented case studies is shown in Figure 17, where the custom code of the corresponding application is the reference, meaning 100%. The *RoSA-based Application* sizes relative to the corresponding custom code show that RoSA-based implementations stay relatively small (3.46%–6.24%), independently from the size of the custom-written applications. Those implementations are small because they depend on the framework. The *RoSA Framework* size relative to the custom applications (the 100% references) shows that the overhead posed by the framework reduces (from an overhead of 120.4% for EWS to −40.72% for CAH/CCAM) as the size of the application increases (from 5481 to 20378 lines of custom application code for EWS and CAH/CCAM, respectively). The negative framework overhead indicates that even a custom implementation might be sub-optimal in case of complex applications. The quality of maintained framework code improves over time, while that does not typically happen with custom implementations developed in one go.

Averaging the sizes of all cases, we observe that on average, a RoSA-based implementation consists of only 5.18% lines of code relative to the custom implementation. The framework code has on average 20.75% more codelines than a custom application. The framework is, however, to be implemented only once and reused any number of times. Implementing a framework pays off when used for several applications. Particularly, implementing the framework and the four presented case studies in RoSA needed in total only 29.01% of the total number of codelines of our four original custom-written applications together; in other words, in total we needed 70.99% less codelines for implementing all four applications with RoSA. These figures confirm our initial hypothesis: an appropriate framework reduces the modeling and development efforts in self-aware systems.

Lastly, we realize that the capabilities and usability of RoSA as a software framework can be enhanced. For example, a graphical interface could help non-programmers to interact with models. Studying typical model patterns on both the agent system and the agent levels could help application developers in making better designs and utilizing available features efficiently.

C. IMPLEMENTING ON EMBEDDED SYSTEM HARDWARE

RoSA is a standalone actor framework with an open-source standard native implementation, to which CAF is the most similar from the existing agent-based frameworks (discussed

in Section III-F). For deploying RoSA in ESs, we limited the implemented features to the essentials for our case studies without limiting the generality of the agent system. The binary size of the RoSA libraries on a x86-64 linux machine is 300 kB, while that of the CAF core library (version 0.17.5) is 7248 kB. This significant (24 times) difference in favor of RoSA indicates that our framework is applicable to considerably smaller systems than CAF.

As a preliminary confirmation of this hypothesis (suitability for ESs), we ran our case studies on the *ODROID XU4* [102] and *Raspberry Pi 3* [103] systems. The former has an eight-core *big.LITTLE* [104] configuration with *Cortex-A7* and *Cortex-A15* cores and the latter has a quad-core configuration with *Cortex-A53* cores. The applications posed a moderate memory footprint well below 4 MB, which fits typical ESs, and processed samples several times faster than required for real-time execution. In Table 2, we have summarized the characteristics of each application implemented on each platform, where the real-time requirements and actual average processing times can be found. It has to be noted that the real-time sampling period depends on the nature of the corresponding application and that the table is not meant to compare the different applications. We plan to extend the evaluation of our software implementation by performing further extensive and vigorous tests by deploying RoSA on other real ES hardware in the future.

IX. CONCLUSION

Self-awareness is a hot topic, but related research and development efforts are fragmented among different fields and communities. Self-aware systems are developed from scratch in many cases. Such method of development, on long term, is redundant, inefficient and uneconomic. A major reason behind this fragmentation is the lack of a common framework that would facilitate development, cooperation, and reuse of existing results.

In this paper, we presented RoSA, a framework that aims to help researchers and engineers to explore the novel design space of self-awareness. RoSA supports modeling of self-aware applications as *agent systems* and modeling of agents based on self-aware *functionalities*. We presented the design principles of the RoSA architecture as well as use cases of RoSA-based modeling for different scenarios. The description of self-aware functionalities offered by RoSA and detailed case studies about applications implemented in RoSA demonstrate the modeling power and applicability of the framework.

Using RoSA relieves application developers from taking care of handling agents and message passing. Predefined functionalities serve as reusable components for defining individual agents. Data processing within agents can be defined as an arbitrary combination of custom application code and existing functionalities. Application code can thus be limited to the important aspects: (i) data processing within agents and (ii) the agent hierarchy of the application.

We promote RoSA as a vehicle for researchers to study various concepts that are related to self-awareness and the relation among them; and also for engineers to prototype and evaluate self-aware features in their designs with ease.

ABBREVIATIONS

AC	Autonomic Computing.
AI	Artificial Intelligence.
CAF	C++ Actor Framework.
CAH	Context-aware Health Monitoring.
CCAM	Confidence-based Context-Aware condition Monitoring.
CPS	Cyber-Physical System.
CSA	Computational Self-Awareness.
DARPA	Defense Advanced Research Projects Agency.
ES	Embedded System.
EWS	Early Warning Score.
IBM	International Business Machines Corporation.
IT	Information Technology.
MAS	Multi-agent System.
MPSoC	Multi-Processor System-on-Chip.
NASA	National Aeronautics and Space Administration.
ODA	Observe-Decide-Act.
RoSA	Research on Self-Awareness.
SoC	System on Chip.
SuO	System under Observation.

ACKNOWLEDGMENT

(Maximilian Götzinger and Dávid Juhász contributed equally to this work.) The authors acknowledge TU Wien Bibliothek for financial support through its Open Access Funding Programme.

REFERENCES

[1] J. Rivera and R. van der Meulen. (Nov. 2014). *Gartner Says 4.9 Billion Connected 'Things' Will be in use in 2015*. [Online]. Available: http://www.gartner.com/newsroom/id/2905717

[2] L. Atzori, A. Iera, and G. Morabito, "The Internet of Things: A survey," *Comput. Netw.*, vol. 54, no. 15, pp. 2787–2805, Oct. 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568

[3] M. Yaqoob, S. R. Qaisrani, M. Waqas, Y. Ayaz, S. Iqbal, and S. Nisar, "Control of robotic arm manipulator with haptic feedback using programmable system on chip," in *Proc. Int. Conf. Robot. Emerg. Allied Technol. Eng. (iCREATE)*, Apr. 2014, pp. 300–305.

[4] D. Genius, E. Faure, and N. Pouillon, "Mapping a telecommunication application on a multiprocessor system-on-chip," in *Algorithm-Architecture Matching for Signal and Image Processing*. Dordrecht, The Netherlands: Springer, 2011, pp. 53–77.

[5] S. Mukhopadhyay, M. Heddes, M. Ravasi, and M. S. Yeo, "System-on-a-chip and multi-chip systems supporting advanced telecommunication functions," U.S. Patent 12 342 625, Jun. 24, 2010.

[6] S. Liu, J. Tang, Z. Zhang, and J.-L. Gaudiot, "Computer architectures for autonomous driving," *Computer*, vol. 50, no. 8, pp. 18–25, 2017.

[7] X. Chen, X. Jiang, and L. Wang, "Development on ARM9 System-on-chip embedded sensor node for urban intelligent transportation system," in *Proc. IEEE Int. Symp. Ind. Electron.*, vol. 4, Jul. 2006, pp. 3270–3275.

[8] N. Moreira, J. Lazaro, U. Bidarte, J. Jimenez, and A. Astarloa, "On the utilization of system-on-chip platforms to achieve nanosecond synchronization accuracies in substation automation systems," *IEEE Trans. Smart Grid*, vol. 8, no. 4, pp. 1932–1942, Jul. 2017.

[9] B. Massot, C. Gehin, R. Nocua, A. Dittmar, and E. McAdams, "A wearable, low-power, health-monitoring instrumentation based on a programmable system-on-chipTM," in *Proc. Annu. Int. Conf. IEEE Eng. Med. Biol. Soc.*, Sep. 2009, pp. 4852–4855.

[10] L. Gurgen, O. Gunalp, Y. Benazzouz, and M. Galissot, "Self-aware cyber-physical systems and applications in smart buildings and cities," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2013, pp. 1149–1154.

[11] A. Jantsch, A. Anzanpour, H. Kholerdi, I. Azimi, L. C. Siafara, A. M. Rahmani, N. TaheriNejad, P. Liljeberg, and N. Dutt, "Hierarchical dynamic goal management for IoT systems," in *Proc. 19th Int. Symp. Qual. Electron. Design (ISQED)*, Mar. 2018, pp. 370–375.

[12] M. Shafique, D. Gnad, S. Garg, and J. Henkel, "Variability-aware dark silicon management in on-chip many-core systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib.* San Jose, CA, USA: EDA Consortium, Mar. 2015, pp. 387–392.

[13] W. Huang, M. R. Stant, K. Sankaranarayanan, R. J. Ribando, and K. Skadron, "Many-core design from a thermal perspective," in *Proc. 45th Annu. Conf. Design Autom. (DAC)*, Jun. 2008, pp. 746–749.

[14] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proc. 50th Annu. Design Autom. Conf. (DAC)*. New York, NY, USA: ACM, 2013, pp. 1:1–1:10

[15] N. Dutt, A. Jantsch, and S. Sarma, "Self-aware cyber-physical systems-on-chip," in *Proc. IEEE/ACM Int. Conf. Comput.-Aided Design (ICCAD)*, Austin, TX, USA, Nov. 2015, pp. 46–50.

[16] P. R. Lewis, M. Platzner, B. Rinner, J. Tørresen, and X. Yao, *Self-Aware Computing Systems: An Engineering Approach* (Natural Computing Series), 1st ed. Cham, Switzerland: Springer, 2016, doi: 10.1007/978-3-319-39675-0.

[17] N. Dutt and N. TaheriNejad, "Self-awareness in cyber-physical systems," in *Proc. 29th Int. Conf. VLSI Design 15th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2016, pp. 5–6.

[18] K. Bellman, N. Dutt, L. Esterle, A. Herkersdorf, A. Jantsch, C. Landauer, P. R. Lewis, M. Platzner, N. TaheriNejad, and K. Tammemäe, "Self-aware cyber-physical systems," *ACM Trans. Cyber-Phys. Syst.*, vol. 4, no. 4, pp. 1–24, 2020.

[19] J.-S. Preden, K. Tammemäe, A. Jantsch, M. Leier, A. Riid, and E. Calis, "The benefits of self-awareness and attention in fog and mist computing," *Computer*, vol. 48, no. 7, pp. 37–45, Jul. 2015.

[20] F. Forooghifar, A. Aminifar, and D. Atienza, "Resource-aware distributed epilepsy monitoring using self-awareness from edge to cloud," *IEEE Trans. Biomed. Circuits Syst.*, vol. 13, no. 6, pp. 1338–1350, Dec. 2019.

[21] S. Kounev, X. Zhu, J. O. Kephart, and M. Kwiatkowska, "Model-driven algorithms and architectures for self-aware computing systems (Dagstuhl seminar 15041)," *Dagstuhl Rep.*, vol. 5, no. 1, pp. 164–196, 2015.

[22] T. Chen, F. Faniyi, R. Bahsoon, P. R. Lewis, X. Yao, L. L. Minku, and L. Esterle, "The handbook of engineering self-aware and self-expressive systems," *CoRR*, vol. abs/1409.1793, pp. 1–81, Sep. 2014. [Online]. Available: http://arxiv.org/abs/1409.1793

[23] H. Psaier and S. Dustdar, "A survey on self-healing systems: Approaches and systems," *Computing*, vol. 91, no. 1, pp. 43–73, Jan. 2011.

[24] P. R. Lewis, A. Chandra, F. Faniyi, K. Glette, T. Chen, R. Bahsoon, J. Torresen, and X. Yao, "Architectural aspects of self-aware and self-expressive computing systems: From psychology to engineering," *Computer*, vol. 48, no. 8, pp. 62–70, Aug. 2015.

[25] T. Bures, D. Weyns, B. Schmerl, J. Fitzgerald, A. Aniculaesei, C. Berger, J. Cambeiro, J. Carlson, S. A. Chowdhury, M. Daun, and N. Li, "Software engineering for smart cyber-physical systems (SEsCPS 2018)-workshop report," *ACM SIGSOFT Softw. Eng. Notes*, vol. 44, no. 4, pp. 11–13, 2019.

[26] M. Götzinger, N. Taherinejad, A. M. Rahmani, P. Liljeberg, A. Jantsch, and H. Tenhunen, "Enhancing the early warning score system using data confidence," in *Proc. Int. Conf. Wireless Mobile Commun. Healthcare*. Cham, Switzerland: Springer, 2016, pp. 91–99.

[27] M. Götzinger, A. Anzanpour, I. Azimi, N. Taherinejad, and A. M. Rahmani, "Enhancing the self-aware early warning score system through fuzzified data reliability assessment," in *Proc. Int. Conf. Wireless Mobile Commun. Healthcare*. Cham, Switzerland: Springer, 2017, pp. 3–11.

[28] M. Götzinger, A. Anzanpour, I. Azimi, N. TaheriNejad, A. Jantsch, A. M. Rahmani, and P. Liljeberg, "Confidence-enhanced early warning score based on fuzzy logic," *Mobile Netw. Appl.*, vol. 8, pp. 1–18, Aug. 2019.

[29] M. Gotzinger, N. TaheriNejad, H. A. Kholerdi, and A. Jantsch, "On the design of context-aware health monitoring without *a priori* knowledge; an AC-motor case-study," in *Proc. IEEE 30th Can. Conf. Electr. Comput. Eng. (CCECE)*, Apr. 2017, pp. 1–5.

[30] M. Gotzinger, E. Willegger, N. TaheriNejad, A. Jantsch, T. Sauter, T. Glatzl, and P. Lilieberg, "Applicability of context-aware health monitoring to hydraulic circuits," in *Proc. IECON-44th Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2018, pp. 4712–4719.

[31] M. Götzinger, N. TaheriNejad, H. A. Kholerdi, A. Jantsch, E. Willegger, T. Glatzl, A. M. Rahmani, T. Sauter, and P. Liljeberg, "Model-free condition monitoring with confidence," *Int. J. Comput. Integr. Manuf.*, vol. 32, nos. 4–5, pp. 466–481, May 2019.

[32] F. Faniyi, P. R. Lewis, R. Bahsoon, and X. Yao, "Architecting self-aware software systems," in *Proc. IEEE/IFIP Conf. Softw. Archit.*, Apr. 2014, pp. 91–94.

[33] L. Guang, E. Nigussie, J. Plosila, J. Isoaho, and H. Tenhunen, "Survey of self-adaptive NoCs with energy-efficiency and dependability," *Int. J. Embedded Real-Time Commun. Syst.*, vol. 3, no. 2, pp. 1–22, Apr. 2012.

[34] J. Schlingensiepen, F. Nemtanu, R. Mehmood, and L. McCluskey, "Autonomic transport management systems—Enabler for smart cities, personalized medicine, participation and industry grid/industry 4.0," in *Intelligent Transportation Systems–Problems and Perspectives*. Cham, Switzerland: Springer, 2016, pp. 3–35.

[35] D. B. Abeywickrama and E. Ovaska, "A survey of autonomic computing methods in digital service ecosystems," *Service Oriented Comput. Appl.*, vol. 11, no. 1, pp. 1–31, Mar. 2017.

[36] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Unconventional Programming Paradigms*, J.-P. Banâtre, P. Fradet, J.-L. Giavitto, and O. Michel, Eds. Berlin, Germany: Springer, 2005, pp. 257–269.

[37] M. C. Huebscher and J. A. McCann, "A survey of autonomic computing—Degrees, models, and applications," *ACM Comput. Surv.*, vol. 40, no. 3, pp. 1–28, Aug. 2008, doi: 10.1145/1380584.1380585.

[38] A. L. Randall and R. C. Walter, "Overview of the small unit operations situational awareness system," in *Proc. IEEE Mil. Commun. Conf. (MILCOM)*, vol. 1, Oct. 2003, pp. 169–173.

[39] M. Rahman, R. Ranjan, R. Buyya, and B. Benatallah, "A taxonomy and survey on autonomic management of applications in grid computing environments," *Concurrency Comput., Pract. Exp.*, vol. 23, no. 16, pp. 1990–2019, Nov. 2011. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1734

[40] J. O. Kephart, "Research challenges of autonomic computing," in *Proc. 27th Int. Conf. Softw. Eng. (ICSE)*, 2005, pp. 15–22.

[41] A. G. Ganek and T. A. Corbi, "The dawning of the autonomic computing era," *IBM Syst. J.*, vol. 42, no. 1, pp. 5–18, 2003.

[42] D. Sinreich, "An architectural blueprint for autonomic computing," IBM Corp., Armonk, NY, USA, White paper, 2006. [Online]. Available: https://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[43] P. Lalanda, J. A. McCann, and A. Diaconescu, *Autonomic Computing: Principles, Design and Implementation*. London, U.K.: Springer, 2013.

[44] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003.

[45] J. Cámara, K. L. Bellman, J. O. Kephart, M. Autili, N. Bencomo, A. Diaconescu, H. Giese, S. Götz, P. Inverardi, S. Kounev, and M. Tivoli, *Self-aware Computing Systems: Related Concepts and Research Areas*. Cham, Switzerland: Springer, 2017, pp. 17–49, doi: 10.1007/978-3-319-47474-8_2.

[46] D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, and M. Vanover, "Autonomic personal computing," *IBM Syst. J.*, vol. 42, no. 1, pp. 165–176, 2003.

[47] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auto. Adapt. Syst.*, vol. 4, no. 2, p. 14, 2009.

[48] C. Landauer and K. L. Bellman, "An architecture for self-awareness experiments," in *Proc. IEEE Int. Conf. Autonomic Comput. (ICAC)*, Jul. 2017, pp. 255–262.

[49] K. L. Bellman, "An approach to integrating and creating flexible software environments supporting the design of complex systems," in *Proc. Winter Simul. Conf.*, 1991, pp. 1101–1105.
[50] M. Salehie and L. Tahvildari, "Autonomic computing: Emerging trends and open problems," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, 2005.
[51] H. Giese, T. Vogel, A. Diaconescu, S. Götz, N. Bencomo, K. Geihs, S. Kounev, and K. L. Bellman, *State of the Art in Architectures for Self-Aware Computing Systems*. Cham, Switzerland: Springer, 2017, pp. 237–275, doi: 10.1007/978-3-319-47474-8_8.
[52] N. Dutt, A. Jantsch, and S. Sarma, "Toward smart embedded systems: A self-aware system-on-chip (SOC) perspective," *ACM Trans. Embed. Comput. Syst.*, vol. 15, no. 2, pp. 22:1–22:27, Feb. 2016.
[53] P. Mercati, A. Bartolini, F. Paterna, T. S. Rosing, and L. Benini, "A linux-governor based dynamic reliability manager for Android mobile devices," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2014, pp. 1–4.
[54] B. Rinner, L. Esterle, J. Simonjan, G. Nebehay, R. Pflugfelder, G. F. Dominguez, and P. R. Lewis, "Self-aware and self-expressive camera networks," *Computer*, vol. 48, no. 7, pp. 21–28, Jul. 2015.
[55] F. Forooghifar, A. Aminifar, and D. A. Alonso, "Self-aware wearable systems in epileptic seizure detection," in *Proc. 21st Euromicro Conf. Digit. Syst. Design (DSD)*, Aug. 2018, pp. 426–432.
[56] B. Jennings and R. Stadler, "Resource management in clouds: Survey and research challenges," *J. Netw. Syst. Manage.*, vol. 23, no. 3, pp. 567–619, Jul. 2015, doi: 10.1007/s10922-014-9307-7.
[57] P. Spathis and M. D. D. Bicudo, "Ana: Autonomic network architecture," in *Autonomic Network Management Principles: From Concepts to Applications*. Oxford, U.K.: Academic, 2011, p. 49.
[58] L. Wanner, S. Elmalaki, L. Lai, P. Gupta, and M. Srivastava, "VarEMU: An emulation testbed for variability-aware software," in *Proc. Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Sep. 2013, pp. 1–10.
[59] J. Strassner, S.-S. Kim, and J. W.-K. Hong, "The design of an autonomic communication element to manage future Internet services," in *Management Enabling the Future Internet for Changing Business and New Computing Services*. Berlin, Germany: Springer, 2009, pp. 122–132.
[60] W. Baek and T. M. Chilimbi, "Green: A framework for supporting energy-conscious programming using controlled approximation," in *Proc. ACM SIGPLAN Notices*, vol. 45, no. 6, 2010, pp. 198–209.
[61] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "SEEC: A framework for self-aware computing," MIT, Cambrige, MA, USA, Tech. Rep. MIT-CSAIL-TR-2010-049, Oct. 2010.
[62] E. Shamsa, A. Kanduri, N. TaheriNejad, A. Probstl, S. Chakraborty, A. M. Rahmani, and P. Liljeberg, "User-centric resource management for embedded multi-core processors," in *Proc. 33rd Int. Conf. VLSI Design 19th Int. Conf. Embedded Syst. (VLSID)*, Jan. 2020, pp. 1–6.
[63] A. Akbar and P. R. Lewis, "Self-adaptive and self-aware mobile-cloud hybrid robotics," in *Proc. 5th Int. Conf. Internet Things, Syst., Manage. Secur.*, Oct. 2018, pp. 262–267.
[64] L. C. Siafara, H. A. Kholerdi, A. Bratukhin, N. TaheriNejad, A. Wendt, A. Jantsch, A. Treytl, and T. Sauter, "SAMBA: A self-aware health monitoring architecture for distributed industrial systems," in *Proc. IECON-43rd Annu. Conf. IEEE Ind. Electron. Soc.*, Oct. 2017, pp. 3512–3517.
[65] L. C. Siafara, H. Kholerdi, A. Bratukhin, N. Taherinejad, and A. Jantsch, "SAMBA–an architecture for adaptive cognitive control of distributed cyber-physical production systems based on its self-awareness," *e i Elektrotechnik und Informationstechnik*, vol. 135, no. 3, pp. 270–277, Jun. 2018, doi: 10.1007/s00502-018-0614-7.
[66] K. Nymoen, A. Chandra, and J. Torresen, "Self-awareness in active music systems," *Self-Aware Computing Systems*. Cham, Switzerland: Springer, 2016, pp. 279–296, doi: 10.1007/978-3-319-39675-0_14.
[67] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," in *Multiprocessor System-on-Chip: Hardware Design and Tool Integration*, M. Hübner and J. Becker, Eds. Berlin, Germany: Springer, 2011, pp. 241–268.
[68] A. Bouajila, J. Zeppenfeld, W. Stechele, A. Bernauer, O. Bringmann, W. Rosenstiel, and A. Herkersdorf, "Autonomic system on chip platform," in *Organic Computing—A Paradigm Shift for Complex Systems* (Autonomic Systems), C. Müller-Schloer, H. Schmeck, and T. Ungerer, Eds. Basel, Switzerland: Birkhäuser, 2011, ch. 4.7, pp. 413–425.
[69] A. Bouajila, J. Zeppenfeld, W. Stechele, A. Herkersdorf, A. Bernauer, O. Bringmann, and W. Rosenstiel, "Organic computing at the system on chip level," in *Proc. IFIP Int. Conf. Very Large Scale Integr.*, Oct. 2006, pp. 338–341.
[70] G. Kornaros and D. Pnevmatikatos, "A survey and taxonomy of on-chip monitoring of multicore systems-on-chip," *ACM Trans. Design Autom. Electron. Syst.*, vol. 18, no. 2, pp. 1–38, Mar. 2013.
[71] N. T. Nejad, M. A. Shami, and P. D. S. Manoj, "Self-aware sensing and attention-based data collection in multi-processor system-on-chips," in *Proc. 15th IEEE Int. New Circuits Syst. Conf. (NEWCAS)*, Jun. 2017, pp. 81–84.
[72] S. M. Jafri, L. Guang, A. Jantsch, K. Paul, A. Hemani, and H. Tenhunen, "Self-adaptive NoC power management with dual-level agents-architecture and implementation," in *Proc. PECCS*, 2012, pp. 450–458.
[73] S. Kounev, P. Lewis, K. Bellman, N. Bencomo, J. Camara, A. Diaconescu, L. Esterle, K. Geihs, H. Giese, S. Götz, P. Inverardi, J. Kephart, and A. Zisman, "The notion of self-aware computing," in *Self-Aware Computing Systems*, S. Kounev, J. O. Kephart, A. Milenkoski, and X. Zhu, Eds. Cham, Switzerland: Springer, 2017, pp. 3–16.
[74] J. Kramer and J. Magee, "Self-managed systems: An architectural challenge," in *Proc. Future Softw. Eng. (FOSE)*, May 2007, pp. 259–268.
[75] U. Aßmann, S. Götz, J.-M. Jézéquel, B. Morin, and M. Trapp, *A Reference Architecture and Roadmap for Models Run.Time Systems*. Cham, Switzerland: Springer, 2014, pp. 1–18, doi: 10.1007/978-3-319-08915-7_1.
[76] L. Guang, E. Nigussie, J. Isoaho, P. Rantala, and H. Tenhunen, "Interconnection alternatives for hierarchical monitoring communication in parallel SoCs," *Microprocessors Microsyst.*, vol. 34, no. 5, pp. 118–128, Aug. 2010.
[77] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson, "Pervasive ecosystems: A coordination model based on semantic chemistry," in *Proc. 27th Annu. ACM Symp. Appl. Comput.* New York, NY, USA: ACM, 2012, pp. 295–302.
[78] C. Savaglio, G. Fortino, and M. Zhou, "Towards interoperable, cognitive and autonomic IoT systems: An agent-based approach," in *Proc. IEEE 3rd World Forum Internet Things (WF-IoT)*, Dec. 2016, pp. 58–63.
[79] I. Carreras, I. Chlamtac, F. De Pellegrini, and D. Miorandi, "BIONETS: Bio-inspired networking for pervasive communication environments," *IEEE Trans. Veh. Technol.*, vol. 56, no. 1, pp. 218–229, Jan. 2007.
[80] A. Bucchiarone, "Collective adaptation through multi-agents ensembles: The case of smart urban mobility," *ACM Trans. Auto. Adapt. Syst.*, vol. 14, no. 2, pp. 1–28, Dec. 2019.
[81] A. Bucchiarone, M. De Sanctis, A. Marconi, and A. Martinelli, "DeMOCAS: Domain objects for service-based collective adaptive systems," in *Service-Oriented Computing—ICSOC 2016 Workshops*. Cham, Switzerland: Springer, 2017, pp. 174–178. [Online]. Available: http://link.springer.com/10.1007/978-3-319-68136-8_19
[82] S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. London, U.K.: Pearson, 2010.
[83] M. Wooldridge and N. R. Jennings, "Intelligent agents: Theory and practice," *Knowl. Eng. Rev.*, vol. 10, no. 2, pp. 115–152, Jun. 1995.
[84] C. Hewitt, "Actor model of computation for scalable robust information systems," in *Proc. Symp. Logic Collaboration Intell. Appl.*, 2017, pp. 1–91.
[85] A. Sadighi, B. Donyanavard, T. Kadeed, K. Moazzemi, T. Muck, A. Nassar, A. M. Rahmani, T. Wild, N. Dutt, R. Ernst, A. Herkersdorf, and F. Kurdahi, "Design methodologies for enabling self-awareness in autonomous systems," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, Mar. 2018, pp. 1532–1537.
[86] L. Guang, "Hierarchical agent-based adaptation for self-aware embedded computing systems," Ph.D. dissertation, Dept. Inf. Technol., Univ. Turku, Turku, Finland, 2012.
[87] J. Hunt, *Introduction to Akka Actors*. Springer, 2014, pp. 383–398.
[88] D. Charousset, R. Hiesgen, and T. C. Schmidt, "Revisiting actor programming in C++," *Comput. Lang., Syst. Struct.*, vol. 45, pp. 105–131, Apr. 2016.
[89] P. Taillandier, B. Gaudou, A. Grignard, Q.-N. Huynh, N. Marilleau, P. Caillou, D. Philippon, and A. Drogoul, "Building, composing and experimenting complex spatial models with the GAMA platform," *GeoInformatica*, vol. 23, no. 2, pp. 299–322, Apr. 2019.

[90] F. Bellifemine, F. Bergenti, G. Caire, and A. Poggi, *Jade—A Java Agent Development Framework*. Boston, MA, USA: Springer, 2005, pp. 125–147.
[91] B. Chen, H. H. Cheng, and J. Palen, "Mobile-C: A mobile agent platform for mobile C/C++ agents," *Softw. Pract. Exper.*, vol. 36, no. 15, pp. 1711–1733, 2006.
[92] N. Collier and M. North, "Parallel agent-based simulation with repast for high performance computing," *Simulation*, vol. 89, no. 10, pp. 1215–1235, Oct. 2013.
[93] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with repast simphony," *Complex Adapt. Syst. Model.*, vol. 1, no. 1, p. 3, Dec. 2013.
[94] N. TaheriNejad, A. Jantsch, and D. Pollreisz, "Comprehensive observation and its role in self-awareness; an emotion recognition system example," in *Proc. Position Papers Federated Conf. Comput. Sci. Inf. Syst.*, Gdansk, Poland, Oct. 2016, pp. 117–124.
[95] A. Jantsch and K. Tammemäe, "A framework of awareness for artificial subjects," in *Proc. 2014 Int. Conf. Hardw./Softw. Codesign Syst. Synth.* New York, NY, USA: ACM, 2014, pp. 20:1–20:3.
[96] L. A. Zadeh, "Fuzzy sets," *Inf. Control*, vol. 8, no. 3, pp. 338–353, Jun. 1965.
[97] J. McGaughey, F. Alderdice, R. Fowler, A. Kapila, A. Mayhew, and M. Moutray, "Outreach and early warning systems (EWS) for the prevention of intensive care admission and death of critically ill adult patients on general hospital wards," *Cochrane Library*, vol. 2007, no. 3, pp. CD005529:1–CD005529:24, Jul. 2007.
[98] R. J. Morgan, F. Williams, and M. M. Wright, "An early warning scoring system for detecting developing critical illness," *Clin. Intensive Care*, vol. 8, no. 2, p. 100, 1997.
[99] W. Thomson and R. Gilmore, "Motor current signature analysis to detect faults in induction motor drives–fundamentals, data interpretation, and industrial case histories," in *Proc. 32nd Turbomachinery Symp.*, Sep. 2003, pp. 145–156.
[100] N. TaheriNejad and A. Jantsch, "Improved machine learning using confidence," in *Proc. IEEE Can. Conf. Electr. Comput. Eng. (CCECE)*, May 2019, pp. 1–5.
[101] H. A. Kholerdi, N. TaheriNejad, and A. Jantsch, "Enhancement of classification of small data sets using self-awareness—An iris flower case-study," in *Proc. IEEE Int. Symp. Circuits Syst. (ISCAS)*, May 2018, pp. 1–5.
[102] HADRKERNEL. (2017). *ODROID-XU4 Manual*. [Online]. Available: https://magazine.odroid.com/wp-content/uploads/odroid-xu4-user-manual.pdf
[103] Raspberry Pi (Trading) Ltd. (2019). *Raspberry Pi Compute Module 3+ Datasheet*. [Online]. Available: https://www.raspberrypi.org/documentation/hardware/computemodule/datasheets/rpi_DATA_CM3plus_1p0.pdf
[104] LITTLE Technology. (2013). *The Future of Mobile*. [Online]. Available: https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf

MAXIMILIAN GÖTZINGER (Member, IEEE) received the B.Sc. and M.Sc. degrees in electrical engineering and information technology from TU Wien (formerly known as the Vienna University of Technology as well), Vienna, Austria, in 2012 and 2015, respectively. He is currently pursuing the Ph.D. degree in computer science with the Department of Future Technologies, University of Turku.

He is also with the Institute of Computer Technology, TU Wien, as a Project Assistant and a Teacher. He has a keen and serious interest in computer science and engineering, as well as teaching. His research interest includes computational self-awareness, for which he is conducting many case studies, such as health and system monitoring. He has published ten peer-reviewed papers, for one of which he received the Best Paper Award. In 2019, he received the one Best Teacher Award and the one Best Lecturer Award for the course digital systems.

DÁVID JUHÁSZ received the B.Sc. and M.Sc. degrees in computer science from Eötvös Loránd University, Budapest, Hungary, in 2010 and 2012, respectively. He is currently pursuing the Ph.D. degree with the Institute of Computer Technology, TU Wien, Vienna, Austria.

He is an Early Stage Researcher of the oCPS Marie Curie ITN Project at TU Wien. He is also a Lead Software Architect at Imsys AB, Stockholm, Sweden. His research interests include development methodologies and runtime systems that enable efficient utilization of complex hardware solutions via a high-level software environment. His current research interests include self-aware systems and execution issues of the state-of-the-art hardware platforms focusing on non-functional requirements. He had contributed to software development on different levels of abstraction as well as design and implementation questions of programming languages, runtime systems, and instruction set architectures.

NIMA TAHERINEJAD (Member, IEEE) received the Ph.D. degree in electrical and computer engineering from The University of British Columbia, Vancouver, Canada, in 2015.

He is currently a "Universitätsassistant" at TU Wien (formerly known also as the Vienna University of Technology), Vienna, Austria, where his areas of work include self-awareness in resource-constrained cyber-physical systems, embedded systems, systems on chip, health-care, memristor-based circuit and systems, and robotics. He has published two books and more than 45 peer-reviewed articles. He received several awards and scholarships from universities, conferences, and workshops he has attended. He has also served as a reviewer, an editor, an organizer, and the chair for various journals, conferences, and workshops.

EDWIN WILLEGGER received the B.Sc. degree in electrical engineering from TU Wien, Vienna, Austria, in 2018, where he is currently pursuing the master's degree in microelectronics and photonics.

From 2014 to 2015, he was with Siemens Austria and was doing research on preventive maintenance of complex mechanical systems. Since 2015, he has been a Research Assistant at TU Wien. His research interest includes the development of self-aware hardware and software systems.

BENEDIKT TUTZER received the B.S. degree in computer engineering from TU Wien, Vienna, Austria, in 2018, where he is currently pursuing the master's degree in embedded systems.

In 2017 and 2018, he was with the Interactive Media Systems Group, TU Wien, researching the applications of virtual-reality headsets as a seeing aid for visually impaired patients at the Vienna General Hospital. Since 2018, he has been with the Institute of Computer Technology, TU Wien, focusing on electronic design automation.

PASI LILJEBERG (Member, IEEE) received the M.Sc. and Ph.D. degrees in information and communication technology from the University of Turku, Turku, Finland, in 1999 and 2005, respectively. He received an Adjunct Professorship in embedded computing architectures, in 2010. He is currently a Full Professor with the Digital Health Technology, University of Turku. He has authored more than 300 peer-reviewed publications. His current research interests include biomedical engineering, the Internet of Things, fog computing, approximate and adaptive computing, wearable sensor, e-health technology, and health data analytics. In that context, he has established and leading the Internet-of-Things for Healthcare (IoT4Health) Research Group.

AXEL JANTSCH (Senior Member, IEEE) received the Dipl.Ing. and Ph.D. degrees in computer science from TU Wien, Vienna, Austria, in 1987 and 1992, respectively.

From 1997 to 2002, he was an Associate Professor at the KTH Royal Institute of Technology, Stockholm, where he was a full Professor in electronic systems design, from 2002 to 2014. Since 2014, he has been a Professor of systems on chips at the Institute of Computer Technology, TU Wien. He has published five books as an Editor and one as an Author, over 300 peer-reviewed contributions in journals, books, and conference proceedings. He has given over 100 invited presentations at conferences, universities, and companies. His current research interests include systems on chips, self-aware cyber-physical systems, and embedded machine learning.

AMIR M. RAHMANI (Senior Member, IEEE) is currently an Assistant Professor of computer science and nursing (joint appointment) at UCI and is also a Life-Time Adjunct Professor (Docent) at the Department of Future Technologies, University of Turku, Turku, Finland. He is the Founder of the Health SciTech Group, University of California at Irvine (UCI), and the Co-Founder of the Internet-of-Things for Healthcare Group (IoT4Health), University of Turku (UTU). He has coauthored more than 200 peer-reviewed publications. His research interests include the Internet of Things (IoT), e-health, wearable sensor design, bio-signal processing, health informatics, and big health data analytics. He is especially excited about novel sensing, computation/analytics, communication, and networking paradigms, applied to healthcare/medical and well-being applications. He is the Associate Editor-in-Chief of the *ACM Transactions on Computing for Healthcare*.