Journal of Integrated Design and Process Science 26 (2022) 351–367
DOI 10.3233/JID-220003
IOS Press

The vision of self-evolving computing systems

Danny Weyns[a,*], Thomas Bäck[b], Renè Vidal[c], Xin Yao[d] and Ahmed Nabil Belbachir[e]
[a]*Katholieke Universiteit Leuven, Belgium and Linnaeus University, Sweden*
[b]*Leiden University, The Netherlands and NORCE Norwegian Research Centre, Norway*
[c]*Johns Hopkins University, USA and NORCE Norwegian Research Centre, Norway*
[d]*University of Birmingham, UK and Southern University of Science and Technology, China*
[e]*NORCE Norwegian Research Centre, Norway*

Abstract. Computing systems are omnipresent; their sustainability has become crucial for our society. A key aspect of this sustainability is the ability of computing systems to cope with the continuous change they face, ranging from dynamic operating conditions, to changing goals, and technological progress. While we are able to engineer smart computing systems that autonomously deal with various types of changes, handling unanticipated changes requires system evolution, which remains in essence a human-centered process. This will eventually become unmanageable. To break through the status quo, we put forward an arguable opinion for the vision of *self-evolving computing systems* that are equipped with an evolutionary engine enabling them to evolve autonomously. Specifically, when a self-evolving computing system detects conditions outside its operational domain, such as an anomaly or a new goal, it activates an evolutionary engine that runs online experiments to determine how the system needs to evolve to deal with the changes, thereby evolving its architecture. During this process the engine can integrate new computing elements that are provided by computing warehouses. These computing elements provide specifications and procedures enabling their automatic integration. We motivate the need for self-evolving computing systems in light of the state of the art, outline a conceptual architecture of self-evolving computing systems, and illustrate the architecture for a future smart city mobility system that needs to evolve continuously with changing conditions. To conclude, we highlight key research challenges to realize the vision of self-evolving computing systems.

Keywords: Unanticipated change, sustainability, computing warehouses, self-adaptation, self-evolution

1. Introduction

Our society is going through a digitization process that penetrates virtually every aspect of our life, from health and industries, to transportation, public services, and entertainment. Consequently, we increasingly depend on the sustainability of computing systems. Yet, achieving this sustainability is challenging (Bernardo & Hillston, 2007; European-Commission, 2021; Lehman & Ramil, 2003) and spans manifold areas, from quality of service and software evolution to energy-awareness and software engineering processes. One key aspect to achieve sustainability of computing systems is managing the complexity that arises from the ever changing conditions these systems face. Such changes may or may not be anticipated when the system was built and include dynamics in the environment, new emerging goals,[1] and the introduction of new technologies. We take this angle of change to sustainability of computing systems.

*Corresponding author: Danny Weyns, Katholieke Universiteit Leuven, Belgium and Linnaeus University, Sweden. Tel: (+32)474-208251. E-mail: danny.weyns@kuleuven.be.
[1]We use goals and requirements interchangeably in this paper.

ISSN 1092-0617/$35.00 © 2022 – IOS Press. All rights reserved.

Currently we can build smart computing systems that can deal with many tasks autonomously, adapt themselves or learn over time to deal with changes. Other tasks can be managed by system operators, for instance, perform predictive maintenance. However, current computing systems can only handle changes that were anticipated, that is, changes that occur within the operational domain for which the system has been built. Current smart computing systems cannot handle unanticipated changes, such as anomalies outside their operational domain, and the emerge of new goals or new technologies. Such changes require evolution of the computing system. Although significant progress has been made on automating the deployment and integration of new elements, software evolution remains in essence a human-driven activity.

With the ever increasing complexity of computing systems and the continuous changes these systems are subjected to, human-driven approaches will eventually become unmanageable (Andersson, Baresi, Ben como, de Lemos, Gorla, Inverardi, & Vogel, 2013; Baresi & Ghezzi, 2010; Bennett & Rajlich, 2000; Dearle, 2007; Reussner, Goedicke, Hasselbring, Vogel-Heuser, Keim, & Martin, 2019). The capacity to handle large amounts of data and the availability of efficient decision algorithms opens perspectives to major breakthroughs towards fully autonomous systems that operate in continuous changing environments (Det- Norske-Veritas, 2020; Weyns, Andersson, Caporuscio, Flammini, Kerren, & Löwe, 2022; Weyns, Bures, Calinescu, Craggs, Fitzgerald, Garlan, Nuseibeh, Pasquale, Rashid, Ruchkin, & Schmerl, 2021b). However, we currently lack fundamental knowledge to turn these long-standing challenges into reality.

When comparing the capabilities of present-day computing systems with those of biological systems a few striking conclusions can be drawn. In contrast to computing systems, biological systems have a remarkable ability to deal with changes. For instance, insects have exceptionally fast reactions and can avoid dangerous situations or locate hidden food sources by swiftly adapting to their environment (Camazine, Deneubourg, Franks, Sneyd, Theraulas, & Bonabeau, 2003). They have also *evolved* dramatically, from one generation to the next, to accommodate changes over time in their habitat and the climate conditions.

Inspired by the principles of biological systems, this paper puts forward an arguable opinion for the vision of *self-evolving computing system*s, i.e., computing systems that evolve themselves autonomously. Figure 1 illustrates how self-evolving computing systems differ from traditional computing systems. A *traditional computing system* takes inputs from the environment and produces outputs in the environment, realizing the users' goals (Jackson, 1997). To deal with changing conditions, such a system can be equipped with smart techniques, either internally (e.g., a learning algorithm) or externally via a feedback loop, enabling the system to *self-adapt* its configuration autonomously to deal with changes (Garlan, Cheng, Huang, Schmerl, & Steenkiste, 2004; Weyns, 2021). A traditional computing system is designed to work in an operational domain, i.e., well-defined conditions of the environment in which the system should achieve its goals. Humans may be involved to *operate* the system, for instance to start/stop the execution of batches of tasks or to perform predictive maintenance. Extending the operational domain, for instance to deal with *new goals or new constraints*, or to mitigate *anomalies*, requires the system to undergo an *evolution* step that typically relies on humans that produce new computing elements that are then *deployed and integrated* into the system, a process that is increasingly automated (Rodriguez et al., 2017).

In contrast, a *self-evolving computing system* maintains a *self-representation* that includes runtime models of the computing system and its goals (self-awareness), and the environment in which the system operates (context-awareness). An *evolutionary learning engine* uses the self-representation to autonomously evolve the architecture of the computing system, in response to *unanticipated changes* that occur throughout the system's lifetime, i.e., new goals or new constraints that appear, or anomalies identified during operation. To that end, the evolution engine runs experiments in a sandbox evolving the system model until it satisfies the new conditions. During this process, the engine can integrate new

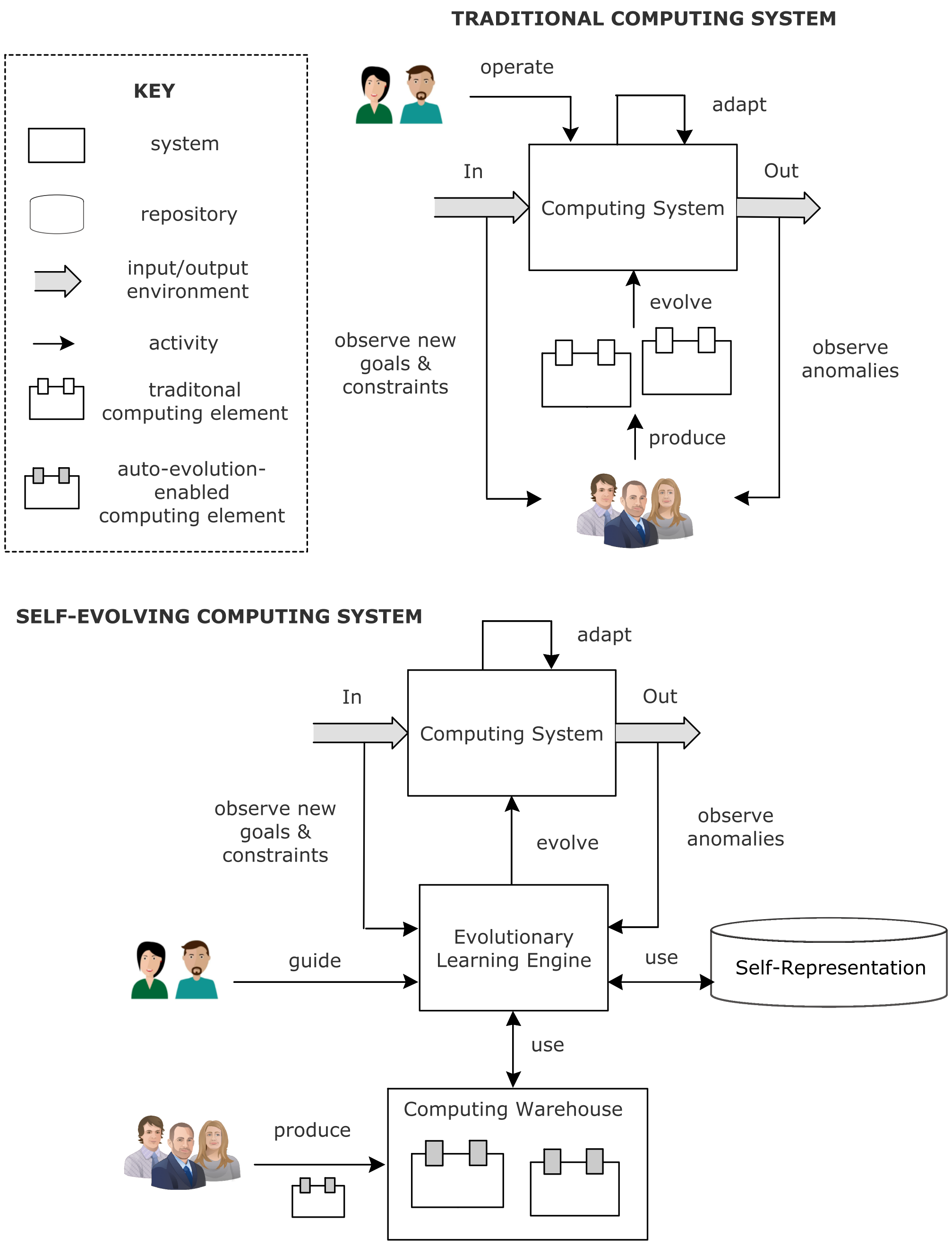

Fig. 1. From traditional computing systems to self-evolving computing systems.

computing elements from *computing warehouses* as needed. These *auto-evolution-enabled* computing elements provide specifications and procedures that enable an evolutionary engine to incorporate these elements autonomously. As shown in Figure 1, a self-evolving computing system takes the human out of the loop of the evolution process. Humans only produce new auto-evolution-enabled computing elements that are readily available for self-evolving computing systems via computing warehouses. Yet, humans may be involved to provide *guidance* to the system, for instance to set constraints on the behavior of the system or express preference of one configuration over another during evolution.

Self-evolving computing systems focus on the evolution aspects of computing systems within the newly proposed paradigm of "lifelong computing" (Weyns, Bäck, Vidal, Yao, & Bel bachir, 2021a). Self-evolving computing systems also resemble similarities with the idea of "self-growing software" proposed by Tamai (2019) as the next paradigm shift in software engineering.

The remainder of this paper starts with a discussion of a selection of key approaches to deal with change and points out why a novel foundation is required (Section 2). Then we introduce an illustrative example (Section 3). We outline a conceptual architecture for self-evolving computing systems (Section 4) and illustrate the architecture for the example. To conclude, we highlight key research challenges for realizing the vision of self-evolving computing systems and we suggest starting points to tackle them (Section 5).

2. State of the Art

Already in the early 2000s, IBM pointed to the manageability problems caused by the growing complexity of computing systems (IBM, 2003). In response, they launched the autonomic computing initiative that was centered on enabling computing systems to manage themselves based on high-level goals, similar to the autonomic nervous system of the human body. Autonomic computing primarily focuses on automating tasks of running computing systems that are traditionally done by operators. Hence, the target of autonomic computing is the operational domain of computing systems. Self-evolution on the other hand targets the *autonomous evolution* of computing systems, hence the target is a change of the operational domain. Self-evolution aims to enable computing systems dealing with unanticipated change by evolving autonomously.

In this section, we summarize the state of the art in two key fields that tackle the problem of managing change of computing systems from two complementary points of view: smart systems and software evolution. Based on this analysis, we motivate the need for self-evolving systems.

2.1. Smart Systems

Tavcar & Horváth (2019) surveyed smart computing systems, with an emphasis on cyber-physical systems. The authors distinguish four levels of smartness mapping to increasingly challenging types of changes to be tackled by the systems, ranging from no changes to unknown changes. Smartness then refers to the capability level of computing systems to handle these types of changes through reasoning, learning, adapting, and evolving. Weyns et al. (2022) extended the notion of smart to "smarter" referring to both computing systems and their engineering processes that continuously adapt and evolve through a perpetual process that continuously improves their capabilities and utility to deal with the uncertainties and new data they face throughout their lifetime. Bures, Weyns, Schmerl, Tovar, Boden, Gabor, Gerostathopoulos, Gupta, Kang, Knauss, Patel, Rashid, Ruchkin, Sukkerd, & Tsigkanos (2017) emphasized that smartness of computing systems enable them to deal with dynamics and uncertainty in the environment, and external threats. The authors highlight that smartness of computing systems is primarily implemented through the software leveraging principles from self-adaptation. Musil, Musil, Weyns, Bures, Muccini, & Sharaf (2017) presented a set of architectural patterns to realize self-adaptation across the software stack of cyber-physical systems.

A classic field of study on smartness is autonomous systems (or intelligent autonomous systems) (Paulovich, Oliveira, & Oliveira, 2018; Tzafestas, 2012). Autonomous systems mimic human (or animal) intelligence, in order to operate independently of direct human supervision. An important sub-field of autonomous systems is multi-agent systems (Wooldrige, 2009) that studies the operation and coordination of autonomous agents that aim at solving problems that go beyond the capabilities of single agents. Different authors have presented patterns that document problem-solution pairs for engineering multi-agent systems (Dastani & Testerink, 2016; Marks, Muller, Vogeli, Jung, Jazdi, & Weyrich, 2018; Schelfthout, Coninx, Helleboogh, Holvoet, Steegmans, & Weyns, 2002). Juziuk,Weyns, & Holvoet (2014) presented a systematic literature overview classifying patterns based on focus, granularity, level of abstraction, and source of inspiration. The field of human-robot teams (Musi & Hirche, 2016) studies collaboration of humans and robots exploiting their complementary skill sets. Another promising key field enabling the realization of smartness is digital twins (Tao, Zhang, Liu, & Nee, 2019). Digital twins are characterized by the seamless integration between the cyber and physical spaces. Digital twins have been successfully applied in product design, production, prognostics and health management, among other fields. Gentelligent systems (Denkena & Morke, 2017) integrate sensing components throughout the production supply chain to improve efficiency, flexibility, and product quality. Recently, the interest in autonomous systems has been expanding significantly with high-profile applications, such as smart robotics (Industry 4.0 driven by the Internet of Things) and smart transportation. For instance, Jazdi (2014) stressed the need to equip Industry 4.0 systems with smart actuators, sensors, and telecommunication technologies, providing these systems access to the higher-level processes and services. Weyns, Iftikhar, Hughes, & Matthys (2018) presented MARTAS that automates the management of Internet-of-Things leveraging statistical model checking at runtime to ensure the system goals under uncertainty. Yu & Xue (2016) referred to smartness of the electricity grid as the integration of information and communication technology with other advanced technologies that enable electric energy generation, transmission, distribution, and usage to be more efficient, effective, economical, and environmentally sustainable. Koutsoukos, Karsai, Laszka, Neema, Potteiger, Volgyesi, Vorobeychik, & Sztipanovits (2018) investigated smart transportation systems using a modeling and simulation environment. Smartness in this context relates to the ability of a system to deal with attacker-defender behavior, including vulnerability analysis to traffic signal tampering, resilient sensor selection for forecasting traffic flow, and resilient traffic signal control in the presence of denial-of-service attacks.

Another classic field of smart systems is self-adaptation. Simultaneous with industrial initiatives, such as autonomic computing (Kephart & Chess, 2003) mentioned above, researchers studied the abilities of computing systems to handle change autonomously (Garlan et al., 2004; Oreizy, Gorlick, Taylor, Heimhigner, Johnson, Medvidovic, Quilici, Rosenblum, & Wolf, 1999). Self-adaptation is based on the principles of feedback computing (Kramer & Magee, 2007; Oreizy et al., 1999; Salehie & Tahvildari, 2009; Weyns, 2021). Over the past two decades, extensive efforts have been put in devising fundamental principles of self-adaptation as well as techniques and methods to engineer self-adaptive systems (Weyns, 2019). Whereas the initial focus was on automating operator tasks based on high-level goals (Garlan et al., 2004; Kephart & Chess, 2003), later research shifted towards taming uncertainties that computing systems face during operation and that are difficult to anticipate before deployment (Calinescu, Weyns, Gerasimou, Iftikhar, Habli, & Kelly, 2018; Cheng et al., 2009a; Moreno, Ca´mara, Garlan, & Schmerl, 2015). This view introduces a perspective that blends system engineering and system operation (Baresi & Ghezzi, 2010; Chen, Bahsoon, & Yao, 2018a; Chen, Li, Bahsoon, & Yao, 2018c; Weyns, Bencomo, Calinescu, Camara, Ghezzi, Grassi, Grunske, Inverardi, Jezequel, Malek, Mirandola, Mori, & Tamburrelli, 2017). Central to any self-adaptive systems are runtime models (Blair, Bencomo, & France, 2009) that provide the system with self-awareness (self-representation and representation of goals) and context-awareness (representation of the environment) (Chen, Bahsoon, & Yao, 2020; Elhabbash, Salama, Bahsoon, & Tino, 2019; Weyns, Malek, & Andersson, 2010). These models are updated at runtime tracking uncertainties (Calinescu, Mi randola, Perez-Palacin, & Weyns, 2020; Esfahani & Malek, 2013; Mahdavi-Hezavehi, Avgeriou, & Weyns, 2017;Weyns, Caporuscio, Vogel, & Kurti, 2015) and then used to analyze the situation and decide when and how to adapt the system to maintain its goals, or gracefully degrade if needed.

2.2. Software Evolution

Evolution is a natural part of the life cycle of software systems that traditionally occurs in incremental development in response to changes in the environment, purpose, or use of the software system (Reussner et al., 2019). Buckley, Mens, Zenger, Rashid, & Kniesel (2005) presented a taxonomy for software evolution with four dimensions of system change: temporal properties (i.e., when do changes happen), objects of change (i.e., where in the system do we make changes), system properties (i.e., what is changed), and change support (i.e., how is the system changed). Earlier, Chapin, Hale, Kham, Ramil, & Tan (2001) identified two other core dimensions: motivations (i.e., why are the changes done) and roles (i.e., who is doing system changes). The ISO/IEC standard for software maintenance[2] distinguish four types of software changes: corrective (bug fixing dealing with errors), adaptive (environment and requirement changes), perfective (optimizing or refactoring the system), and preventive modifications (preventing problems).

During the past decades, the traditional view of software that evolves through periodic releases has been replaced by continuous evolution of software (Rodriguez et al., 2017). Software organizations today develop, release, and learn from software in rapid parallel cycles (typically from hours to a few weeks). This approach is commonly referred as continuous deployment (CD) (Järvinen, Huomo, Mikkonen, & Tyrväinen, 2014). CD is based on the principles of agile development (Dingsyr, Nerur, Balijepally, & Moe, 2012) and DevOps (Mishra & Otaiwi, 2020) that aim at increasing the deployment speed and quality of systems. CD leverages on continuous integration (CI) (Meyer, 2014) that automates tasks such as compiling code, running tests, and building deployment packages. Among the benefits of CI/CD are rapid innovation, shorter time-to-market, increased customer satisfaction, continuous feedback, and improved developer productivity. Yet, an important concern of current practice in software maintenance is (intentional or unintentional) technical debt, i.e., longer-term negative effects on systems that result from sub-optimal decisions (Li, Avgeriou, & Liang, 2015), in particular in the context of agile development. Furthermore, researchers have argued that the current level of automation needs to be enhanced (Rodriguez et al., 2017), and last but not least, to develop sustainable computing systems, we need sustainable software development processes (Andersson et al., 2013; Dick & Naumann, 2010; Georgiou, Rizou, & Spinellis, 2019; Naumann, Dick, Kern, & Johann, 2011; Weyns et al., 2022; Weyns & Iftikhar, 2022).

With the increasing exposure of computing systems to change, the volumes of data they need to process, and the seamless integration of humans in the loop (Musil, Musil, Weyns, & Biffl, 2015; Selic, 2020; Sztipanovits, Koutsoukos, Karsai, Kottenstette, Antsaklis, Gupta, Goodwine, Baras, & Wang, 2012; Zeng, Yang, Lin, Ning, & Ma, 2020), computing systems face uncertainties that are difficult or even impossible to predict before deployment. Hence, engineers may not be able to obtain sufficient knowledge to make all design decisions before the system is deployed. This calls for postponing design decisions until after deployment when the required knowledge becomes available. The design decisions are then enacted through continuous adaptation and evolution (Baresi & Ghezzi, 2010; Weyns, 2021). To that end, a number of important building blocks have been studied. We highlight two: anomaly detection and lifelong learning.

Anomaly detection (or outlier or novelty detection) aims at identifying data instances that significantly deviate from the majority of data instances in a data set (Grubbs, 1969). Anomaly detection has been used in a variety of domains, e.g., intrusion detection, fault prevention, defect detection, and unexpected flow detection. A plethora of methods have been developed (Boukerche, Zheng, & Alfandi, 2020; Chen, Tino, Rodan, & Yao, 2014), including proximity-based approaches that rely on relations between nearby data points, projection techniques that convert data into a space with reduced

[2]International Organization for Standardization. ISO/IEC 14764. 2014. URL: www.iso.org/standard/39064.html

dimensionality to improve outlier detection, outlier detection for multi-dimensional data such as recursive binning and re-projection, windowing for online time series that incrementally builds and updates models with new data, learning model spaces for fault diagnosis, and deep learning anomaly detection, such as deep neural network auto-encoders. Yet, dealing with highly complex data remains an open problem. Anomaly detection mechanisms enable a computing system to autonomously identify behavior at the boundaries or outside its operational domain, providing a basis building block for the realization of self-evolving systems.

Lifelong learning (or continual learning) refers to the ability of a system to continually accommodate new knowledge to learn new tasks that were not predefined (Thrun & Mitchell, 1995). Different approaches for lifelong learning have been developed relying on supervised, unsupervised, and reinforcement learning (Chen & Liu, 2018), and recently lifelong learning based on neural networks has gaining increasing interest (Parisi, Kemker, Part, Kanan, & Wermter, 2019). A key challenge for lifelong learning is dealing with catastrophic forgetting that refers to the loss of previous learning while learning new information; this may lead to failures for systems operating in real-world environments (Hasselmo, 2017). Different approaches have been proposed to deal with this problem, such as dynamic allocating new neurons or network layers to accommodate novel knowledge, and using complementary learning networks with experience replay, yet more research is needed apply these techniques to real-world systems (Parisi et al., 2019). Lifelong learning techniques provide another basic block for the realization of self-evolving computing systems.

2.3. Why Self-Evolving Computing Systems?

When we look at the current landscape of research, we can observe two principle lines of work. The first line studies the application of smart techniques enabling systems to deal with changes autonomously during operation. The second line studies the evolution of computing systems with an emphasizes on tools for automating the deployment and integration of computing elements. We advocate that a key underlying problem with these existing approaches is the lack of an integrated perspective on handling change—anticipated and unanticipated—in an autonomous manner. Compared to traditional (or conventional) systems, smart systems are equipped with capabilities to handle a variety of changes autonomously. Yet, the target domain of smart systems is in essence their operational domain, that is, their capabilities are confined to what they have been built for. The aim of software evolution lays essentially in revising or extending the operational domain. While several steps in the process of software evolution have been automated in the past decades, the actual evolution of the software remains in essence a human-driven activity. Autonomous and self-adaptive systems have expanded the operational domain of computing systems substantially, enabling them to deal with changes during operation to enhance their efficiency and being most robust, yet the scope remains bounded to anticipated changes. Anomaly detection mechanisms allow identifying deviations from expected behaviors, and lifelong learning enables learning-based systems dealing with new tasks during operation. Yet, besides their current limitations for real-world problems, these techniques offer only basic blocks to realize a true integration of continuous adaptation and evolution aiming at mitigating the effects of uncertainty that spans both anticipated and unanticipated change. To tackle the challenges of continuous change, anticipated and unanticipated, a new integrated perspective for the engineering and operation of future computing systems is needed. Self-evolving computing systems aim to offer such a perspective.

3. Future Smart City Mobility Scenario

We illustrate the need for self-evolving computing systems with an example of a future smart city mobility scenario. A research study called “New autoMobility” (Grötker, 2015) highlighted how automated and networked vehicles and trains can be usefully integrated into a user-friendly, efficient and sustainable mobility system in the future. Such a system would consist of mobility hubs, car sharing and self-parking vehicles, and autonomous trains. Flexible, time-and-space-dependent mobility pricing will ensure more evenly distributed usage of mobility resources and prevent traffic gridlock. Vehicles will be able to warn each other (directly or indirectly) in dangerous situations creating a cooperative mixed traffic. Such intelligent, networked transport protects the environment and the climate and improves quality of life.

Establishing automated mobility requires a phased introduction and continuous evolution of a mobility platform to align with a variety of changes. This poses difficult often conflicting challenges, spanning business, technical, social, and legal aspects. For example, the introduction of automated traffic will happen only gradually, so initially automated and conventionally controlled vehicles will co-exist. Depending on local conditions, there may be a need to manage the level of pollution in areas with more intensive traffic of conventionally controlled vehicles. This may require the need for tracking the levels of pollution in these areas and take measures when needed. Such measures may range from temporally redirecting conventional vehicles in certain areas using smart traffic boards, up to increasing prices for polluting vehicles for instance to part in sensitive areas. However, with changing usage profiles, transitions to automated mobility, and novel technological advances, these provisions will need to evolve.

Central to the functionality and safety of mobility will be the collection and processing of data and information from various sources. Managing this data requires a suitable framework that creates connectivity between vehicles, the infrastructure, and traffic management systems, ensuring safety while respecting the personal interests and privacy concerns of the users at any time. Tackling these challenges and balancing the trade-offs between the various needs will require an integrated computing system that is capable to operate, adapt, and evolve autonomously throughout its lifetime in a continuously changing environment. We illustrate how a self-evolving computing system could offer such a unique solution.

4. Conceptual Architecture for Self-Evolving Computing Systems

In this section, we present a conceptual architecture for self-evolving computing systems. To deal with the continuous changes a self-evolving computing system faces throughout its lifetime, we outline five complementary requirements for a self-evolving computing system. These requirements naturally target the ability of self-evolving systems to deal with anticipated change (1), to discover unanticipated changes and evolve autonomously (2-4), and to integrate humans in the loop when needed (5).

(1) A self-evolving computing system should be able to handle vast amounts of data and realize its goals under changing but anticipated conditions;
(2) A self-evolving computing system should be able to discover and integrate new computing elements autonomously;
(3) A self-evolving computing system should be able to autonomously detect unanticipated conditions, i.e., learn conditions outside its operational domain, including anomalies, new goals and constraints;
(4) A self-evolving computing system should be self-aware and context-aware enabling it to autonomously evolving its architecture to realise its goals;
(5) Depending on the domain at hand, some activities of a self-evolving computing system may be supported by humans.

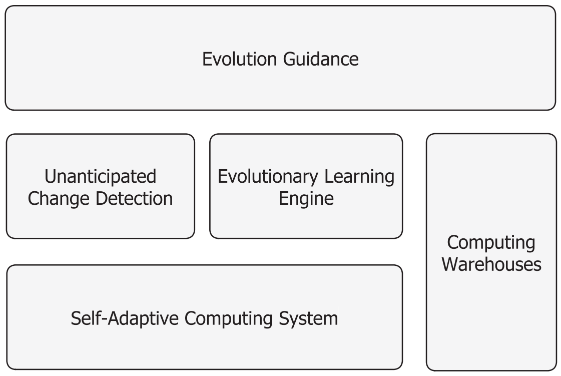
Evolution Guidance
Unanticipated Change Detection
Evolutionary Learning Engine
Computing Warehouses
Self-Adaptive Computing System

Fig. 2. Conceptual architecture for self-evolving computing systems with the different building blocks.

Requirement (1) is a basic requirement for systems that need to achieve their goals while dealing with huge amounts of data and operating under uncertainty. Requirements (2) to (4) are key for enabling systems to evolve autonomously when encountering unanticipated changes. As for requirement (5), support for human guidance is particularly important: (i) in domains with critical goals where humans will have the ultimate control over the system by setting boundaries on the system behavior, ensuring the trustworthiness of the system, (ii) for systems that require human interaction to set high level goals or express preferences among possible options generated by the system (in contrast to performing standard operating activities).

To achieve these requirements, we propose a conceptual architecture for self-evolving computing systems as shown in Figure 2. We explain the different building blocks and illustrate each of them with examples of the future mobility scenario. Starting points to realize the building blocks are explained in Section 5.

Self-Adaptive Computing System. As a basis, a self-evolving computing system comprises an *self-adaptive computing system* that integrates regular computing elements and learning algorithms, enabling it to handle a vast amount of data and realize the goals of its users. Furthermore, the self-adaptive computing system is equipped with smart techniques enabling it to deal with changes within its operational domain, i.e., changing operation conditions and uncertainties that can be managed by adapting the running architectural configuration of the self-adaptive computing system, without the need for updates or the integration of new computing elements or learning algorithms. As such, a self-adaptive computing system realizes requirement (1). To account for unanticipated changes that requires evolution (see evolutionary learning engine below), the self-adaptive computing system should support automatic updates of its running architecture.

Figure 3 illustrates a self-evolving computing system for the smart city mobility scenario. We focus here on the self-adaptive computing system (lower box left) that comprises the smart city area with a mobility hub that connects different modes of public transport, conventional cars and smart vehicles, pedestrians, and a variety of sensors (cameras, smart boards, parking sensors, etc.) that measure the density of traffic, occupation of automated trains, usage of parking lots, movements of pedestrians, etc. The data is collected by a *mobility tracking platform* and stored and updated in a *mobility data repository*. The data is continuously processed by a *learning service center* that learns and predicts relevant system parameters, such as mobility distribution, traffic safety, etc. These parameters together with other data obtained from the Cloud (e.g., weather forecasts) are then used by the *adaptation manager* that continuously optimizes the different objectives of the mobility system and their trade-offs, using the *mobility control platform*. For instance, when a camera detects an increase of passengers of smart

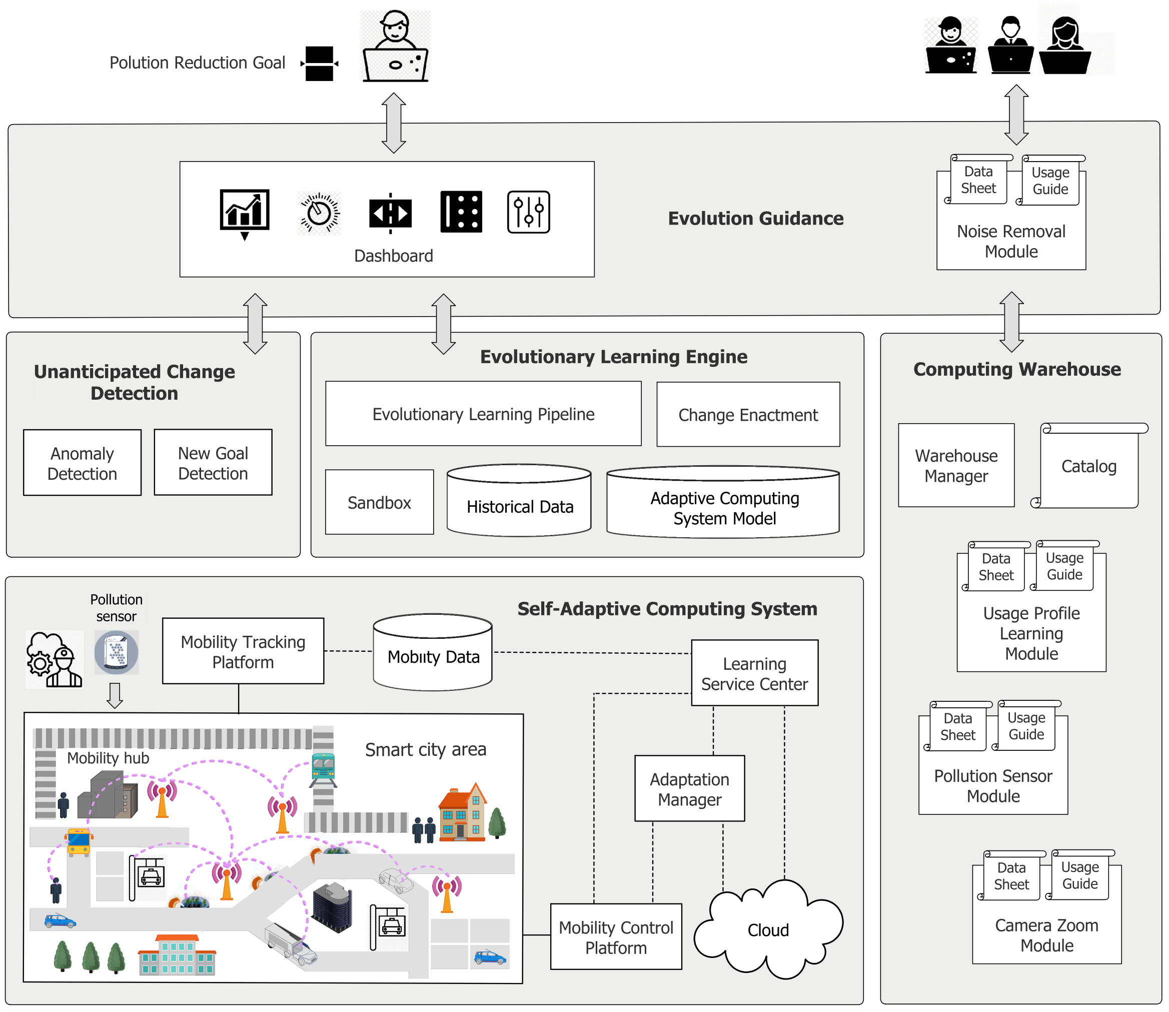

Fig. 3. Illustration of the conceptual architecture for a smart city mobility scenario

vehicles for a particular trajectory, the frequency of these transports may be increased dynamically and the ticket price may be adjusted temporally.

Computing Warehouses. Self-evolving computing systems are supported by *computing warehouses* that offer new computing elements, realizing requirement (2). Computing warehouses leverage the principles of off-the-shelf components and services, open source software, and open data. Computing warehouses can be operated directly by producers of new auto-evolution-enabled computing elements or indirectly via a broker. We refer to the elements provided by computing warehouses as *auto-evolution-enabled computing elements*; examples are a module that offers improved or new functionality, a connector to connect with and use a new external service, a template of new learning algorithm, a repository of data, etc. It is important that self-evolving computing systems can incorporate auto-evolution-enabled computing elements autonomously during operation. To that end, each auto-evolution-enabled computing element is equipped with a *data sheet* that specifies its functions, properties, usage requirements, etc., and a *usage guide* that specifies the procedures that need to be followed for using the element. These specifications require both a well-defined syntax and an ontology that defines the semantics of the properties and usage of the elements. Depending on the requirements, new auto-evolution-enabled computing elements may require certification before making them available in a warehouse. All interactions with the computing warehouse happen via a *warehouse manager*. Clients can search the available elements via a *catalog* that lists the elements with their data sheets and usage guides; using a computing element may be subject to a contract.

Figure 3 shows a few examples of new auto-evolution-enabled computing elements for the smart mobile city scenario (box right). The *camera zoom module* provides the software that is required to activate and use zoom lenses on cameras. The *usage profile learning module* offers new learning models of users of a smart city mobility system, possibly derived from studies. The *pollution sensor module* offers the software to start using sensors that measure particular pollution parameters of the environment in the city.

Unanticipated Change Detection. A key feature of self-evolving computing systems is their ability to detect unanticipated changes, i.e., changes that cannot be handled by the build-in learning and adaptation mechanisms of the self-adaptive computing system, realising requirement (3). Such unanticipated changes can be triggered either by an *anomaly* the self-adaptive computing system encounters, or by new goals that are added to the system. When encountering such an event, unanticipated change detection will trigger the evolutionary self-learning engine to start an evolution of the self-adaptive computing systems (see below).

As an example, assume that anomaly detection (middle box left in Figure 3) discovers that the lenses of cameras are dirty resulting in poor quality images. To deal with this problem, a *noise removal learning module* is added to the computing warehouse that offers a new learning algorithm, for instance a convolutional neural network to handle noisy images. This module will then be used by the evolutionary learning engine for evolving the architecture configuration of the self-adaptive computing system (further explained below). As another example, consider the introduction of a new goal to reduce pollution in the smart city area caused by mobility. To deal with this new goal, an operator adds a new *pollution reduction goal* to the evolutionary learning engine via the dashboard. This will trigger the evolutionary learning engine to start an evolution of the self-adaptive computing system taking into account this new goal (further explained below).

Evolutionary Self-learning Engine. At the heart of a self-evolving computing system is an *evolutionary learning engine* that autonomously evolves the self-adaptive computing system to handle any unanticipated changes that cannot be handled by the build-in learning and adaptation mechanisms, realizing requirement (4). When anticipated change detection discovers an anomaly or when a new goal is added to the system, the evolutionary learning engine starts to evolve its internal *model of the self-adaptive computing system*. This runtime model contains an up-to-date representation of the architecture of the self-adaptive computing system along with its goals (self-awareness), and relevant parts of the environment (context-awareness). The evolution of the model is conducted by an *evolutionary learning pipeline* that evolves the architectural configuration of the self-adaptive computing system to obtain its goals. During this process, the engine may integrate new auto-evolution-enabled computing elements provided by computing warehouses as needed. To evolve the system architecture, the engine runs experiments, executing different subsequent variants of the evolved model in a sandbox. Using suitable metrics for assessing the performance of the evolving architectural models of the self-adaptive computing system in each evolutionary step, the engine will optimize the self-adaptive computing system model, resulting in a novel architecture that mitigates the unanticipated change that triggered the evolution. During the experiments, the engine may exploit *historical data*, for instance to train a learning module, and experimental results may be stored for reuse later. *Change enactment* will then replace the running architecture of the self-adaptive computing system with the novel architecture.

As an example, when discovering that the lenses of cameras are dirty (continuing the example above), the evolutionary learning engine (middle box in Figure 3) searches the computing warehouse for a solution. Based on the shared ontology, the engine identifies the new noise removal learning module. The evolutionary learning engine then runs online experiments in the *sandbox*, evolving the model of the current architecture of the self-adaptive computing system and integrating the new noise removal learning module. The engine will use the resolution and quality improvements of images as performance metrics. During this process, the engine may exploit historical data to accelerate the evolution process, and particular experimental results may be stored for later usage. Once the novel architecture is identified that satisfies the system goals, the current configuration will be evolved through change enactment.

As another example, when the new pollution reduction goal is added to the system (continuing the other example above), the evolutionary learning engine will search in the catalog of the computing warehouse and find the (newly added) pollution sensor module. Based on the usage guidance provided by this module a set of new *pollution sensors* will be activated in the smart city area (possibly involving a field worker). The evolutionary self-learning pipeline will then evolve the architecture of the self-adaptive computing system by extending the mobility tracking platform with functionality to track air pollution and set the configurations of the sensors via the mobility control platform (both derived from the pollution sensor module). Furthermore, the new goal will be added to the adaptation manager. Finally, the learning module will be enhanced to take into account the data of the mobility data module produced by the pollution sensors. To configure the learning model, the engine may use historical data collected by the system. Once the new architecture is configured, it can be deployed via change enactment enabling the smart city mobility system to reduce the pollution by adjusting its settings, e.g. adapting conventional traffic via smart traffic boards.

Evolution Guidance. Depending on the domain at hand, human experts may be involved to *guide the evolution* of a self-evolving computing system, realizing requirement (5). Evolution guidance can range from a basic dashboard that shows key performance indicators of a self-evolving computing system and offers “knobs” allowing operators to upload new computing elements, add new goals or define constraints on the behavior of the system to ensure its trustworthiness, up to full-fledged embodied AI that exploits intelligent user interfaces enabling operators to guide the evolution process of self-evolving computing systems interactively (Kephart, Dibia, Ellis, Srivas tava, Talamadupula, & Dholakia, 2019). New goals or constraints may refer to various concerns of users, such as performance, safety, privacy, energy consumption, environmental protection, or ethics. Evolution guidance may include the option for operators to provide feedback about discovered anomalies or give advice on architecture evolution at the evolutionary learning engine, among others.

For instance, in the smart city mobility scenario, see Figure 3 (box at the top), evolution guidance enables software developers to add new auto-evolution-enabled modules to the computing warehouse, such as a new learning module for noise removal. Evolution guidance also offers an interactive dashboard enabling an operator to support the evolutionary learning engine with identifying new software architectures of the computing-learning system. For instance, the operator may suggest (possibly new) quantitative and qualitative criteria (goals) to guide a evolutionary pipeline in identifying new architectural configurations. The feedback of the operator may be incorporated into the fitness function allowing the learning pipeline to distinguish between promising and poor architectural configurations when evolving the model of the self-adaptive computing system, enhancing its performance.

5. Research Challenges Ahead

To conclude, we summarize the novelty of self-evolving computing, highlight key challenges to realize the vision of self-evolving computing systems, and provide starting points to tackle them.

Smart approaches have demonstrated their value for dealing with changes *within the operational domain* of computing system that are composed of regular computing elements. Self-evolving computing extends this to the operational domain of computing systems that integrate regular computing elements with learning algorithms, enabling these systems to deal with a vast amount of highly complex data. Currently, we rely on software evolution to deal with *outside the operational domain*, i.e., changes that were not anticipated when the system was built and deployed. The evolution of software systems is currently still a human-driven process that is supported by tools that automate the continuous integration and deployment of new computing elements. Lifelong learning provides the means to deal with new tasks during operation, yet, this evolution targets learning algorithms. Self-evolving computing on the other hand exploits computing warehouses, enabling self-evolving computing systems to evolve autonomously, thereby selecting and integrating new computing elements autonomously during operation based on the needs at hand. Optionally, humans can offer support to self-evolving computing systems, for instance, for setting goals on performance, safety, privacy, etc., and providing guidance to support the evolutionary learning process if needed.

We motivated and described how self-evolving computing enables dealing with the lasting problem of how to engineer long running computing systems that can autonomously adapt and evolve to deal with ever changing conditions, anticipated and unanticipated. Yet, realizing the vision of self-evolving computing, raises fundamental challenges. We list six key achievements that are required to tackle these challenges:

(1) A novel overarching modeling approach for the design of self-evolving computing systems. Contrary to traditional software architecture design languages (Muccini & Vaidhyanathan, 2021), a new modeling approach is required that should provide first-class support for specifying heterogenous computing systems that integrate computing and learning elements, as well as the different types of building blocks of self-evolving computing systems. This modeling approach will enable a designer to analyse the compliance of the model of a self-evolving computing system with its high-level goals.

(2) The definition of standardized representations and interfaces of auto-evolution-enabled computing elements (regular and learning elements) that can be seamlessly integrated by self-evolving computing systems. Contrary to existing component-based modeling approaches, see e.g., (Bruneton, Coupaye, Leclercq, Quema, & Stefani, 2004), auto-evolution-enabled computing elements require two types of meta data: (i) meta data that enables self-evolving computing systems to characterize elements and select an element as needed, and (ii) meta data to incorporate a selected element autonomously. The first type of meta data is similar to a "data sheet," while the second type is similar to a "usage guide." Enabling self-evolving computing systems to reason about and integrate auto-evolution-enabled computing elements require both a well-defined (standardized) syntax and a shared ontology.

(3) Novel methods and algorithms for realizing self-adaptation of heterogeneous computing systems that need to deal with conflicting goals and operate under uncertainty and resource constraints. An interesting approach to tackle this challenge is the use of dynamic, preference-based, multi-objective, on-line optimization, leveraging state-of-the-art knee-point identification (Yu, Jin, & Olhofer, 2020), and preference-based (Palar, Yang, Shimoyama, Emmerich, & Bäck, 2018), and on-line optimization (Chen, Li, & Yao, 2018b). Here the Pareto-frontier becomes a moving target, while the objectives can change when an architecture evolution is applied by the self-evolving computing system.

(4) A novel family of anomaly and novelty discovering methods for complex high-dimensional data relying on unsupervised learning. One approach to tackle this challenge is to model the data as a

union of low-dimensional manifolds (You, Robinson, & Vidal, 2017). Anomalies are then data points that do not lie in any manifold, i.e., outliers, while novelties are data points that belong to a new manifold, e.g., a new class. The challenge here will be to identify nonlinear manifolds that change over time. Additionally, the solution should be able to deal with multi-modal online data streams, e.g., leveraging temporal convolutional autoencoders (Thill, Konen, Wang, & Bac̆k, 2021).

(5) A novel evolutionary self-learning pipeline for evolving heterogeneous computing systems to deal with unanticipated changes (anomalies, novelties, new goals). Core to such a solution will be: up-to-date architectural models of the underlying heterogenous system with its goals and constraints, and the context in which the system operates. These models should account for the evolution of the system. Evolving the current software architecture requires suitable architectural variation operators that comply with the syntactical and semantical constraints of the evolving architecture. One approach to tackle this is using a (1,λ) algorithm (Bäck, Foussette, & Krause, 2013) that selects the best "offspring" and iterates the evolution through simulation in a sandbox. New evaluation functions with guarantees will be required, e.g., leveraging statistical model checking of the candidate architectures, and (ii) preference-based, multi-objective optimization providing approximations of Pareto optimality.

(6) Novel notations and mechanisms that enable system operators to add new goals and interact with the evolutionary self-learning pipeline. Changing goals is an important trigger for evolving computing systems. This requires a dashboard for humans to interact with the system and modify its goals. Unlike existing goal models (e.g., Cheng, Sawyer, Bencomo, & Whittle (2009b)), self-evolving computing systems require models that evolve dynamically. Goals may be provided with meta data that refers to elements of computing warehouses (e.g., a goal for a new modality of traffic may have meta data about sensors and software to track that traffic). The models should provide mechanisms that automatically translate the changes of the goals to a format that can act as a trigger to evolution. A self-evolving system may be equipped with mechanisms that enable the system to communicate the options for evolution and ask the human to advise on the selection if needed. The dashboard may supports this type of interaction. For instance, the system may show a subset of candidate architectural configurations along with a qualification of the options. The human may then select one of the options to continue the evolution process, leveraging for example de Winter, van Stein, & Bäck (2021).

An additional open challenge is how to handle the need for dynamic resource management. While the warehouse may to some degree deal with resource provision and management, the acquisition of hardware and other resources that are needed to support self-evolution may require dedicated support.

Addressing these challenges requires the combined expertise in a variety of areas: dynamic software architectures and scalable and trustworthy approaches for self-adaptation (to deal with the challenges of adaptation of heterogeneous computing systems), unsupervised learning and runtime goal models (to deal with the challenges of unanticipated change detection), self-awareness, dynamic learning architectures, and evolutionary learning mechanisms (to deal with the challenges of evolutionary learning), and software engineering (to deal with the challenges of computing warehouses and evolution guidance). Only the synergy between these specializations can adequately yield solutions to realize the vision of self-evolving computing.

References

Andersson, J., Baresi, L., Bencomo, N., de Lemos, R., Gorla, A., Inverardi, P., & Vogel, T. (2013). Software Engineering Processes for Self-Adaptive Systems. *Springer*, 51-75. https://doi.org/10.1007/978-3-642-35813-5_3

Bäck, T., Foussette, C., & Krause, P. (2013). Contemporary Evolution Strategies. *Natural Computing Series, Springer*.

Baresi, L., & Ghezzi, C. (2010). The Disappearing Boundary between Development-Time and Run-Time. In Future of Software Engineering Research. *ACM*, 1722. https://doi.org/10.1145/1882362.1882367

Bennett, K., & Rajlich, V. (2000). Software Maintenance and Evolution: A Roadmap. In *Conference on The Future of Software Engineering (Limerick, Ireland) (ICSE '00). Association for Computing Machinery, New York, NY, USA*, 7387. https://doi.org/10.1145/336512.336534

Bernardo, M., & Hillston, J. (Eds.). (2007). Formal Methods for Performance Evaluation, *7th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2007, Bertinoro, Italy, May 28-June 2, 2007, Advanced Lectures. Lecture Notes in Computer Science, Vol. 4486. Springer*.

Blair, G., Bencomo, N., & France, R.B. (2009). Models@ run.time. *Computer*, *42*(10), 22-27. https://doi.org/10.1109/MC.2009.326

Boukerche, A., Zheng, L., & Alfandi, O. (2020). Outlier Detection: Methods, Models, and Classification. *ACM Comput Surv*, *53*, 3, Article 55 (June 2020), pp. 37. https://doi.org/10.1145/3381028

Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., & Stefani, J.-B. (2004). An Open Component Model and Its Support in Java. In *Component-Based Software Engineering. Springer*, 7-22.

Buckley, J., Mens, T., Zenger, M., Rashid, A., & Kniesel. G.r. (2005). Towards a Taxonomy of Software Change: Research Articles. *Journal on Software Maintenance and Evolution*, *17*(5) (Sept. 2005), 309332.

Bures, T., Weyns, D., Schmerl, B., Tovar, E., Boden, E., Gabor, T., Gerostathopoulos, I., Gupta, P., Kang, E., Knauss, A., Patel, P., Rashid, A., Ruchkin, I., Sukkerd, R., & Tsigkanos, C. (2017). Software Engineering for Smart Cyber-Physical Systems: Challenges and Promising Solutions. *SIGSOFT Software Engineering Notes*, *42*(2), 1924. https://doi.org/10.1145/3089649.3089656

Calinescu, R., Mirandola, R., Perez-Palacin, D., & Weyns, D. (2020). Understanding Uncertainty in Self adaptive Systems. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems*, 242-251. https://doi.org/10.1109/ACSOS49614.2020.00047

Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., & Kelly, T. (2018). Engineering Trustworthy Self-Adaptive Software with Dynamic Assurance Cases. *IEEE Transactions on Software Engineering*, *44*(11), 1039-1069. https://doi.org/10.1109/TSE.2017.2738640

Camazine, S., Deneubourg, J.-L., Franks, N., Sneyd, J., Theraulas, G., & Bonabeau, E. (2003). Organization in Biological Systems. Princeton Studies in Complexity, USA.

Chapin, N., Hale, J., Kham, K., Ramil, J., & Tan, W. (2001). Types of Software Evolution and Software Maintenance. *Journal of Software Maintenance*, *13*(1) (2001), 330.

Chen, H., Tino, P., Rodan, A., & Yao, X. (2014). Learning in the Model Space for Cognitive Fault Diagnosis. *IEEE Transactions on Neural Networks and Learning Systems*, *25*(1), 124-136. https://doi.org/10.1109/TNNLS.2013.2256797

Chen, R., Li, K. & Yao, X. (2018b). Dynamic Multiobjectives Optimization With a Changing Number of Objectives. *IEEE Transactions on Evolutionary Computation*, *22*(1), 157-171. https://doi.org/10.1109/TEVC.2017.2669638

Chen, T., Bahsoon, R., & Yao, X. (2018a). A Survey and Taxonomy of Self-Aware and Self-Adaptive Cloud Autoscaling Systems. *ACM Comput Surv*, *51*(3), Article 61 (June 2018), pp. 40. https://doi.org/10.1145/3190507

Chen, T., Bahsoon, R., & Yao, X. (2020). Synergizing Domain Expertise With Self-Awareness in Software Systems: A Patternized Architecture Guideline. *Proc IEEE*, *108*(7) (2020), 1094-1126. https://doi.org/10.1109/JPROC.2020.2985293

Chen, T., Li, K., Bahsoon, R., & Yao, X. (2018c). FEMOSAA: Feature-Guided and Knee-Driven Multi-Objective Optimization for Self-Adaptive Software. *ACM Transactions on Software Engineering and Methodology*, *27*(2), Article 5 (2018), pp. 50. https://doi.org/10.1145/3204459

Chen, Z., & Liu, B. (2018). Lifelong Machine Learning. Morgan & Claypool.

Cheng, B., et al. (2009a). Software Engineering for Self-Adaptive Systems: A Research Roadmap. *Springer*, 1-26. https://doi.org/10.1007/978-3-642-02161-9_1

Cheng, B., Sawyer, P., Bencomo, N., & Whittle, J. (2009b). A Goal-Based Modeling Approach to Develop Requirements of an Adaptive System with Environmental Uncertainty. In Model Driven Engineering Languages and Systems. Springer.

Dastani, M., & Testerink, B. (2016). Design patterns for multi-agent programming. *International Journal of Agent-Oriented Software Engineering*, *5*(2-3), 167-202.

de Winter, R., van Stein, B., & Bäck, T. (2021). SAMO-COBRA: A Fast Surrogate Assisted Constrained Multi-objective Optimization Algorithm. In Evolutionary Multi-Criterion Optimization. Springer.

Dearle, A. (2007). Software Deployment, Past, Present and Future. In *2007 Future of Software Engineering. IEEE Computer Society, USA*, 269284. https://doi.org/10.1109/FOSE.2007.20

Denkena, B., & Morke, T. (2017). Cyber-Physical and Gentelligent Systems inManufacturing and Life Cycle. Academic Press.

Det-Norske-Veritas. (2020). Technology Outlook 2030 - Safer, Smarter, Greener. (2020), 1-110. www.dnvgl.com

Dick, M. & Naumann, S. (2010). Enhancing Software Engineering Processes towards Sustainable Software Product Design. In *Integration of Environmental Information in Europe, Klaus Greve and Armin, B. Cremers (Eds.)*. Shaker Verlag, Aachen.

Dingsyr, T., Nerur, S., Balijepally, V., & Moe, N. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of Systems and Software*, *85*(6) (2012), 1213-1221. https://doi.org/10.1016/j.jss.2012.02.033 Special Issue: Agile Development.

Elhabbash, A., Salama, M., Bahsoon, R., & Tino, P. (2019). Self-Awareness in Software Engineering: A Systematic Literature Review. *ACM Transactions on Autonomous and Adaptive Systems*, *14*(2), Article 5 (Oct. 2019), pp. 42. https://doi.org/10.1145/3347269

Esfahani, N., & Malek, S. (2013). Uncertainty in Self-Adaptive Software Systems. *Springer*, 214-238. https://doi.org/10.1007/978-3-642-35813-5_9

European-Commission. 8/2021. Advanced Computing. (8/2021). https://www.nsf.gov/funding/pgm summ.jsp?pims_id=503306

Garlan, D., Cheng, S., Huang, A., Schmerl, B., & Steenkiste, P. (2004). Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure. *Computer*, *37*(10) (Oct. 2004), 4654. https://doi.org/10.1109/MC.2004.175

Georgiou, S., Rizou, S., & Spinellis, D. (2019). Software Development Lifecycle for Energy Efficiency: Techniques and Tools. *ACM Comput Surv*, *52*(4), Article 81 (Aug. 2019), pp. 33. https://doi.org/10.1145/3337773

Grötker, R. (2015). New autoMobility: The Future World of Automated Road Traffic. *National Academy of Science and Engineering, acatech Germany* (2015). https://elib.dlr.de/101368/2/acatech POSITION PAPER New autoMobility web.pdf

Grubbs, F.E. (1969). Procedures for detecting outlying observations in samples. *Technometrics*, *11*, 1.

Hasselmo, M.E. (2017). Avoiding Catastrophic Forgetting. *Trends in Cognitive Sciences*, *21*(6) (2017), 407-408. https://doi.org/10.1016/j.tics.2017.04.001

IBM. (2003). An Architectural Blueprint for Autonomic Computing. (2003). citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.150.1011 &rep=rep1&type=pdf

Jackson, M. (1997). The Meaning of Requirements. *Annals of Software Engineering. Springer 10480*, *3*(1) (1997), 5-21. https://doi.org/10.1023/A:1018990005598

Järvinen, J., Huomo, T., Mikkonen, T., & Tyrväinen, P. (2014). From Agile Software Development toMercury Business. In Software Business. Towards Continuous Value Delivery. Springer.

Jazdi, N. (2014). Cyber physical systems in the context of Industry 4.0. In *IEEE International Conference on Automation, Quality and Testing, Robotics*. 1-4. https://doi.org/10.1109/AQTR.2014.6857843

Juziuk, J., Weyns, D., & Holvoet, T. (2014). Design Patterns for Multi-agent Systems: A Systematic Literature Review. In *Agent-Oriented Software Engineering*. Vol. 9783642544323. Springer, 77-97.

Kephart, J., & Chess, D. (2003). The vision of autonomic computing. *Computer*, *36*(1), 41-50.

Kephart, J., Dibia, V., Ellis, J., Srivastava, B., Talamadupula, K., & Dholakia, M. (2019). An Embodied Cognitive Assistant for Visualizing and Analyzing Exoplanet Data. *IEEE Internet Computing*, *23*(2) (2019), 31-39. https://doi.org/10.1109/MIC.2019.2906528

Koutsoukos, X., Karsai, G., Laszka, A., Neema, H., Potteiger, B., Volgyesi, P., Vorobeychik, Y., & Sztipanovits, J. (2018). SURE: A Modeling and Simulation Integration Platform for Evaluation of Secure and Resilient CyberPhysical Systems. *Proc. IEEE*, *106*(1) (2018), 93-112. https://doi.org/10.1109/JPROC.2017.2731741

Kramer, J., & Magee, J. (2007). Self-Managed Systems: An Architectural Challenge. In Future of Software Engineering. *IEEE*, 259-268. https://doi.org/10.1109/FOSE.2007.19

Lehman, M. & Ramil, J. (2003). Software evolutionBackground, theory, practice. *Inform Process Lett*, *88*(1) (2003), 33-44. https://doi.org/10.1016/S0020-0190(03)00382-X To honour Professor, W.M. Turski's Contribution to Computing Science on the Occasion of his 65th Birthday.

Li, Z., Avgeriou, P., & Liang, P. (2015). A systematic mapping study on technical debt and its management. *Journal of Systems and Software*, *101*, 193-220. https://doi.org/10.1016/j.jss.2014.12.027

Mahdavi-Hezavehi, S., Avgeriou, P., & Weyns, D. (2017). A Classification Framework of Uncertainty in Architecture-Based Self-Adaptive SystemsWithMultiple Quality Requirements. InManaging Trade-Offs in Adaptable Software Architectures, Mistrik, I., Ali, N., Kazman, R., Grundy, J., & B. Schmerl (Eds.). Morgan Kaufmann, 45-77. https://doi.org/10.1016/B978-0-12-802855-1.00003-4

Marks, P., Muller, T., Vogeli, D., Jung, T., Jazdi, N., & Weyrich, M. (2018). Agent Design Patterns for Assistance Systems in Various Domains - a Survey. In *IEEE International Conference on Automation Science and Engineering (CASE)*. 168-173. https://doi.org/10.1109/COASE.2018.8560391

Meyer, M. (2014). Continuous Integration and Its Tools. *IEEE Software*, *31*, 03 (may 2014), 14-16. https://doi.org/10.1109/MS.2014.58

Mishra, A., & Otaiwi, Z. (2020). DevOps and software quality: A systematic mapping. *Computer Science Review*, *38* (2020), 100308. https://doi.org/10.1016/j.cosrev.2020.100308

Moreno, G., Ca´mara, J., Garlan, D., & Schmerl, B. (2015). Proactive Self-Adaptation under Uncertainty: A ProbabilisticModel Checking Approach. In *10th Joint Meeting on Foundations of Software Engineering. ACM*, 112. https://doi.org/10.1145/2786805.2786853

Muccini, H. & Vaidhyanathan, K. (2021). Software Architecture for ML-based Systems: What Exists and What Lies Ahead. *arXiv:2103.07950* [cs.SE]

Musil, A., Musil, J., Weyns, D., Bures, T., Muccini, H., & Sharaf, M. (2017). Patterns for Self-Adaptation in Cyber-Physical Systems. *Springer*, 331-368. https://doi.org/10.1007/978-3-319-56345-9_13

Musil, J., Musil, A., Weyns, D., & Biffl, S. (2015). An Architecture Framework for Collective Intelligence Systems. In *12th Working IEEE/IFIP Conference on Software Architecture*. 21-30. https://doi.org/10.1109/WICSA.2015.30

Musi, S. & Hirche, S. (2016). Classification of human-robot team interaction paradigms. *IFAC-PapersOnLine*, *49*(32), 42-47. https://doi.org/10.1016/j.ifacol.2016.12.187 Cyber-Physical & Human-Systems.

Naumann, S., Dick, M., Kern, E., & Johann, T. (2011). The GREENSOFT Model: A reference model for green and sustainable software and its engineering. *Sustainable Computing: Informatics and Systems*, *1*(4) (2011), 294-304. https://doi.org/10.1016/j.suscom.2011.06.004

Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimhigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., & Wolf, A.L. (1999). An architecture-based approach to self-adaptive software. *IEEE Intelligent Systems and their Applications*, *14*(3) (1999), 54-62.

Palar, P.S., Yang, K., Shimoyama, K., Emmerich, M., & Bäck, T. (2018). Multi-Objective Aerodynamic Design with User Preference Using Truncated Expected Hypervolume Improvement. In Genetic and Evolutionary Computation Conference (Kyoto, Japan). Association for Computing Machinery, New York, NY, USA, 13331340. https://doi.org/10.1145/3205455.3205497

Parisi, G.I., Kemker, R., Part, J.L., Kanan, C., & Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*, *113*, 54-71. https://doi.org/10.1016/j.neunet.2019.01.012

Paulovich, F., De Oliveira, M., & Oliveira, O. (2018). A Future with Ubiquitous Sensing and Intelligent Systems. *ACS Sensors*, *3*(8) (2018), 1433-1438. https://doi.org/10.1021/acssensors.8b00276

Reussner, R., Goedicke, M., Hasselbring, W., Vogel-Heuser, B., Keim, J., & Martin, L. (2019). Managed Software Evolution. Springer Nature.

Rodriguez, P., et al. (2017). Continuous deployment of software intensive products and services: A systematic mapping study. *Journal of Systems and Software*, *123*, 263-291. https://doi.org/10.1016/j.jss.2015.12.015

Salehie, M. & Tahvildari, L. (2009). Self-Adaptive Software: Landscape and Research Challenges. *ACM Trans Auton Adapt Syst*, *4*(2), Article 14 (May 2009), pp. 42. https://doi.org/10.1145/1516533.1516538

Schelfthout, K., Coninx, T., Helleboogh, A., Holvoet, T., Steegmans, E., & Weyns, D. (2002). Agent implementation patterns. *Workshop on Agent-Oriented Methodologies*, 119-130.

Selic, B. (2020). Controlling the Controllers: What Software People Can Learn From Control Theory. *IEEE Softw*, *37*(6) (2020), 99-103. https://doi.org/10.1109/MS.2020.3006970

Sztipanovits, J., Koutsoukos, X., Karsai, G., Kottenstette, N., Antsaklis, P., Gupta, V., Goodwine, B., Baras, J., & Wang, S. (2012). Toward a Science of CyberPhysical System Integration. *Proc IEEE*, *100*(1), 29-44. https://doi.org/10.1109/JPROC.2011.2161529

Tamai, T. (2019). Key Software Engineering Paradigms and Modeling Methods. *Springer International Publishing, Cham*, 349-374. https://doi.org/10.1007/978-3-030-00262-6_9

Tao, F., Zhang, H., Liu, A., & Nee, A. (2019). Digital Twin in Industry: State-of-the-Art. *IEEE Transactions on Industrial Informatics*, *15*(4) (2019), 2405-2415. https://doi.org/10.1109/TII.2018.2873186

Tavcar, J. & Horváth, I. (2019). A Review of the Principles of Designing Smart Cyber-Physical Systems for Run-Time Adaptation: Learned Lessons and Open Issues. *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, *49*(1) (2019), 145-158. https://doi.org/10.1109/TSMC.2018.2814539

Thill, M., Konen, W., Wang, H., & Bäck, T. (2021). Temporal convolutional autoencoder for unsupervised anomaly detection in time series. *Applied Soft Computing*, *112*, 107751. https://doi.org/10.1016/j.asoc.2021.107751

Thrun, S. & Mitchell, T.M. (1995). Lifelong Robot Learning. In *The Biology and Technology of Intelligent Autonomous Agents. Springer*, 165-196.

Tzafestas, S.G. (2012). Advances in intelligent autonomous systems. Springer.

Weyns, D. (2019). Software Engineering of Self-adaptive Systems. In *Handbook of Software Engineering.*, Sungdeok Cha, Richard Taylor, N., & Kyo, C. Kang (Eds.). 399-443. https://doi.org/10.1007/978-3-030-00262-6

Weyns, D. (2021). Introduction to Self-Adaptive Systems: A Contemporary Software Engineering Perspective. Wiley. ISBN 978-1-119-57494-1.

Weyns, D., Andersson, J., Caporuscio, M., Flammini, F., Kerren, A., & Löwe, W. (2022). A Research Agenda for Smarter Cyber-Physical Systems. *Journal of Integrated Design and Process Science* (2022). https://doi.org/10.3233/JID-210010

Weyns, D., Bäck, T., Vidal, R., Yao, X., & Belbachir, A.N. (2021a). Lifelong Computing. arXiv abs/2108.08802 (2021).

Weyns, D., Bencomo, N., Calinescu, R., Camara, J., Ghezzi, C., Grassi, V., Grunske, L., Inverardi, P., Jezequel, J-M., Malek, S., Mirandola, R., Mori, M., & Tamburrelli, G. (2017). Perpetual Assurances for Self-Adaptive Systems. In *Software Engineering for Self-Adaptive Systems III*. Assurances, R. de Lemos, Garlan, D., Ghezzi, C., & H. Giese (Eds.). Springer International Publishing, Cham, 31-63.

Weyns, D., Bures, T., Calinescu, R., Craggs, B., Fitzgerald, J., Garlan, D., Nuseibeh, B., Pasquale, L., Rashid, A., Ruchkin, I., & Schmerl, B. (2021b). Six Software Engineering Principles for Smarter Cyber-Physical Systems. In *IEEE International Conference on Autonomic Computing and Self-Organizing Systems, ACSOS* 2021, Companion Volume, Washington, DC, USA, September 27 - Oct. 1, 2021. IEEE, 198-203. https://doi.org/10.1109/ACSOS-C52956.2021.00058

Weyns, D., Caporuscio, M., Vogel, B., & Kurti, A. (2015). Design for Sustainability = Runtime Adaptation U Evolution. In *1st International Workshop on Sustainable Architecture: Global collaboration, Requirements, Analysis (Dubrovnik, Cavtat, Croatia)*. https://doi.org/10.1145/2797433.2797497

Weyns, D. & Iftikhar, M.U. (2022). ActivFORMS: A Formally-Founded Model-Based Approach to Engineer Self-Adaptive Systems. *ACM Transactions on Software Engineering and Methodology*, *31*(3) (2022).

Weyns, D., Iftikhar, U., Hughes, D., & Matthys, N. (2018). Applying Architecture-Based Adaptation to Automate the Management of Internet-of-Things. In *Software Architecture. Springer*, 49-67.

Weyns, D., Malek, S., & Andersson, J. (2010). FORMS: A Formal Reference Model for Self-Adaptation. In *Proceedings of the 7th International Conference on Autonomic Computing* (Washington, DC, USA) (*ICAC' 10*). Association for Computing Machinery, New York, NY, USA, 205214. https://doi.org/10.1145/1809049.1809078

Wooldrige, M. (2009). An Introduction to MultiAgent Systems. Wiley. ISBN 978-0-470-51946-2.

You, C., Robinson, D., & Vidal, R. (2017). Provable Self-Representation Based Outlier Detection in a Union of Subspaces. arXiv:1704.03925 [cs.CV]

Yu, G., Jin, Y., & Olhofer, M. (2020). Benchmark Problems and Performance Indicators for Search of Knee Points in Multiobjective Optimization. *IEEE Transactions on Cybernetics*, *50*(8), 3531-3544. https://doi.org/10.1109/TCYB.2019.2894664

Yu, X. & Xue, Y. (2016). Smart Grids: A CyberPhysical Systems Perspective. *Proc. IEEE*, *104*(5), 1058-1070. https://doi.org/10.1109/JPROC.2015.2503119

Zeng, J., Yang, L., Lin, M., Ning, H., & Ma, J. (2020). A survey: Cyber-physical-social systems and their system-level design methodology. *Future Generation Computer Systems*, *105*, 1028-1042. https://doi.org/10.1016/j.future.2016.06.034