Comparison of Approaches for Self-Improvement in Self-Adaptive Systems

Christian Krupitzer*, Felix Maximilian Roth*, Martin Pfannemüller*, and Christian Becker*
*Chair of Information Systems II, University of Mannheim
Schloss, 68131 Mannheim, Germany
Email: {christian.krupitzer, felix.maximilian.roth, martin.pfannemueller, christian.becker}@uni-mannheim.de

***Abstract*—Various trends such as mobility of devices, Cloud Computing, or Cyber-Physical Systems lead to a higher degree of distribution. These systems-of-systems need to be integrated. The integration of various subsystems still remains a challenge. Self-improvement within self-adaptive systems can help to shift integration tasks from the static design time to the runtime, which fits the dynamic needs of these systems. Thus, it can enable the integration of system parts at runtime.**

In this paper, we define self-improvement as an adaptation of an Autonomic Computing system's adaptation logic. We present an overview of approaches for self-improvement in the domains of Autonomic Computing and self-adaptive systems. Based on a taxonomy for self-adaptation, we compare the approaches and categorize them. The categorization shows that the approaches focus either on structural or parameter adaptation but seldomly combine both. Based on the categorization, we elaborate challenges, that need to be addressed by future approaches for offering self-improving system integration at runtime.

I. Introduction

Trends as Cyber-Physical Systems with its growing number of mobile and embedded devices as well as the omnipresence of (wireless) networks results in a higher degree of distribution. Research communities in the domains of Autonomic Computing, Organic Computing, or self-adaptive systems try to tackle these challenges through shifting activities from design time to runtime, which leads to the need of automated system integration. The foundation of self-adaptation are the *self-* properties* [1], [2]. One of them is self-improvement, which supports the integration of system parts at runtime as it supports the "continuous development" of systems through continuous improvements.

In this paper, we focus on self-improvement within self-adaptive systems. We provide the following contributions. First, we present an overview on approaches for self-improvement. Second, we compare the approaches based on our taxonomy for self-adaptation [3]. Last, we discuss the strengths and weaknesses of the approaches and elaborate challenges that need to be addressed in future research.

The structure of the remaining paper reflects these contributions. Section II introduces the concept of self-adaptation as well as defines the terms self-adaptive systems (SASs) and self-improvement. Section III presents surveys that are similar to this work. In Section IV, we present the different approaches for self-improvement in the domain of SASs. Section V compares the approaches using the taxonomy from Section II as metric. In Section VI, we discuss the approaches and derive challenges for future work. Finally, Section VII concludes the paper with a summary.

II. Background

Self-adaptation is the ability of a system to adapt its behavior to changes in the system itself or in its environment [4], [1]. Self-adaptation has different dimensions that have to be taken into account when implementing an SAS. Next, we present these dimensions, define the terms *self-adaptive system* and *self-improvement*, as well as present the structure of these systems.

A. Taxonomy on Self-Adaptation

In [3], we present a taxonomy on self-adaptation and use it for categorizing engineering approaches for SASs. As shown in Figure 1, this taxonomy consists of five dimensions: reason, time, technique, level, and adaptation control. In the following, we explain these dimensions in detail.

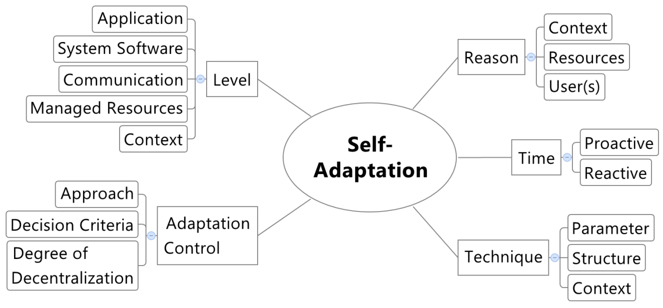

Fig. 1: Taxonomy for self-adaptive systems, based on [3].

The first dimension is the *reason* for an adaptation. A reason can be a change in context, in the system's resources, or a change (e.g., changing goals) caused by the user which includes a possible administrator. The *time* dimension is divided into reactive (reaction after a change) and proactive (action before a change). *Techniques* can be parameter adaptation or structural adaptation (including algorithmic and compositional adaptation). Additionally, the context itself can be adapted. As the *level* of the adaptation, we identified the application itself, the system software, the communication, the technical resources, or the context. The last dimension is *adaptation control*. It is split into three subdimensions: adaptation approach, adaptation decision criteria, and degree of decentralization. The *approach* can be internal (i.e., interwoven with the resources) or external (i.e., separated from the resources). In literature, the following *decision criteria* are present: models, rules/policies, goals, or a utility (function). The *degree of decentralization* describes if various subsystems are responsible for controlling the adaptation or whether the functionality is centralized. We will compare different approaches for self-improvement of SAS based on this taxonomy in Section V.

B. Self-Adaptive Systems and Self-Improvement

According to [4], a *self-adaptive system* (SAS) "modifies its own behavior in response to changes in its operating environment". Such systems consist of two main elements: the managed resources (MRs) and the adaptation logic (AL) [2]. MRs can be all types of computational resources and range from small scale smartphones, laptops, or robotics to large scale systems-of-systems like cars, production facilities, or data centers and provide the functionality of the system. The AL monitors the MRs as well as the environment and performs adaptations on the MRs. Therefore, the AL implements some kind of feedback loop, such as the MAPE cycle [1] known from Autonomic Computing.

The AL implements a set of the *self-* properties* [1], [2]. IBM identified four self-* properties as most important for Autonomic Computing systems: *self-configuration*, *self-optimization*, *self-healing*, and *self-protection* [1]. With respect to this paper, we analyze the self-improvement property. As there is not a common definition present in literature, we define *self-improvement* as following:

> *Self-improvement of the AL is the adjustment of the AL to handle former unknown circumstances or changes in the environment or the MRs.*

In our understanding, a system can only self-improve, if the AL itself is changed. Otherwise, the AL can neither handle unknown situations nor improve the performance of adaptations. In contrast, self-optimization does change the MR but not the AL. The same is true for self-optimizing hierarchical approaches (e.g., [5] or [6]) as the hierarchy offers decision-making on different levels with different scopes but do not change the AL in a substantial way. Section IV presents different approaches for self-improvement in SASs.

III. Related Work

In literature, different surveys on SASs and Autonomic Computing can be found. This section presents an overview and highlights the differences to this work.

In [3], we present a taxonomy on self-adaptation and use it for the categorization of engineering approaches for SASs. There, we focused on how to build the AL for changing the MR. Contrary, in this work, we focus on the level above and how to change the AL at runtime. In [7], Macías-Escrivá *et al.* describe approaches, research challenges, and applications for SASs. Salehie and Tahvildari presented an overview over the landscape of self-adaptive software and related research challenges [2].

Other authors focus on formal specifications within SASs and presented surveys on formals methods. In [8], Bradbury *et al.* survey 14 formal specification approaches based on graphs, process algebras, logic, and other formalisms. Weyns *et al.* present a systematic literature review that showed that the number of studies that employ formal methods in SASs remains still low [9].

Further surveys concentrate on more specific aspects. Psaier and Dustdar focused on the self-healing aspect and categorized approaches for self-healing in ten research areas [10]. McKinley *et al.* highlight the difference regarding parameter vs. compositional/structural adaptation and survey approaches for both of them [11]. Oreizy *et al.* discuss the spectrum of adaptation from static activities to dynamic ones [4].

Another two surveys focus on specific aspects within the Autonomic Computing domain. Huebscher and McCann presented an overview of Autonomic Computing and its applications [12]. Dobson *et al.* focus on aspects of autonomic communications [13].

All these surveys provide important insights into the field of SASs. However, to the best of our knowledge, no survey in the field focuses on approaches for adaptation of the adaptation logic. This is the focus within this paper. In the following, we present approaches for self-improvement and compare them.

IV. Approaches for Self-Improvement

AL adaptation may have several goals, such as (i) self-healing (recovering from failures) or (ii) self-improvement. According to [14], reasons for self-improvement can be the need for an adjustment of the AL's structure to reflect changes in the MRs or an enhancement of the performance through proactive adaptation of the AL's parameters. As an example, we consider an adaptive production cell with a dynamic interaction scheme of robots (cf. [15]). In this scenario, rules define which robots should interact. By changing rules, the interaction scheme can be adapted, e.g., for fitting the production plans of different items. In this case the coordination of MRs can be improved. However, for improving the system over time, the AL needs to be changed for reacting to new conditions that have not been taken into account at design time. Otherwise, the AL can only respond to known and anticipated events.

This section provides an overview of existing approaches for self-improvement or evolution, respectively, of the AL.

1) Three Layer Architecture (3LA): Figure 2 shows the *Three Layer Architecture* (3LA) by Kramer and Magee [16]. Within the 3LA, MRs are part of the Component Control layer. The layer provides the interfaces for monitoring and adapting the resources. Beyond that, small self-tuning algorithms can be included as well. Additionally, the layer detects situations that cannot be handled by the current setup and propagates them to the Change Management layer. Using predefined plans, the Change Management layer determines a sequence of actions to handle the new situation identified through the monitored state. If no predefined plan matches the given situation, the Goal Management layer is invoked. This layer is responsible for the creation of the plans for the Change Management layer.

As the name of the layer indicates, it is based on a set of user-defined goals that can change over time.

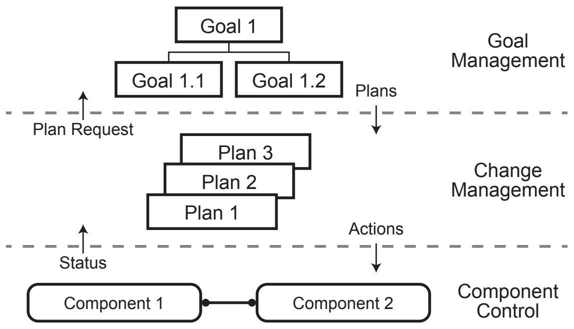

Fig. 2: Overview of the Three Layer Architecture [16].

2) ActivFORMS: The basic idea of the *ActivFORMS* approach is the direct execution of formal models using a virtual machine instead of implementing the models in code [17]. The behavior of the system can be verified both at design time and at runtime and, therefore, it is possible to guarantee a correct adaptation behavior of the system. Following the architecture proposed by Kramer and Magee [16], *ActivFORMS* divides the system into three layers. The bottom layer consists of the MRs. Above, the Active Model Engine (representing the AL of the system) contains the virtual machine which executes the formal model. A formal model is represented as a network of timed automata and contains the AL in form of a MAPE-K loop. Whenever the system detects that the currently executed formal model cannot deal with the state of the system, the uppermost layer – the Goal Management layer – is invoked. The Goal Management layer tries to find a different formal model that can satisfy the system goals and send it to the Active Model Engine, which executes the new formal model and adapts the resources accordingly.

3) NoMPRoL: SASs usually operate in unstable and dynamic environments. This leads to incomplete and inaccurate models due to high complexity and uncertainty. As a result, a model should be updated when the environment changes. In the *NoMPRoL* approach [18], an SAS is composed of the three layers known from the 3LA approach [16]. The Change Management layer acts based on reactive plans which are generated in the Goal Management layer. The reactive plans consist of actions for known and anticipated states and can, therefore, handle discrepancies between the expected and the actual behavior. As the domain model evolves with changes in the environment, the Goal Management layer changes the reactive plan over time for adjusting it to new states. The Goal Management layer observes the reaction of the environment to actions of the Component layer using execution traces. With the help of probabilistic rule learning, the system generates rules based on these execution traces that represent known and anticipated states. Finally, the rules are used to update the Goal Management layer and, accordingly, the reactive plans of the Change Management layer.

4) Dynamic Control Loops (DCL): Often, an SAS is composed of several, independent control loops. Each control loop is responsible for a certain behavior of the system. All control loops together form the AL. In [19], the authors propose a system for adding/removing control loops to/from the AL. Additionally to a Java framework, which enables the change of control loops at runtime, the authors of [19] propose a technique to generate models for systems composed out of several control loops. With the support of a goal model compiler, the approach supports the system developer in the identification of control loops.

5) PLASMA: PLASMA [20] utilizes user-defined goals and two kinds of models as adaptation decision criteria: (i) a domain model that captures all possible states of the system's components and (ii) an adaptation model which describes the possible architectural configurations of the system. The system has three distinct layers. The application layer is the lowest layer and consists of the MRs. The adaptation layer offers plan-based adaptation of the MRs in the application layer. Finally, the planning layer handles the generation of plans for the other layers. The plans for the AL describe the desired architecture for the application. Likewise, the plans for the application layer define the possible adaptations. A possible reason for an adaptation can be changes in the high-level goals of the system, which are provided by the user, or component failures.

6) FUSION: FUSION supports the development of feature-oriented SASs [21]. The architecture of the system can be divided into three parts: the running system, an adaptation cycle, and a learning cycle. A feature can either be active or inactive. It is not intended that features are added to the system at runtime. The adaptation cycle adapts the MRs by turning features on and off. Therefore, it collects data from the running system, calculates the utility of the overall system and checks, whether goals of the system are violated. In case of a violation, it tries to find a different selection of features, which increases the utility and satisfies the goals. For this process, it relies on a shared knowledge base. The learning cycle is responsible for creating the knowledge base by learning the impact of adaptation decisions.

7) KAMI: The *KAMI* approach focuses on models for non-functional requirements like reliability or performance [22]. The AL uses these models to reason about adaptations. Usually, models for estimating these properties solely rely on estimates by either domain experts or they can be extracted from similar running systems. In *KAMI*, models should not only be used at design time but instead also be updated at runtime to fit system's evolution. By collecting data from the running system, a Bayesian estimator can update the model and, therefore, keep the model in sync with the current situation. This model can then be used by the AL for further analysis, e.g., to detect whether the non-functional requirements are fulfilled. Beyond that, the AL can predict possible violations in the future. In both situations, a violation triggers an adaptation of the system to counteract the deficiency. Using a plug-in architecture, the approach can be used with different model types suitable for different requirements.

8) Organic Traffic Control (OTC): Within the *Organic Traffic Control* (OTC), an SAS uses evolutionary algorithms for the control of road traffic signals in urban areas ([23], [24]). The MRs are traffic light controllers. Values for their cycle times and the offset to phases of other traffic lights can be adjusted. These parameters are modified by a learning classifier system which uses rules and selects an action based on the highest expected reward. The associated action of the selected rule contains the values for the parameters of the traffic light controller. Additionally, the system performs an off-line optimization of the parameters. Therefore, unforeseen traffic situations are generated and by combining the parameters of the traffic light controller using evolutionary algorithms, new combinations are generated. These combinations are evaluated using a traffic simulator of the intersection and, finally, added to the learning classifier system of the AL.

In [24], the OTC is extended. There, intersections collaborate and can form dynamic progressive signal systems (DPSSs). The coordination between the intersections improves the traffic flow in the area of the connected intersections.

9) FESAS ALM: In [14], we propose an approach for adapting the AL which enables self-improvement. We extend an SAS with an additional layer: the *Adaptation Logic Manager* (ALM) for adapting the AL of the SAS. The ALM is an AL for the AL, hence, it consists of a feedback loop represented by MAPE components. Furthermore, additional components are introduced, e.g., for prediction of future events or learning rules. The communication between the AL and the ALM is performed via a proxy, the so called Proxy ALM. The Proxy ALM is integrated in the FESAS Middleware for developing SASs [25]. The Proxy ALM collects information from the AL (e.g., the structure, algorithms, and monitored data) and sends this information to the ALM. In case there is potential to improve the AL's performance, the ALM will adapt the AL. The Proxy ALM receives adaptation plans from the ALM.

10) Learning and Evolution in DSPLs: In [3], we present an overview of approaches that use Dynamic Software Product Lines (DSPL) approaches for reasoning. Often, developers specify SPLs at design time and the information is used for finding alternative configurations and adaptation paths at runtime. In [26], the authors present an approach for extending DSPLs with learning and evolution. A reinforcement learning approach searches new adaptation rules in the configuration space. These rules are added to the AL. Additionally, evolution is triggered if the user adds new requirements or if the learning was not successful. This can happen if learning could not find a configuration for the current context. In this case, developers can re-define the DSPL. After redefining the configuration space, learning is triggered again.

11) Requirements@Runtime (Reqs@RT): In [27], the authors present an approach where the designer can change requirements at runtime. In order to support requirement changes, a goal model (based on FLAGS [28]) and an implementation model are maintained. A mapping between these two models allows the correct handling of requirement changes at runtime. Adaptations resulting from such changes can have an impact on the goal model as well as the implementation model.

12) Further Approaches: Further approaches can be found in literature that do not focus on but could handle specific aspects for self-improvement. In the following, we present some of them.

EUREMA [29] offers an approach for modeling megamodels in hierarchies that integrate different runtime models. Models in higher layers monitor relations and adapt runtime models in lower layers. In [30], the authors offer MAPE-K templates for formal modeling of the behavior in the AL. The authors of [31] formalize patterns for self-adaptation with corresponding feedback loops and create a taxonomy. They claim that this supports structural adaptation of the AL as the formalization offers exchangeability. However, none of these approaches include components that automatically use the information at runtime for self-improvement of the AL.

In [32], the authors describe a technique for synthesizing changes of different versions of controllers (comparable to an AL). However, this solution must be implemented individually for each system and adaptation is performed by system administrators.

Through machine learning, it is possible to improve the AL, e.g., through learning new rules or updating goals. Different approaches can be found in literature. For further information about these approaches, the interested reader is referred to the overviews presented in [21] or [3]. However, most of these approaches are highly use case dependent [21] or cannot cope with new context situations.

V. Comparison

In the following, we compare the approaches from Section IV using the aforementioned taxonomy on self-adaptation from [3]. We neglect approaches for adapting the MR and focus on adapting the AL, as this corresponds to our definition of self-improvement (cf. Section II). Furthermore, for the comparison, we exclude the approaches from Section IV-12 as they are not integrated into an approach for self-improvement.

1) Three Layer Architecture (3LA): According to [16], the reasons to adapt the AL in *3LA* are the introduction of new goals by the user, changes in the context, or technical resources (e.g., a component failed). It is a reactive framework with goal-based reasoning that supports parameter and compositional adaptation of the AL (depends on the approach; not specified in [16]). The approach is centralized and, since the responsibility for adaptation reasoning is separated from the system's functionality, it is an external approach as well.

2) ActivFORMS: In *ActivFORMS*, the Goal Management layer reacts if (i) the user adds new goals or changes existing ones or (ii) the Active Model Engine cannot handle a change in a technical resource or the context. Therefore, the adaptation is reactive. In both cases, it triggers a goal-based adaptation of the active model. Since the model gets changed, we categorize it as a parametric approach. ActiveFORMS uses a centralized, external approach for adapting the AL.

3) NoMPRoL: With respect to the taxonomy, the reason for an adaptation is a context change in the system. The execution traces are continuously collected and analyzed. The analysis changes values of the planning model (parameter adaptation). However, it reacts on analyzed results which makes it a reactive approach. The adaptation decision criteria are based on a domain model in combination with a probabilistic rule learner using the execution traces. The adaptation is realized using a centralized and external approach.

4) Dynamic Control Loops (DCL): The API for adding and removing control loops enables structural adaptation of the AL. As the administrator triggers a change of the control loops, this is a reactive adaptation. The approach can be characterized as external and centralized, as a clearly defined interface for interaction exists. However, the authors claim that the adaptation could also be triggered by some component of the system [19]. The decision criteria of an adaptation is not specified by the authors.

5) PLASMA: PLASMA reacts on changing (user) goals and system component failures, resulting in a reactive approach. The external Planning Layer exchanges complete plans in the Adaptation Layer. Hence, it offers parameter adaptation. PLASMA uses goals and models as decision criteria. It works in a centralized fashion.

6) FUSION: In *FUSION*, the learning cycle detects new patterns in observed data from context and MRs and reacts by adapting the feature models accordingly (parameter adaptation). Therefore, the adaptation is reactive. It is implemented as centralized, external adaptation logic and uses goal utility functions as decision criteria.

7) KAMI: KAMI uses an online parameter adaptation technique. The type of analysis performed on the model determines whether the approach is used reactively, proactively, or both. Predictions about possible future violations are possible, making *KAMI* proactive. Using the predictions and by recognizing changes in the context, *KAMI* updates the runtime model accordingly but is limited to parameter adaptations because only numerical values of the models can be updated. The update of the knowledge base is done externally. *KAMI* uses the system model as decision criterion in the centralized control.

8) Organic Traffic Control (OTC): In [23], an off-line simulator in combination with an evolutionary algorithm is used to learn parameters for unknown traffic situations. Using utility functions, the result of the simulation is evaluated. If the simulation results improve the traffic situation (context), the AL is adapted proactively with the improved set of parameters. In [23], the OTC is limited to a single intersection, hence, it is centralized.

Additionally to the learning classifier system in [23], the creation of DPSSs introduced in [24] is organized in a decentralized way. The collaborations represent a structural adaptation technique. DPSSs are formed as a response to the current traffic situation. Hence, it is reactive. Within both systems, adaptations are controlled externally and the evaluation of different traffic situations uses utility functions.

9) FESAS ALM: A first prototype implementation of the ALM is currently under development. Due to simplicity reasons, the prototype implementation follows a centralized approach. The ALM is added as an additional layer to the AL (external approach) for improved maintainability and reduced dependability [33]. It responds to changes in the MRs or the context. Besides reactive, structural adaptation of the AL, the ALM offers proactive parameter adaptation in form of learning new rules. The current prototype implementation of the ALM uses rules and utility functions for reasoning.

10) Learning and Evolution in DSPLs: The approach for learning and evolution of DSPLs presented in [26] adds additional layers for rule learning and configuration space evolution to an Autonomic Computing system. Therefore, it is an external approach. The authors do not make any claims about the degree of decentralization, however, the system model shows a centralized design. Adaptation is triggered by context change or when the user adds new requirements. Hence, it is reactive. Evolution is model-based as it uses mathematical models and delivers a new DSPL configuration space as output. Learning runs continuously to find and add new adaptation rules (parameter adaptation), hence, it is proactive. The reinforcement learning approach is utility-based.

11) Requirements@Runtime (Reqs@RT): In [27], adaptations are triggered by the user or when the goal model changes. In both cases, the goal model is used for reasoning and potentially adapted (parameter adaptation). Thus, the approaches are reactive. The decision module for AL adaptations is directly interwoven in the AL, which makes the approach internal. Rules are specified in order to cope with requirement changes. They use a centralized approach.

VI. Discussion

The last section compared 12 approaches for self-improvement using the taxonomy of [3] as metric. Table I shows the results of the approaches' comparison. In this section, we discuss the results of the comparison and derive challenges for self-improvement.

Most of the approaches integrate reactive adaptations. Four approaches combine reactive and proactive behavior. Only the approach in [23] works purely proactively. In many cases, a reactive adaptation can be sufficient as usually the AL should find an appropriate adaptation for the MRs. Hence, a reactive adaptation of the AL acts as a backup mechanism. However, self-improvement works best if the AL is proactively enhanced as it eliminates adaptation delays. Developers of future approaches for self-improvement should consider both

Approach	Time	Reason	Technique	Adaptation Control		
				Approach	Decision Criteria	(De)centralization
3LA [16]	Reactive	Context/MR/User	not specified	External	Goal	Centralized
ActivFORMS [17]	Reactive	Context/MR/User	Parameter	External	Goal	Centralized
NoMPRoL [18]	Reactive	Context	Parameter	External	Model/Rules	Centralized
DCL [19]	not specified	User	Structure	External	not specified	Centralized
PLASMA [20]	Reactive	MR/User	Parameter	External	Model/Goal	Centralized
FUSION [21]	Reactive	Context/MR	Parameter	External	Goal/Utility	Centralized
KAMI [22]	Reactive/Proactive	Context	Parameter	External	Model	Centralized
OTC [23]	Proactive	Context	Parameter	External	Utility	Centralized
OTC DPSS [24]	Reactive/Proactive	Context	Structure/Parameter	External	Utility	Decentralized
FESAS [14]	Reactive/Proactive	Context/MR	Parameter/Structure	External	Rules/Utility	Centralized
DSPLs [26]	Reactive/Proactive	Context/User	Parameter	External	Model/Utility	Centralized
Reqs@RT [27]	Reactive	MR/User	Parameter	Internal	Rules	Centralized

TABLE I: Approaches for the evolution of the adaptation logic classified using the taxonomy of [3]. As we concentrated on self-improvement of the AL, the level is always the application (here: the AL). System administrators are classified as users.

possibilities for higher flexibility and improved adaptation results.

As Table I shows, there is a high diversity within the adaptation reason dimension. This indicates that the reason in the approaches might be use case specific. Therefore, we further analyzed the domains of the approaches' use cases. We identified four domains: traffic management, web services, data center management, and robotics. These are core domains of Autonomic Computing and SAS. Table II shows the use cases. Future work should elaborate on common, generic strategies to offer more reusable approaches for self-improvement or provide guidelines within use cases and generic guidelines across use cases. Therefore, developing a benchmark for comparing the runtime performance of the approaches is necessary.

Approach	Use case	Domain
3LA [16]	not specified	-
ActivFORMS [17]	Robotic warehouse transportation	Robotics
NoMPRoL [18]	Robotic factory transportation	Robotics
DCL [19]	Dust cleaning robot	Robotics
PLASMA [20]	Robotic convoy, e.g., for inventory management	Robotics
FUSION [21]	Travel Reservation System	Web services
KAMI [22]	Medical assistance web service orchestration	Web services
OTC [23]	Traffic lights at single intersection	Traffic management
OTC DPSS [24]	Traffic lights at multiple intersections	Traffic management
FESAS [14]	not specified	-
DSPLs [26]	VM management, product line management	Data center management
Reqs@RT [27]	Web portal for ordering food from restaurants	Web services

TABLE II: Use cases of the approaches.

The majority of the methods uses parameter adaptation. Only two approaches provide both possibilities. [19] includes structural adaptation only, however, it does not offer an automated approach. Additionally, none of the approaches with structural adaptation has a proactive behavior. Future approaches should include structural adaptation of the AL. This might be beneficial to better fit changes in the MRs or the context. For example, consider the adaptive production cell [15] from Section IV. Assuming a master/slave pattern [34], slaves are not able to coordinate if the master crashes. Therefore, a new master has to be selected - a structural adaptation of the AL is required.

The fact that almost every method works with an external approach corresponds to the findings of [33]. There, the authors claim that an external approach offers better maintainability as well as extensibility. In terms of the degree of decentralization only one method (the OTC) works decentralized. All remaining approaches are centralized. This reflects that in most approaches the AL is centralized, too. The fact that a global view is facilitated by a centralized setting might also be a reason. One possible challenge for future work is to offer self-improvement in decentralized settings to improve scalability. The results indicate a correlation between proactive methods and utility functions. However, the decision criteria are mixed for reactive adaptations. This reveals (comparable to the adaptation reason) that the decision criteria might be use case specific. Future work could focus on generalizing or defining guidelines, when to use which criteria.

One has to mention, that it is possible to achieve self-improvement by considering the AL itself as MR and adding an additional component for adapting the AL. This way, common approaches for building SASs (e.g., Rainbow [35], Archstudio [4], or some of the approaches presented in [3]) could be used for adapting the AL and self-improvement, respectively. However, to the best of our knowledge, current research projects have not addressed this so far.

VII. Conclusion and Outlook

In this paper, we presented and compared approaches for self-improvement in the Autonomic Computing and self-adaptive systems domain. We compared 12 approaches from different research communities, such as goal-based evolution, DSPLs, machine learning, and requirements@runtime.

The comparison showed that most of the approaches use an external, centralized approach for control. Decision criteria as well as reason indicate use case specific implementations.

Here, future work could elaborate on generic solutions or guidelines for use cases. Regarding the technique, parameter adaptation prevail structural adaptation. As shown in the example in Section VI, structural adaptation of the AL can be beneficial. Future work should address this. Most of the approaches focus on reactive adaptation. A stronger focus on proactive adaptation would be beneficial as it offers adaptation without interruption. In the case of system integration, structural proactive adaptation could prepare the integration of system parts proactively and boost the integration process. Developing a benchmark would enable the evaluation and comparison of the different approaches' runtime performance. This is an important aspect for future work.

We try to tackle these challenges within the FESAS project [25], [14]. There, we implement a framework for self-improvement that can integrate different approaches and offer support for developers for facilitating self-improvement. The framework should combine proactive, parameter adaptation in the form of rule learning as well as reactive structural adaptation.

Acknowledgment

This work was supported by the Julius-Paul-Stiegler-Memorial-Foundation. The authors would like to thank their former student Fabian Kajzar for his contribution.

References

[1] J. O. Kephart and D. M. Chess, "The Vision of Autonomic Computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[2] M. Salehie and L. Tahvildari, "Self-Adaptive Software: Landscape & Research Challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, p. Art. 14, 2009.

[3] C. Krupitzer, F. M. Roth, S. VanSyckel, G. Schiele, and C. Becker, "A survey on engineering approaches for self-adaptive systems," *Perv. and Mobile Comp. Journal*, vol. 17, no. B, pp. 184–206, 2015.

[4] P. Oreizy, M.M. Gorlick, R.N. Taylor, D. Heimhigner, and G. Johnson et al., "An Architecture-Based Approach to Self-Adaptive Software," *IEEE Int. Sys.*, vol. 14, no. 3, pp. 54–62, 1999.

[5] IBM Corporation, "An Architectural Blueprint for Autonomic Computing," IBM Corporation, Hawthorne, USA, Tech. Rep., 2005. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC Blueprint White Paper V7.pdf

[6] H. Giese and W. Schäfer, "Model-driven development of safe self-optimizing mechatronic systems with mechatronicuml," in *Assurances for Self-adaptive Systems*, ser. LNCS, J. Cmara, R. de Lemos, C. Ghezzi, and A. Lopes, Eds. Springer, 2013, vol. 7740, pp. 152–186.

[7] F. D. Macías-Escrivá, R. Haber, R. del Toro, and V. Hernandez, "Self-adaptive systems: A survey of current approaches, research challenges and applications," *Expert Systems with Applications*, vol. 40, pp. 7267–7279, 2013.

[8] J. S. Bradbury, J. R. Cordy, J. Dingel, and M. Wermelinger, "A Survey of Self-Management in Dynamic Software Architecture Specifications," in *Proc. WOSS*, 2004, pp. 28–33.

[9] D. Weyns, M. U. Iftikhar, D. G. de la Iglesia, and T. Ahmad, "A Survey on Formal Methods in Self-Adaptive Systems," in *Proc. C3S2E*, 2012, pp. 67–79.

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

[11] P. McKinley, S. Sadjadi, E. Kasten, and B. H. C. Cheng, "Composing Adaptive Software," *IEEE Computer*, vol. 37, no. 7, pp. 56–64, 2004.

[12] M. C. Huebscher and J. A. McCann, "A survey of Autonomic Computing – Degrees, Models, and Applications," *ACM Comp. Surveys*, vol. 40, no. 3, pp. 1–28, 2008.

[13] S. Dobson, F. Zambonelli, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, and N. Schmidt, "A Survey of Autonomic Communications," *ACM Trans. Auton. Adapt. Syst.*, vol. 1, no. 2, pp. 223–259, 2006.

[14] F. M. Roth, C. Krupitzer, and C. Becker, "Runtime evolution of the adaptation logic in self-adaptive systems," in *Proc. ICAC*, 2015, pp. 141–142.

[15] M. Güdemann, F. Nafz, F. Ortmeier, H. Seebach, and W. Reif, "A Specification and Construction Paradigm for Organic Computing Systems," in *Proc. SASO*, 2008, pp. 233–242.

[16] J. Kramer and J. Magee, "Self-Managed Systems: an Architectural Challenge," in *Proc. FOSE*, 2007, pp. 259–268.

[17] M. U. Iftikhar and D. Weyns, "Activforms: Active formal models for self-adaptation," in *Proc. SEAMS*, 2014, pp. 125–134.

[18] D. Sykes, D. Corapi, J. Magee, J. Kramer, and A. Russo et al., "Learning revised models for planning in adaptive systems," in *Proc. ICSE*, 2013, pp. 63–71.

[19] H. Nakagawa, A. Ohsuga, and S. Honiden, "Towards dynamic evolution of self-adaptive systems based on dynamic updating of control loops," in *Proc. SASO*, 2012, pp. 59–68.

[20] H. Tajalli, J. Garcia, G. Edwards, and N. Medvidovic, "PLASMA: A plan-based layered architecture for software model-driven adaptation," in *Proc. ASE*, 2010, pp. 467–476.

[21] N. Esfahani, A. Elkhodary, and S. Malek, "A learning-based framework for engineering feature-oriented self-adaptive software systems," *IEEE Trans. Softw. Eng.*, vol. 39, no. 11, pp. 1467–1493, 2013.

[22] I. Epifani, C. Ghezzi, R. Mirandola, and G. Tamburrelli, "Model evolution by run-time parameter adaptation," in *Proc. ICSE*, 2009, pp. 111–121.

[23] H. Prothmann, F. Rochner, S. Tomforde, J. Branke, C. Müller-Schloer et al., "Organic control of traffic lights," in *Autonomic and Trusted Computing*, ser. LNCS. Springer, 2008, vol. 5060, pp. 219–233.

[24] S. Tomforde, H. Prothmann, F. Rochner, J. Branke, J. Hähner et al., "Decentralised Progressive Signal Systems for Organic Traffic Control," in *Proc. SASO*, 2008, pp. 413–422.

[25] C. Krupitzer, F. M. Roth, S. VanSyckel, and C. Becker, "Towards Reusability in Autonomic Computing," in *Proc. ICAC*, 2015, pp. 115–120.

[26] S. Amir Molzam, A. Metzger, C. Quinton, L. Baresi, and K. Pohl, "Learning and Evolution in Dynamic Software Product Lines," in *Proc. SEAMS*, 2016, to be published.

[27] L. Pasquale, L. Baresi, and B. Nuseibeh, "Towards adaptive systems through requirements@ runtime," in *Workshop on Models@run.time*, 2011.

[28] L. Baresi, L. Pasquale, and P. Spoletini, "Fuzzy goals for requirements-driven adaptation," in *Proc. RE*, 2010, pp. 125–134.

[29] T. Vogel and H. Giese, "Model-driven engineering of self-adaptive software with eurema," *ACM Trans. Auton. Adapt. Syst.*, vol. 8, no. 4, pp. 18:1–18:33, 2014.

[30] D. G. De La Iglesia and D. Weyns, "Mape-k formal templates to rigorously design behaviors for self-adaptive systems," *ACM Trans. Auton. Adapt. Syst.*, vol. 10, no. 3, pp. 15:1–15:31, 2015.

[31] M. Puviani, G. Cabri, and F. Zambonelli, "A taxonomy of architectural patterns for self-adaptive systems," in *Proc. C3S2E*, 2013, pp. 77–85.

[32] V. Panzica La Manna, J. Greenyer, C. Ghezzi, and C. Brenner, "Formalizing correctness criteria of dynamic updates derived from specification changes," in *Proc. SEAMS*, 2013, pp. 63–72.

[33] J.Floch, S. Hallsteinsen, E. Stav, F. Eliassen, and K. Lund et al., "Using architecture models for runtime adaptability," *IEEE Software*, vol. 23, no. 2, pp. 62–70, 2006.

[34] D. Weyns, B. R. Schmerl, V. Grassi, S. Malek, R. Mirandola et al., "On Patterns for Decentralized Control in Self-Adaptive Systems," in *Software Engineering for Self-Adaptive Systems II*, ser. LNCS. Springer, 2013, vol. 7475, pp. 76–107.

[35] D. Garlan, S.-W. Cheng, A.-C. Huang, B. R. Schmerl, and P. Steenkiste, "Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure," *IEEE Computer*, vol. 37, no. 10, pp. 46–54, 2004.