Seminar SS 2011
Current Trends for Self-* Systems - Research Roadmap for Self-Adaptive Systems

Thomas Kühne

University Paderborn,
kuehne@mail.upb.de,
6382410
Supervisor: Masud Fazal-Baqaie

Abstract. Ultra-Large-Scale software systems or systems which are in a rapidly changing environment are difficult to manage manually at runtime, because of their huge complexity. Those systems should react autonomously to environmental changes like increasing load or changing goals to reduce the required administration effort and increase the service quality. With this so called self-adaption, the administration complexity is reduced, however the development challenges rise. This paper deals with the general research challenges and provides an overview of this domain.

Keywords: self-adaptive, self-management, autonomic computing, research roadmap

1 The need for Self-Adaption

Todays largest software systems are so called "Software-intensive" systems which usually have an increasing set of functionality with each product release. Therefore the future will lead to Ultra-Large-Scale (ULS) software systems which have the complexity of e.g. the whole internet. Besides the growing functionality especially ubiquitous systems e.g. mobile phones are facing a rapidly changing environment compared to traditional personal computers or server systems.

Both aspects - the huge complexity and a fast changing environment - cause a problem for the human actor managing the system at runtime [9, 8, 4, 14]. The consequences are increasing administration costs for enterprise systems and a lower acceptance and therefore sales rate of consumer products.

A solution to these problems is to automate the runtime management of those systems as far as possible in such a way that humans only need to provide high-level goals [9]. The system then cares for the details on how to achieve or maintain the goals by using the feedback from its own context at runtime. This behavior is called self-adaption or in some cases also self-management or autonomic computing. Although the approaches reduce the required maintenance effort at runtime they add a lot of additional complexity to the software system.

Therefore, the central issue of self-adaptive systems is to deal with the additional complexity at design time to enable the system solving those maintenance issues autonomously at runtime. This approach introduces many new questions like how to model and realize goals and how to change systems at runtime.

Based on the named challenge this paper presents a running example in Section 2 which points out the general idea of self-adaption. Next, the abstract problem and solution domain are described in Section 3. The problem domain explains the general terms used in the application area of self-adaptive systems whereas the solution domain describes the fundamental concepts used to realize self-adaptive systems. After that an overview of the research challenges and solution approaches based on the software development process are given in Section 4.

This seminar paper is based on reference [14] and [4]. The presented challenges and approaches have been selected based on those references but also on a broad analysis of the research field.

2 Running Example

The following example gives an idea of the problems to be solved in the area of self-adaption and also emphasizes the difference between normal requirements and self-adaptive requirements.
An intuitive example is the field of mobile phones where the primary purpose are obviously phone-calls. The administration might cover e.g. intelligent battery management and intelligent sound volume management. Both can be achieved manually e.g. the former by disabling the connection to the cell tower when entering the subway and the latter by reducing the volume manually when going to work. A self-adaptive solution seems evidently more user-friendly.

These additional requirements are different to the primary functional requirement: They require a change of the system's behavior in reacting to changes in the environment.

In the given example the mobile phone would have to check for the location via GPS to consequently change the power used for the antenna or the volume of the ringtone. Those checks also cost some amount of energy and therefore might be even conflicting with each other. Using the GPS module is really expensive in terms of energy consumption and therefore frequent checks reduce the battery life very quickly. On the one hand the autonomous battery management should reduce the number of checks but on the other hand the autonomous volume management must guarantee to be silent in certain locations e.g. at work. Otherwise the user would not trust the system and has to check the volume again manually which makes the self-adaption in this case pointless.

3 Domain Description

After motivating self-adaption informally this section will provide at first a definition of self-adaption. Furthermore, a description of the domain structure is

given to provide an understanding of the terms used in the field of self-adaption. Also the categories of goals are explained to define the possible range of self-adaptive systems. Finally, the central process which controls the self-adaption is elaborated.

3.1 Definition of Self-Adaptive Software

A fundamental definition of self-adaptive software is provided by [12]:

Definition 1. *"Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible."*

Furthermore a technical definition that focuses on the change of the processes at runtime is given by [14]:

Definition 2. *"A generic process model parameterized by graph constraints which define compatible structural models (customizers) as parameters of the process model."*

The conclusion of both definitions is that self-adaption is basically about changing the behavior of systems at runtime by modifying their configuration according to the goals of the software system.

Those definitions describe the general nature of self-adaptive systems. Nevertheless an explicit definition of what the difference to all the other software systems is has not been given yet.

The general differentiation between self-adaptive software systems and other software systems according to [14] is that self-adaptive systems are closed instead of open-loop systems. Open-loop systems or non-feedback systems [11] use only the current system state and their predefined model of the system to compute the output for a given input. Consequently those systems are not using any feedback from the environment to check whether or not the output fulfilled the goal according to the given input. Closed-loop systems instead are using feedback from the environment to improve their behavior.

This leads to another description of self-adaptive systems which is based on the source of changes in the software system [3]: Changing requirements are the cause for software evolution in usual software systems while a changing context is the cause for self-adaption.

3.2 Domain Structure

Based on the general definition of self-adaptive systems this subsection defines the terms in the domain of self-adaption according to [14]. The domain is divided into two main entities: The self-adaptive *system (self)*, which is the mobile phone in the running example, and the *environment* around the system, which is represented by the possible locations e.g. the office, subway etc., and actors using the system, which is just a single one in the example - the owner of the phone. System and environment are both the *context* of the system.

The system has objectives which are formulated as *goals* and is responsible for fulfilling the goals using self-adaption, which is the actual problem. Referring to the example the goals are described by the intelligent battery and sound volume management: The former goal is to save as much battery life as possible whereas the latter goal is to ensure always an appropriate noise level of the ringtone depending on the current location.

Furthermore, the system contains *sensors* to observe the context and *effectors* to apply changes to itself. Due to changes in the context detected by the sensors one or more goals may not be fulfilled anymore. This requires the self-adaption of the system using the effectors. A change in the environment occurs in the example when the location changes, e.g. when entering the office is being detected by the GPS sensor. Consequently, the goal stating that the sound volume should always have an appropriate noise level is not fulfilled any more and the mobile phone now has to reduce the volume through the volume control effector.

The system also has *knowledge* which allows it to reason about previous attempts to fulfill the goals and improve itself. As an example let us assume that the GPS sensor has not been accurate because the office is shielded by huge buildings which in turn sometimes resulted in a delayed reduction of the volume. The system decides therefore to reduce the volume already when approaching the office and overcomes the problem using this knowledge.

3.3 Categories of goals: Self-* Properties

The running example already gave a rough idea about the kind of goals a self-adaptive system might have and the previous section defined the basic terms. Nevertheless the range of possible goals is still vague and therefore this section provides a categorization of goals for self-adaptive systems.

Goals can be categorized into adaptive or self-* properties [9]. The following list represents well known properties according to [14]:

- **Self-optimization** has already been mentioned in the example as an intelligent battery management. The goal is to reduce the phones energy consumption by disabling or reducing all functions which are not used or can not be used at the moment. This might be the signal strength in the subway when there is no cell tower reachable. But the phone even might learn where a connection is possible and where not and based on this information trigger the check for new emails automatically before entering the subway.
- **Self-configuration** is based on the idea to automate the configuration of components. This kind of self-* objective may be applied in the mobile phone example as an "intelligent network discovery" module which tries to find routes to the next cell tower through other mobile phones when out of range by reconfiguring the network module. This enables the user to use his phone almost everywhere while the phone cares for the details of the network route.
- **Self-healing** deals with the identification of failures and the recovery. An example is the installation of a malfunctioning software component by the

user on the phone. This unintended functionality uses all the cpu time available and therefore would empty the battery within minutes. The self-healing module detects this as a malfunction and directly shuts down the component, searches for an appropriate software patch and updates the application.
- **Self-protection** is the "catch-all" objective which is responsible for everything which has not yet been covered by self-healing. If e.g. the identified patch in the previous example is incompatible with another one which makes self-protection prevent the setup to avoid any damage to the existing applications. Furthermore this objective covers the protection from attacks by applying the right counter measures.

Summarizing the goals, a self-adaptive system might improve its performance, configure components, recover from failures and protect itself from e.g. attacks. After this categorization of the goals the topic of the next section is how the system detects unfulfilled goals and how it reacts to that.

3.4 Domain Process: Adaption Loop

The result of an unfulfilled goal (reactive) or a change in the environment which may cause an unfulfilled goal (proactive) is the self-adaption of the system. This is in general a process which leads to changes in the system itself. According to the definition of the source of such changes in Section 3.2, which is not only the environment but also the system itself, the adaption causes again a change which might lead to another unfulfilled goal. Such a goal is again a trigger for self-adaption. Therefore, the whole self-adaption process can be modeled as a loop - the so called adaption loop [14]. In the context of autonomous systems, where the concept has been developed, the loop is called MAPE-K [9] which stands for Monitor, Analyze, Plan, Execute and Knowledge. The "MAPE" part is actually the same as the four phases in the adaption loop, while "K" explicitly refers to a common knowledge base.

In the following paragraph the four process phases of the adaption loop are described. The foundation is the "intelligent network discovery" extension of the running example in Section 3.2: The goal of the mobile phone is maintain a connection whenever possible. Therefore the system has a GPRS sensor to detect a reduction of the signal strength and a Bluetooth sensor to detect other phones in the area. Those other phones can be used to bridge a connection even when the next cell tower is too far away.

- **Monitoring**: The first phase is obviously observing the context based on the events from sensors which includes the environment and the system itself. Deviations from the desired system state are identified based on thresholds or through event correlation and are reported as symptoms. In the example the environment consists of the cell tower and all other phones in the area. Sensors are the GPRS module to monitor the signal strength to the cell tower and a Bluetooth module to detect other phones. In the scenario the signal strength drops because the cell phones distance to the cell tower increases which is reported as a symptom.

- **Detecting**: Upon this information the system needs to detect possible problems according to the goal and to check whether any reaction is required. In this case the result of the detecting process is a request for change. Referring to the example the system notices a continuously reducing signal strength. The conclusion is that a total loss of the connection is expected which will violate the goal because there have been other phones detected in the area.
- **Deciding**: In the third phase the system has realized that some action is required and evaluates the detailed decisions: What should be changed and how. In the example the phone consequently creates a list of all reachable phones ordered by their signal strength as an alternative route to the cell tower.
- **Acting**: The last phase is the utilization of effectors to adapt the behavior in the desired way. Applied to the example, the phone now tries to route the connection in the given order over another phone. This ensures that the connection will not be lost and the goal remains fulfilled.

Finally, the process starts over again according to the closed-loop property with the monitoring process checking whether the changes had the desired effect.

4 Research Challenges & Solution Approaches

The previous section explained the domain of self-adaption. Based on this foundation, this section describes on the one hand the research challenges and on the other hand the solution approaches. The presentation of both is based on the software engineering process.

Using the development process as the foundation is reasonable, because the design of self-adaptive systems affects the whole development process. Furthermore, the autonomic computing paradigm [9] specifies a clear separation of the system's implementation and the adaption loop mechanisms. However expressing such a separation is still a major research challenge in the development process [13].

Therefore, the overall challenge is to establish a systematic development of self-adaptive systems. It has to make the adaptive solution explicit along the software-engineering-process [4] and allows the design of re-useable, complex self-adaptive systems.

The first phase is the requirement analysis where the major issue is the definition of the requirements or in terms of self-adaptive systems the definition of the goals. The next phase is the system design which deals with adaptive architectures that change themselves to achieve the given goals. In the following implementation phase the specified functionality has to be implemented based on generic adaption engines which are introduced in this section. Finally, the system needs to be tested and verified which is in the last phase.

4.1 Requirements Analysis

Usually, the software engineering process starts with collecting the requirements. All requirements containing the expectation that the system should ensure them

autonomously at runtime need to be translated into goals. This is the first challenge [14]. Goals are the foundation for the system design towards the adaption loop as described in Section 3. Furthermore, a major problem when modeling goals are conflicts, e.g. performance and safety goals obviously often have contradictory objectives [7]. Those named challenges are already well known in the field of goal modeling. However, self-adaptive systems are additionally facing potentially unknown environments [4]. This results in uncertainty and incomplete requirement specifications. Consequently, this idea leads to the assumption that not all possible situations can be anticipated during the requirement analysis [14]. This causes another problem: It is therefore not possible to detect and resolve all conflicts at design time. Uncertainty can be explicitly introduced to relax goals which may help to reduce potential conflicts but conflicts will nevertheless occur. As a result, another idea in the field of self-adaptive systems is the use of goals as runtime objects [14] [4]. This is the foundation to enable the system to resolve conflicts between goals autonomously instead of manually at design time. The conflicts can be either resolved using simple goal prioritization or through optimization techniques using utility functions [14].

The challenges of goal modeling are now illustrated using the mobile phone example. This is the foundation for the description of the solution approaches in the following paragraphs.

The goals of the mobile phone contain a conflict between the battery management and the volume management. The battery usage is directly associated with a precise position. The mobile phone has to ensure the volume of the ringtone depending on the location information provided by the energy consuming GPS module. At the same time it also has to increase the battery life. Obviously those goals are interrelated: Increasing the accuracy of the volume management decreases the battery life and vice versa. An obstacle is the accuracy or availability of the GPS signal which also influences the position information and therefore the goal of the volume management. The uncertainty will be introduced when developing the example along the approaches.

Addressing the issues of translating requirements into goals and uncertainty the goal modeling approach KAOS [7] and the modeling approach for uncertainty, RELAX [16] have been combined [5] and are being presented in this section.

KAOS is a methodology for specifying a goal model and is independent from self-adaptive system approaches. The starting point are the high level goals derived from the requirements. The first step is a refinement of the goals and the identification of possible conflicts, obstacles posing a threat to goals and new goals resolving the obstacles or conflicts. Mapped to the example the requirements are on the one hand a mobile phone that has a long battery lifetime and an automatic location dependent volume control. These are translated into the goals: "The battery usage should be as low as possible" (G1) and the second one more specific into "The volume should be low in the office" (G2). Those goals are conflicting as described before. An obstacle for the volume management is an inaccurate or unavailable position information.

KAOS has no support for modeling uncertainty and therefore does not address the feature of self-adaption. Hence RELAX is used to extend the capability of KAOS which is described in the next paragraph.

RELAX is a modeling language based on natural language using a predefined vocabulary to have a precise formal semantic for uncertainty factors. The idea is to treat uncertainty not only as a problem but as a chance: Critical goals may remain un-relaxed while non-critical goals or non-critical aspects of goals may be relaxed. In the further development process un-relaxed goals are implemented as invariants while relaxed goals as variant. Especially important is that in this approach that it is impossible to anticipate every possible situation. Instead, in later design phases the system needs to deal with the incomplete definition and has to be very robust. Applying RELAX to the goals of the running example to reduce the conflict potential may result in the following goal definitions:

- G1: The battery SHALL be used AS FEW AS POSSIBLE.
- G2: The ringtone volume SHALL be reduced AS EARLY AS POSSIBLE AFTER entering the office and AS CLOSE AS POSSIBLE TO 2 minutes thereafter.

The goal G1 now explicitly states "AS FEW AS POSSIBLE" which remains almost identical to "as low as possible" but now have a precise formal semantic and can also be easier detected as such a relaxation. In goal G2 "AS EARLY AS POSSIBLE AFTER" and "AS CLOSE AS POSSIBLE TO" introduce a new relaxation which on the one hand reduced the need for a precise position information at any time and therefore makes it easier to achieve G1. On the other hand it reduces the threat potential of an inaccurate signal to G2 to some extend because there is now a timeframe in which the volume has be to reduced instead of the exact point in time when entering the office.

The combination of these approaches is already an interesting and promising approach but some questions remain open. An open research issue concerning KAOS and RELAX is a precise limitation of the goal relaxation to avoid uncertainty in cases where the goals are just too important for the system [5]. This may be quantified by a "risk" associated to the goals, but is still an open question and just a vague idea provided by the authors of RELAX. Also the idea of using optimization techniques which requires the definition of utility functions is not considered by KAOS and RELAX.

4.2 System Design

After defining the requirements through goals as described in the previous section the next question is how does the logical design of a system has to look like, that is able to autonomously manage these goals. Modeling adaptive software systems additionally requires the consideration of the adaption logic besides the application logic. The former executes the adaption loop and accesses the latter through its sensors and effectors. The primary overall challenge for self-adaptive systems is the separation of the design of those two enabling reuse,

simplifying maintenance and providing a clear separation of concerns [13]. The second challenge is to bridge the gap between the requirements analysis and the technical details as in the traditional system design. At first a high-level target architecture for the adaption loop is required, which then can be used as the foundation for a design process and to model the lower-level details in further steps. Challenging for architectures in huge systems is to minimize the impact on the system's performance, because a centralized adaption loop does not scale [4] [14]. The design process furthermore needs a modeling notation which supports the concept of the adaption loop [13]. UML is in general an appropriate language, but does not aid a clear separation of adaptivity aspects and the other functional parts of the system. It has no domain-specific semantics for the concepts of self-adaptivity. Due to the huge complexity of self-adaptive systems the mixture of the adaptivity- and system-logic makes it difficult to keep the concerns separated and to implement both properly. Having a clear design of the adaptivity, the system design itself is not any different compared to the usual design, but requires many design decisions which are specific to adaptive systems [14]. A question for example is what kind of component model allows the required amount of reconfiguration capabilities at runtime and should be therefore used.

Concerning the challenges this section presents at first the two fundamental high level design patterns for the adaption loop MAPE-K [9] and a three-layer reference model [10]. Based on them adapt-cases [13] as a design language for adaptive systems is described which supports the required separation of adaptivity and functional aspects. Finally, the open research questions are discussed.

The idea behind MAPE-K is to separate the adaption behavior from the system, but also implies an environment with many of those decentralized interacting adaptive systems. MAPE-K stands for Monitor, Analyze, Plan, Execute and Knowledge and defines a basic process to detect unfulfilled goals in the system and to adapt the system adequately. This has already been explained as the adaption loop in Section 3.4: the four process phases are directly derived from the four MAPE steps. MAKE-K additionally defines a shared Knowledge base to reason about previous changes and actions which is not explicitly mentioned in the adaption loop. The MAPE-K loop is embedded in an autonomic manager and associated to a so called managed element which is the system itself. Together they form an autonomic element being able to interact with other autonomic elements through their autonomic managers.

MAPE-K is in general a very high level approach not providing much more detailed concepts than described in the previous paragraph. The three-layer reference model is similar to some extend from a high level perspective but is a much more elaborated concept. The three-layer reference model in general tackles the problems of handling the complexity and enabling reuse, as MAPE-K does by separating the concerns of adaptivity and the rest of the system as well as the scalability bottleneck through decentralization. Furthermore, it reuses the existing architecture description languages (ADL) and thereby creates an

integrated approach down to the implementation. ADL solutions already provide capabilities for software configuration, deployment and reconfiguration.

The thee-layer reference model defines a layer each for goal management, change management and component control.

- Component Control Layer: Starting from the bottom the component control layer includes sensors, effectors and the adaption loop. Referring to MAPE-K the layer seems similar to the whole autonomic element although there is no explicit mapping mentioned by either approach.
- Change and Goal Management Layer: The other two layers are responsible for any problems occurring in the component control which can not be solved in the bottom layer itself. The change management layer then reports the problems to the goal management layer which creates a new plan and delegates the execution again down to the change management. In the reference model the assumption is that those exceptional calculations for new plans are very time consuming and may change the way how components interact. This may also have huge impacts on the whole system. The adaption mechanisms, which are in the component control are fairly simple in comparison to those in the goal management layer and change only parameters in the system.

To handle the scalability challenge the authors of the reference model tried to decentralize the whole system like MAPE-K. However, due to the centralized bus that is used to provide a reliable communication, the architecture does not scale. The communication bottleneck remains a research challenge which is discussed in the end of the section.

Having these two high-level approaches for the actual target system design at hand the answer to the question of how to model an adaptive system from scratch by using only the requirements still remains high-level and therefore vague. Hence, adapt-cases extends the well understood and accepted modeling notation of use-cases using UML profiles to provide a clear separation of adaption aspects and functional aspects of the system during the design process. The general approach is based on MAPE-K, but to reflect all aspects of the system it provides much more detailed semantics and syntax which are described in the next paragraphs. Also adapt-cases can directly build on the results of the previously presented combined approach of KAOS and RELAX. This closes the gap to the requirement analysis phase.

Adapt-cases add a distinction between normal use-cases and adapt-cases, which adapt the former and are derived from the goals specified in the requirement engineering phase. Adapting in this case means optimizing, protecting, etc. on a high-level as defined by the self-* properties in Section 3.3. The further refinement is based on the MAPE-K loop and divided into three main language packages: A monitor package which reflects the phases monitor and analyze, the adaption package that maps the plan and execute phases, and the adaption context package for the knowledge aspect. The monitoring package provides language concepts to model thresholds which trigger events. In the next step the adaption package allows to create corresponding actions, which are invoked by

those events. Actions may have multiple alternatives and the selection of those is modeled using OCL. The adaption context package contains a basic hierarchical component structure of the system which is used to link events and actions to system components. In the final step the system design is further detailed using sequence charts to specify the behavior inside the monitor and adaption package based on the adaption context package. Thus, adapt-cases overall provide a very complete and at the same time clearly separated model of the adaptivity.

The two presented high-level architectures and the design approach are a good foundation for the system design but some questions remain open which will be pointed out in the following paragraphs: The next paragraph deals with challenge of closing the gap between the requirement analysis and implementation phase, the one after with missing scalability of a centralized adaption loop.

First, the challenge of closing the gap between the requirement analysis and the implementation phase remains: Although this gap is closed through adapt-cases, as shown in a case study [13], there is still no systematic approach for this transition [13]. On the other end of the phase towards the technical solutions adapt-cases may not be able to represent all the possible features in the adaption frameworks and must be further investigated [13].

Second, scalability is still a major challenge: Systems modeled using one of the more detailed approaches (adapt-cases, three-layer reference architecture) will not be able to withstand the huge load in large systems up to ULS at runtime. Therefore, decentralized adaption-loops are required as proposed in the high-level approach MAPE-K. Such a decentralization results in further questions regarding the management of shared knowledge and inter-process communication [14]. Centralized communication connecting the decentralized components was an approach based on the three layer reference architecture but did not scale [10]. To overcome this problem the authors of the three layer reference architecture stated that such a system requires components working even with an inconsistent view of the overall context to reduce the communication overhead. The field of multi-agent systems already has a theoretical foundation for decentralized architectures, but it has not been implemented into real world scenarios and also needs to be integrated into a systematic development approach for adaptive systems [2].

Third, besides the scalability challenge, reducing the performance impact of sensors and effectors is also not solved yet: In large systems event correlation is used as a monitoring technique to handle the huge amount of single events [14]. Event correlation allows e.g. to aggregate subsequent error events into a single event or to remove duplicated events. However, it requires a detailed knowledge about previous events and is a research problem of its own [1]. Adapt-cases provide only support for modeling thresholds, which is only one possibility for event correlation, and therefore modeling other types remains an open issue.

Fourth, the complexity of ULS has to be handled by the system designer. The general approach, as already mentioned, is the separation of adaptivity related aspects from the rest of the system. Another research direction is to

enable the system to solve problems at runtime through machine-learning and thereby reducing the complexity at design time. As mentioned in the requirement analysis phase the idea of using utility functions in addition or instead of static goals is an approach in this direction. It especially helps to achieve real self-optimization in cases where a binary specification of the desired behavior in terms of desired and undesired goals is probably not applicable [10]. But utility functions are difficult to understand and therefore it is a great challenge to create an appropriate language. A key to such will be algorithms for preference elicitation that support the developer during the design process [15].

4.3 Implementation

In the previous section the questions related to the logical and technical design of the adaption engine and the system itself were discussed. Consequently, following the software engineering process the next question addresses the implementation of the adaptive system. Again the general idea is to keep the adaption engine separated from the system [6]. A central issue in this phase is therefore how such a generic adaption engine can be wired to a wide variety of software systems. The more detailed challenges are: First, a decentralized solution that can be applied to large scale systems. Second, the external adaption engine should also provide a range of sensors and effectors to reduce the implementation cost. Third, the solution must also provide capabilities to establish trust in terms of reliability and security of the system [4] [14]. This is essential for an autonomously acting software. Self-adaptivity also adds at lot of complexity to the system, which makes it much more difficult to trace errors: Finding the root cause of errors in a dynamically changing system is much more difficult than in static systems. Besides that the question of scalability remains on the technical level [4]: The overhead induced by monitoring may easily outweigh the performance benefit of the adaption.

This section presents the well known Rainbow Framework [6] as an approach to solve those challenges and ends with open research questions regarding the framework, and also the implementation phase in general.

Rainbow's major focus is on the separation of functionality as much as possible to maximize the reuse. Therefore, its centralized architecture is divided into two parts: the re-useable adaption infrastructure and the implementation-specific adaption knowledge. The latter contains the target system's architectural model and can be reused depending e.g. on the architectural style of two systems. The adaption infrastructure consists of three layers: A system layer with sensors and effectors, an architecture layer containing the execution engine and a translation layer between both that maps the architectural model elements to concrete representations of the target system. The adaption knowledge contains the operational model of the target system with parameters and constraints and is available at runtime. This circumstance allows the framework to check whether a change is valid according to this model and therefore addresses the challenge of reliability.

The framework also tries to extend the application area: Rainbow requires system access hooks in the target system for monitoring and adaption. Having this requirement met it can also be applied to legacy systems.

The open research issue is especially to achieve scalability and failover which is prevented by the centralized architecture [6]. Another issue which is not mentioned is the establishment of human trust. The design of effective monitoring approaches has also not been mentioned either, however they assume that specialized sensors will be implemented by external developers [6]. Therefore this remains an open research issue as well.

4.4 Test and Verification

The last phase is also the least focused phase by research [14]. After the implementation phase is finished the remaining question is whether the system behaves correctly. Because of the lack of solution approaches, this section presents only the challenges which are in the focus of the current research.

The first challenge has already been mentioned in the requirement analysis phase. The general assumption was that it is not possible to anticipate every possible situation which leads to incomplete requirement specifications [14, 4]. Consequently, it is not even clear what correct behavior in an unknown context is and therefore the term "desired" instead of "correct" should be used [4].

Another result of incomplete requirements is that probabilistic verification techniques are treated as a promising research direction [4]. The foundation for those techniques is a solid system model where all unknown attributes are added as probability distribution functions and where finally model checking techniques can be applied.

Using a sound model as a foundation might lead to an approach to prove every possible system state to be as desired. However, considering on the one hand large scale systems and on the other hand the amount of possible states compared to non-adaptive software that arises from self-adaption, the approach results in a state explosion.

This leads to the general assumption that a runtime verification of the changes may be a possible solution. Having performance as a key challenge for self-adaptive systems, which has already been mentioned in all previous design phases, this approach might also be difficult, because runtime verification will most likely cause an additional overhead. Nevertheless, this approach is already used in adapt-cases and the Rainbow framework.

Overall the verification of the system is especially relevant for safety-critical systems where also legal questions play an important role [4]: In case of failures caused by autonomous behavior the manufacturer will also be responsible.

The fundamental question which has to be answered besides the correct or desired functionality is: Does the system always reach a stable state after a single change has been triggered and no further external events occur? A stable state is reached when the system does not change itself again due to previous changes. Considering that changes may be the cause for other changes this question is important and not simple.

5 Conclusions

The goal of this paper was to provide an overview of the domain of self-adaption and the current research challenges and approaches. At first a domain description was presented with the common terms used in the research field of self-adaption and the fundamental concept of the adaption loop. Based on that an overview of the current research challenges, approaches and open issues along the different phases of the software development process was given. Due to the limited space only selected topics on a high level have been discussed and many research questions and solution approaches remain untouched. Nevertheless the selected challenges and approaches receive either a high attention in the research community (MAPE-K, KAOS & RELAX, Rainbow, the three-layer reference architecture) or are very new approaches (adapt-cases). Therefore the general research direction has been pointed out and the results may be used as a starting point for a further study.

Although there are popular and new research approaches which are based on each other the overall research field seems very cluttered and is difficult to analyze. Especially the unclear name of the research field might also lead to redundant research approaches: Depending on the authors Self-Adaption, autonomous computing, self-management and organic computing are sometimes used for the same research field and sometimes not [14]. Also the fundamental concept of the adaption loop, MAPE-K and the concept of control loops in general are all using often different terms for the same semantic entities which also causes an unnecessary confusion. For example the loop phases in MAPE-K and the adaption loop can be matched but have different names.

Overall there are also many open ends in the research field: A general problem is scalability which is required because self-adaptive systems should handle the complexity in ULS. The common idea is to tackle the problem using a decentralized adaption loop, but beside vague ideas, none of the investigated approaches had any kind of solution concerning this problem. Additionally, machine-learning concepts would allow the system to adapt to situations that have not been considered at design time. This would provide another layer of abstraction because developers do not have to consider every possible situation any more when using machine-learning. Finally another challenge which has not been considered by any of the approaches at all is human trust. It is definitely essential and without it the application area of self-adaptivity will be very limited: In the mobile phone example the user would have to reduce the volume manually at work or at least check it manually when he does not trust the system, which makes the adaptivity pointless.

References

1. Albaghdadi, M., Briley, B., Evens, M.W., Sukkar, R., Petiwala, M., Hamlen, M.: A framework for event correlation in communication systems. In: Proceedings of the 4th IFIP/IEEE International Conference on Management of Multimedia Networks and Services: Management of Multimedia on the

Internet. pp. 271–284. MMNS '01, Springer-Verlag, London, UK (2001), http://portal.acm.org/citation.cfm?id=645749.667569
2. Andersson, J., Lemos, R.D., Malek, S., Weyns, D.: Towards a classification of self-adaptive software systems. Computer 5525 (2009)
3. Caporuscio, M., Funaro, M., Ghezzi, C.: Graph transformations and model-driven engineering. chap. Architectural issues of adaptive pervasive systems, pp. 492–511. Springer-Verlag, Berlin, Heidelberg (2010), http://portal.acm.org/citation.cfm?id=1985522.1985547, 0
4. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software engineering for self-adaptive systems. chap. Software Engineering for Self-Adaptive Systems: A Research Roadmap, pp. 1–26. Springer-Verlag, Berlin, Heidelberg (2009), 104
5. Cheng, B.H., Sawyer, P., Bencomo, N., Whittle, J.: A goal-based modeling approach to develop requirements of an adaptive system with environmental uncertainty. In: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems. pp. 468–483. MODELS '09, Springer-Verlag, Berlin, Heidelberg (2009)
6. wen Cheng, S., cheng Huang, A., Garlan, D., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37, 46–54 (2004), 407
7. Dardenne, A., van Lamsweerde, A., Fickas., S.: Goal-directed requirements acquisition. Science of Computer Programming, 20(1-2):3–50 (1993)
8. Kephart, J.O.: Research challenges of autonomic computing. In: Proceedings of the 27th international conference on Software engineering. pp. 15–22. ICSE '05, ACM, New York, NY, USA (2005), http://doi.acm.org/10.1145/1062455.1062464, 201
9. Kephart, J., Chess, D.: The vision of autonomic computing. In: Computer. vol. 36, pp. 41 – 50. IEEE Computer Society (01 2003), 2579
10. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: 2007 Future of Software Engineering. pp. 259–268. FOSE '07, IEEE Computer Society, Washington, DC, USA (2007), http://dx.doi.org/10.1109/FOSE.2007.19, 229
11. Kuo, B.C.: Automatic control systems (6th ed.). Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1991)
12. Laddaga, R.: Self-adaptive software. Tech. Rep. 98-12 (1997)
13. Luckey, M., Nagel, B., Gerth, C., Engels, G.: Adapt cases: Extending use cases for adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'11) at the 33rd IEEE/ACM International Conference on Software Engineering (ICSE'11), Waikiki, Honolulu (USA). pp. 30–39. SEAMS '11, ACM, New York, NY, USA (May 2011)
14. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4, 14:1–14:42 (May 2009), http://doi.acm.org/10.1145/1516533.1516538, 96
15. Walsh, W.E., Tesauro, G., Kephart, J.O., Das, R.: Utility functions in autonomic systems. Autonomic Computing, International Conference on 0, 70–77 (2004)
16. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.H.C., Bruel, J.M.: Relax: Incorporating uncertainty into the specification of self-adaptive systems. In: Proceedings of the 2009 17th IEEE International Requirements Engineering Conference, RE. pp. 79–88. RE '09, IEEE Computer Society, Washington, DC, USA (2009), http://dx.doi.org/10.1109/RE.2009.36, 26