Goal-oriented Requirements Engineering for Self-Adaptive Service Compositions

Michael Meier
6278107
University of Paderborn
michaelm@mail.upb.de

Abstract. Self-Adaptation is a fundamental feature for service compositions in order to react to problems that come from their loose coupling. Due to the lack of modeling possibilities for adaptation in existing goal-based approaches, a new methodology is introduced here for specifying adaptation in existing goal models. Further, the approach offers continuous integration of adaptation throughout the whole development process allowing requirements traceability. This seminar thesis is based on the papers [3] and [4] by Luciano Baresi and Liliana Pasquale.

Keywords: Adaptation requirements, Goal model, Adaptive goals, Requirements traceability

1 Introduction

The amount of requirements for software is continuously increasing, making software more and more complex. Hence, the development of monolithic applications has become expensive and error-prone and is therefore not always practical anymore. As a reaction to this development, monolithic applications were split into several components making the software less complex again and thus decreasing the maintenance effort as well as reducing the development costs again. At the same time these components offer more flexibility as they can for example be reused in different departments of a company. To combine several independent components for accomplishing a mutual goal, a business process is created, which consists of several subprocesses calling the appropriate components. Components offering functionality for arbitrary processes are called services in that context. The services serving a business process to fulfill its goal are called a service composition.

Service Compositions A service composition exists when multiple services are connected in a service-oriented architecture being called by a business process to fulfill a mutual goal. The advantage of a service composition is that a service can be easily replaced by another service, making the whole business process more flexible. Also, these services are designed to be easily reused by different applications, lowering the development costs.
As an example we use an integration environment (fig. 1), which realizes a continuous integration business process to integrate source code to an executable software application by updating sources as well as building and testing them afterwards. This is realized by three called local services (VersionControl-, Build- and Test-Service).

Fig. 1. Integration environment with business process and service composition

However, the disadvantage of a service composition is that due to the distributed and loosely coupled nature, service compositions are unreliable compared to monolithic applications. The unreliability itself can be divided in physical faults, development faults and interaction faults [5]. Physical faults can occur, when communicating services are located on different hardware devices and there are network malfunctions. Development faults are situations, where services want to communicate with the help of predefined interfaces, which are incompatible to each other. At last, interaction faults can happen, when a service implementation is different from its requirement resulting in so called ”Quality of Service violations” or if a partner service has slow response times, which causes the calling service run into a timeout.

In order to prevent a failure of the whole business process in such situations, the failing service has to react accordingly. In the case of a failing partner service, the calling service can for example just call another remote service, which offers the same functionality (e.g. use the remote version control service instead of the local one as shown in fig. 1). In addition to failures, a service can also react to new or changing circumstances in its environment (e.g. when a test run is running too long, the process tries to acquire some additional remote test services to speed up). The reaction of a service in a case of a failure or other predefined circumstances is called self-adaptation.

Self-Adaptive Service Compositions One solution to realize self-adaptation is to define all possible evolutions (i.e. reactions to all possible events) at design-time. But obviously this is not always feasible as on the one hand, some requirements or features (of the operational environment) may be unknown at
design-time. And on the other hand, a definition of reactions to all possible events would result in an too complex design of the process.

So the more convenient solution is to formulate the adaptation as a fundamental requirement (along with the functional and non-functional requirements) and thus making the system recognize, when an event occurs where it has to react on. The adaptation requirement defines how the system can recognize such an event and how to react to this.

**Requirements Engineering for Self-Adaptive Service Compositions** Requirements can be specified within requirements models. In general, these models represent functional as well as non-functional requirements and are realized with (traditional) goal models, like **KAOS** [6].

In the approach of this paper, additional requirements are introduced to represent adaptation capabilities. As mentioned before, these requirements have to describe when to execute adaptation and what to change in that case. This new approach is not realizable with traditional goal models like **KAOS**, because they do not consider adaptation at all.

Although there are already existing approaches to model adaptation in goals specification [8] [11], they only list different strategies that can be performed and do not offer explicit support to the actual evolution of the system (i.e. check when to apply adaptation and enforce the actions regarding the adaptation).

To that reason, Baresi et al. [3] [4] have modified the existing **KAOS** goal model and extended it by adaptive goals in order to be able to explicitly define the "when" and "what" of the adaptation. Additionally they introduce a runtime infrastructure [2] on a more concrete abstraction layer, which uses these definitions to continuously check the "when" and enforce the "what". The former is determined by assessing satisfaction values of functional or non-functional requirements at runtime. The latter is defined by adaptation actions, which are triggered when that predefined satisfaction value of some requirement reaches a boundary. The adaptation action then consists e.g. of adding a new goal to the goal model or modifying an existing one. These two kinds of definitions have to be supported throughout the whole life cycle of the process, as the satisfaction values have to be assessed continuously to react accordingly at the specific moment, at which the adaptation is required to happen.

**Goal-oriented Requirements Engineering for Self-Adaptive Service Compositions** With the modified goal model, it should be possible to enforce requirements and adaptation from the design down to the execution and thus to allow a continuous support of adaptation in a system. To that purpose, three different levels of abstraction are introduced (fig. 2):  

1. **Requirements Analysis** On this most abstract layer, the traditional (functional and non-functional) as well as adaptation requirements are formally defined and refined in a goal model.
2. **System Design** On this intermediate and more concrete layer, the previously defined functional and non-functional requirements are built to an executable process represented in a functional model. Additionally the adaptation requirements are translated to supervision directives, which specify where to get the data from needed for assessing the requirements, how to ensure the continuous assessment of those and finally how to trigger the adaptation. This is represented in a supervision model.

3. **Implementation** On this most concrete layer, the runtime infrastructure is located, which executes the previously defined functional model and supervision model.

![Overall approach](image)

These three layers define the road-map of this paper and represent the following three main sections. Thereby, the main focus is on the requirements analysis section including the properties of the KAOS goal model, the modifications of it as well as the definition of adaptation requirements. The following system design section describes the properties of the functional as well as supervision model. At last, the implementation section briefly explains the role of that layer, as it only executes the two models from the specification layer which is discussed in detail in [2].

## 2 Requirements Analysis

On this most abstract layer (fig. 2), the traditional (functional and non-functional) as well as adaptation requirements are formally defined as goals in a goal model. For that purpose, the requirements are split into the definition of the goal and
the operationalization part representing the functionality behind the goal. It is abstracted from the concrete implementation and execution of these goals but instead the goals are refined into several subgoals and viewed isolated from each other.

At first, the requirements are defined in a traditional goal model. It uses refinement to create a hierarchical structure of goals and subgoals and formalization as well as operationalization for a detailed definition of the requirements. For the adaptation requirements, the traditional goal model is modified introducing fuzzy logic operators and membership functions to explicitly express adaptation. At last, the adaptive goals are expressed by defining adaptation strategies as well as adaptation actions.

2.1 KAOS goal model

A goal model is used to hierarchically structure goals and decompose them into several subgoals (fig. 3). In the same way, requirements can be used as goals and structured in that kind of model. In this paper the KAOS [6] goal model is used to describe requirements.

![Goal model of integration environment](image)

**Fig. 3.** Goal model of integration environment

**Refinement** There are two ways to decompose a goal into subgoals. The first is to create several conjoined subgoals. Then the satisfaction of the goal depends on all underlying subgoals (AND-refinement). The other way is to create alternative combinations of subgoals. This time the satisfaction of the goal depends on (at least) one subgoal (OR-refinement). The refinement of a goal into subgoals ends, when it can be operationalized (i.e. when the goal can be solved by a simple operation). This is true for all leaf goals.

In the example of the integration environment we have the overall goal *Integrate*, which consists of the four conjoined subgoals *UpdateSources*, *Build*, *Test* and *ShortIntegrationTime* (fig. 3). The parent goal is satisfied if and only if all four subgoals are satisfied.
Types of goals Besides the hierarchical structure created during refinement, goals have different types.

Concerning the functional view of the goal model, there are behavioral goals, which represent functional requirements, and soft goals, which represent non-functional requirements. The satisfaction of soft goals thereby depend on how behavioral goals are achieved.

Concerning the temporal view of the goal model, there are achieve/cease goals, which have to be fulfilled sometimes in the future/past (i.e. punctually), and maintain/avoid goals, which have to be fulfilled always in the future/past (i.e. permanently).

In the example (fig. 3), the three goals UpdateSources, Build and Test are behavioral goals, which have to be achieved sometimes in the future (i.e. update the sources at some time, build the sources and test the compiled binaries). At the same time, ShortIntegrationTime is a soft goal, which depends on the other three behavioral goals (i.e. ensure a short integration time which depends on the duration of updating the sources, building the sources and testing the compiled binaries). This soft goal has to be maintained the whole life cycle the system is running (i.e. always in the future).

Formalization All these kinds of properties characterizing different types of goals can and have to be formalized, so that the satisfaction of these goals or requirements can be evaluated live at runtime of the system.

In the KAOS goal model, goals are formalized by using properties of Linear Temporal Logic. These properties are represented by traditional logic operators. They all return a binary value (true or false) representing, whether the requirement is satisfied or not. Such requirements, which are defined by a binary value, are called crisp requirements.

Some of the LTL properties are:

Sometimes in the past (♦ φ) Requires that φ has been true at some point in the past.

Sometimes in the future (◇ φ) Requires that φ must be true at some point in the future.

Always in the past (□ φ) Requires that φ has been true at all points in the past.

Always in the future (◊ φ) Requires that φ must be true at all points in the future.

In the previous state (● φ) Requires that φ has been true in the previous state.

In the next state (○ φ) Requires that φ must be true in the next state.

With the help of these properties, crisp requirements can be formalized, which is necessary to be continuously assessed at runtime. The requirements, which can not be formalized with the help of these properties, are introduced in the section concerning the modified KAOS goal model.
Goal | Formal definition
---|---
G1.1 ♦ (ip.state = initialized) ⇒ ♦ t<T\text{update} (ip.state = sourcesUpdated)
G1.2 ♦ (ip.state = sourcesUpdated) ⇒ ♦ t<T\text{build} (ip.state = buildFinished)
G1.3 ♦ (ip.state = buildFinished) ∧ ip.buildSuccessful ⇒ ♦ t<T\text{test} (ip.state = testFinished)
G1.4 □ (t ≤ T\text{integration}|T\text{integration} = 2h)

Table 1. Formal definition of goals in KAOS goal model

The formalization of the goals in the running example is shown in table 1. Goal 1.1 is e.g. formally defined, that when the integration process \(ip\) has the state \(initialized\), then at some point in the future (within a given amount of time) it must be in the state \(sourcesUpdated\). Goal 1.2 and 1.3 are defined the same way only with the difference, that goal 1.3 is only executed, when the build was successful. Soft goal 1.4 has to maintain an integration time, that has to be lower than two hours in this example.

Before the lacks of this kind of formalization are explained, another kind of formalization is introduced, which further describes how these goals are achieved and when they may be executed.

**Operationalization** [1], [10] is the method to semi-automatically infer a set of operations from the formal definition of one or more goals we just introduced. In fact, it formally describes the actions contained in a (leaf) goal, where refinement stops (i.e. it can be operationalized). It is needed to link the formally defined goals with the actually executed operations (i.e. enforce requirements traceability) as well as generate functional and supervision models, which are needed by the runtime infrastructure to actually execute the operations and continuously assess their satisfaction.

Besides the name of the operation, it is described by input and output variables (e.g. the integration process variable \(ip\) in table 1) as well as several conditions, which must hold at specific states. These are the following:

- **ReqPre** Required preconditions define the application states in which the operation is allowed to be performed.
- **TrigPre** Triggering conditions define the application states that activate the operation execution.
- **ReqPost** Required postconditions define additional conditions that must be true after the execution of an operation.

For the running example of the integration environment, the operationalization of the leaf goal Build represented by the corresponding operation Build uses the variables integration environment \(ie\) and integration process \(ip\) as input and output. It is triggered by the method \(startBuild()\) (TrigPre) if and only if the
integration process is in the state *sourcesUpdated* (ReqPre). Afterwards it must be in the state *buildFinished* (ReqPost).

**Drawback** As mentioned before, one drawback of the *KAOS* goal model is that it only supports *LTL* properties, which can only express crisp requirements (those returning a binary value). This way it can only be assessed, if the requirement is satisfied or not, leaving out the corresponding satisfaction (resp. violation) level. But this measure is important for fuzzy requirements/goals that rely on specific quantities.

For example the soft goal *ShortIntegrationTime* from the running example, which requires, that an integration process may not last longer than two hours, is true when the running time is less than two hours and false otherwise. In this case, *LTL* properties do not make a difference between the process running for ten minutes or running for one and a half hour. Obviously the first one is "better" and farther from violating the requirements than the latter.

Additionally, if we want to introduce adaptation to this process, it is good to know, what the exact satisfaction level is. Probably we do not want to adapt anything in the first case but instead we want to react in the second case, before the process violates the requirement, because then it is too late to adapt anything (i.e. the requirement will remain unsatisfied).

Another drawback of the *KAOS* goal model is that it does not support the definition of adaptive goals at all. These facts lead us to the modification of the *KAOS* goal model by Baresi et al.

### 2.2 Modified KAOS goal model

Due to the drawbacks mentioned in the previous section, Baresi et al. have introduced several modifications and additions to the *KAOS* goal model to solve these issues.

**Fuzzy logic operators** To solve the issue of not being able to assess the actual satisfaction level for fuzzy requirements, the authors introduce Fuzzy Logic operators, which were already used in RELAX [12] to define requirements. These are the following:

- **AS EARLY|LATE AS POSSIBLE ϕ**: Defines for temporal quantities ϕ, that they should occur as early (resp. late) as possible - i.e. the satisfaction is higher if ϕ occurs earlier (resp. later).
- **AS CLOSE AS POSSIBLE TO q ϕ**: Assesses the proximity of quantities or frequencies ϕ to a certain value q - i.e. the satisfaction is higher for values of ϕ closer to q.
- **AS MANY|FEW AS POSSIBLE ϕ**: Defines for quantities ϕ, that there should be as many (resp. few) as possible - i.e. the satisfaction is higher if the value of ϕ is higher (resp. lower).
The benefit of these operators is, that two states, which could not be distinguished for crisp requirements, can now be differentiated. It is now possible to express fuzzy requirements, which return a concrete satisfaction value instead of a binary value.

So in our example, the process running ten minutes has a higher satisfaction value compared to the one running for one hour and a half (regarding the requirement to run less than two hours). Now goal 1.4 can be expressed as followed:

\[ G1.4 : \textit{AS EARLY AS POSSIBLE} t_{\text{Integration}} \]

To unify the satisfaction value \( x \) of a fuzzy requirement (i.e. \( x \in [0, 1] \) for all goals), another component is introduced.

**Membership functions** Membership functions map an arbitrary value \( x \) of a Fuzzy Logic operator to the default interval \([0, 1]\). Therefore, each fuzzy requirement has a custom membership function, returning a value between 0 and 1 for all input values (\( \forall x : f(x) = [0, 1] \)).

The soft goal 1.4 representing a fuzzy requirement is assigned the membership function shown in fig. 4. For an integration time of 0 the goal has the satisfaction value 1. When lasting one hour, the the satisfaction value of the goal is only 0.7. And when finally reaching the boundary of two hours, the satisfaction value reaches 0.

![Fig. 4. Membership function for fuzzy goal G1.4](image)

**New possibilities** With the modifications of the KAOS goal model (i.e. the use of fuzzy logic operators and membership functions), each goal representing a fuzzy requirement now has a detailed satisfaction level. At this point we assume, that all these satisfaction levels are updated dynamically during runtime of the system. This makes it now possible to maintain the goal model live at runtime, i.e. we can continuously assess the requirements. With this "live" goal model, we can now activate adaptation when needed.
2.3 Adaptive goals

Adaptation is a reaction of a system to certain changes or observations in an environment. When in our example the system observes that the integration process in running more than one hour and a half, we may want to react to this situation. This reaction is defined with the help of adaptive goals, which were introduced by Baresi et al. for the modified KAOS goal model.

Adaptive goals consist of two parts. On the one hand, they represent adaptation strategies in the goal model. These adaptation strategies define, when adaptation should be applied. This moment is depending on the satisfaction of crisp or fuzzy goals. On the other hand, they are able to trigger suitable adaptation actions. These actions define, what to change when that adaptation is trigger (e.g. add or remove a goal in the goal model, modify an operation or membership function).

Adaptation strategies Adaptation strategies define, when an adaptation should be applied (i.e. the moment a specific strategy should be trigged). For that purpose, an adaptive goal is associated with a set of membership functions. These membership functions represent different adaptation strategies that can be applied (e.g. one for normal load and one for high load).

Compared to a fuzzy goal, which ”only” has one membership function mapping a custom value or expression to a satisfaction value, adaptive goals now map each satisfaction value of that fuzzy goal to multiple membership values (one for each membership function). The strategy (resp. membership function) having the highest membership value for a given satisfaction value is the best strategy to use in that specific state of the application. Thus, if the currently activate strategy has the highest membership value, nothing is done. But if the highest membership value is returned by another strategy, adaptation has to be applied (i.e. the adaptation actions of that strategy have to be executed).

Fig. 5. Membership function for adaptive goal
For our example we introduce an adaptive goal for the soft goal 1.4, that reacts, if the integration is running a specific time, to prevent it from exceeding the two hour boundary. This adaptive goal is assigned the membership functions shown in fig. 5. So the soft goal is "starting" at the integration time 0 and thus has a satisfaction value of 1 at that point (compare to fig. 4). For the satisfaction value of 1 the system now checks the membership values of the different strategies (fig. 5). The strategy Normal has a membership value of 1 at that point while the strategy High has a value of 0.5. The two other strategies both have a value of 0 at that point. Comparing these four different membership values the strategy Normal has the highest one meaning that this strategy is triggered if it is not already active. For an increasing integration time the satisfaction value is continuously decreasing (fig. 4). Passing the satisfaction value of 0.7, the strategy Normal now has a membership value < 0.5 while the strategy High has a constant membership value of 0.5 (fig. 5). This has the effect that at this point the strategy of the adaptive goal is switched from Normal to High activating the corresponding adaptation of that strategy. The next point for triggering another strategy is after the satisfaction value drops blow 0.4. At that point the membership value of the strategy Maximum is > 0.5 while the one of strategy High is still at its constant value of 0.5. Finally, the last point for switching strategies is, when the membership value drops below 0.1, because at that point the strategy Violation has a higher membership value than all the other strategies.

**Adaptation actions** The adaptation actions define what to change in the goal model in the case that adaptation is triggered. Adaptation actions can change the goal model in different ways. They can add or remove goals, modify the definition of goals (e.g. change the membership function) or just add, remove or modify single operations.

The adaptation actions for the strategies in the example could be, that while applying the Normal strategy the integration process may only use one service, whereas triggering the High strategy it may occupy 25 percent of all free services (and 50 percent with the Maximum strategy). This is done by modifying the operations Build and Test to use more servers. Shortly before reaching the two hours boundary, the Violation strategy is triggered, assuming that the currently running process will exceed the limit. Therefore the membership function of the fuzzy goal is modified in a way, that it is lowered by a specific value forcing the previous strategies to occur earlier next time.

Further, adaptation actions have two types of temporal properties. On the one hand, there are permanent adaptations, which exist in the goal model until they are possibly changed by another adaptation. When for example the first version control service is down (fig. 1), the system adapts to use another one. This persists permanently until the adaptation may observe later on, that the first one is available again and activate the adaptation again.

On the other hand, there are transient adaptations, which only exist as long as the adaptation action is active. When for example the integration process
interrupts at some state during building for some reason, a transient adaptation
could be activated to restart that process at the last safe state. After returning
and starting the process again in this state, this adaptation is no longer persistent
in the system.

In this section the KAOS goal model was introduced as well as the modifi-
cations to allow the specification of adaptive goals. In the next section, a more
concrete layer is introduced, which uses the formalisms introduced in this section
to build an executable process of the goals and to define a supervision directive,
which supports the mechanisms needed to observe the system for triggering
adaptations.

3 System Design

On this intermediate and more concrete layer (fig. 2), the previously defined
functional and non-functional requirements are built to an executable process
represented in a functional model. Additionally the adaptation requirements are
translated to supervision directives, which specify where to get the data from
needed for assessing the requirements, how to ensure the continuous assessment
of those and finally how to trigger the adaptation. This is represented in a
supervision model.

3.1 Functional specification

The functional specification is a more concrete representation of the stated goals
and requirements in the top layer. It is represented by a functional model de-
scribing a set of abstract processes, which are able to achieve the objectives
stated in the goal model. These abstract processes are directly mapped to the
goals they are representing to provide continuous traceability of the satisfaction
of the goals. The abstract processes are ordered in an executable process (i.e.
the workflow), which can be derived out of the pre- and postconditions which
were defined during the operationalization step.

Operations must be executed in sequential order, i.e. operation Op1 must
be executed after operation Op2, if and only if its preconditions depend on the
postconditions of Op2. This means for example, that the operation Test must
be executed after the operation Build, as the preconditions of Test require, that
the build has finished, which is a postcondition of the operation Build.

Operations can be executed in any order, i.e. operation Op1 can be executed
before or after Op2, if and only if the preconditions of the two operations do not
depend on the postconditions of the other ones.

These rules allow to build an executable process out of the operationalization
of our integration environment example as shown in fig. 6. The methods Update-
Sources, Build and Test must be executed sequentially to reach the last state, as
the preconditions of Build and Test depend the postconditions of UpdateSources
and Build.
### 3.2 Supervision directives

The supervision directives are components which observe and apply the adaptation requirements. They are contained in a supervision model, which task is to continuously measure and update the satisfaction levels of the goals. This is done by continuously collecting data from the running system, using them for monitoring the satisfaction of the requirements and finally applying adaptation when needed (fig. 7).

**Data collectors** Data collectors retrieve information about the actual behavior of the system and provides monitoring components with the necessary information to evaluate the goals’ satisfaction.

The data can be collected from various sources, such as external sensors (e.g. temperature sensor of the cpu), process internal variables (e.g. the state variable of the integration process), messages exchanged with partner services (e.g. the integration process calling the `startTest()` method of the test service to start the test process) or historical data collected with previous process executions (e.g. the average duration of the last ten integrations).

Further, there are three different delivery methods to provide the monitors with the retrieved data - push, periodic push and pull. With the push method, the data is sent to the monitor by the data collector as soon it is collected, measured, calculated or changed (i.e. when the amount of free servers changes, the value is pushed). With the periodic push method, the data is sent to the monitor every specified time frame (e.g. send the duration of the current integration to the monitor every minute). At last with the pull method, the data is sent by the data collector when requested by the monitoring component (e.g. the integration duration has just passed a predefined boundary so the amount of the free servers is requested).

**Monitors** Monitors calculate the degree of goals’ satisfaction by evaluating the LTL expressions and membership functions of the corresponding goals. To
achieve this, each monitor works with one or more data collectors (e.g. retrieve data pushed by the data collectors to them or request data from them) and afterwards checks the requirement’s satisfaction with the retrieved data in particular states of the process. These states are manually selected and represent safe states, in which adaptation could be applied. During evaluation of the satisfaction in a safe state, the process is blocked in whole to prevent it from leaving the safe state, since when a monitored goal is violated, adaptation has to be applied. In that case, the monitor triggers the adaptation through an adaptor.

**Adaptors** While a monitor decides the point in time, when an adaptation has to be trigged, an adaptor knows, what to do in that case. So as described above, the monitor triggers the adaptor in case of adaptation. The adaptation actions specified in the adaptive goals are translated into lower level actions, which are executed by the adaptor.

In this section the functional and supervision model were introduced, which represent an executable process of the stated functional and non-functional goals as well as a plan how to execute the adaptive goals. In the next section, the implementation is introduced, which finally executes the abstract models.

## 4 Implementation

On this most concrete layer (fig. 2), the implementation is located, which executes the previously defined functional model and supervision model. The former is executed by an execution engine and the latter by a supervision infrastructure.

The execution engine is running the concrete application instance and provides the suitable infrastructure to execute the functional model defined in the layer above.

The supervision infrastructure executes the data collection directives and applies the monitoring and adaptation activities defined in the supervision model.

## 5 Conclusion

In this paper an approach with three layers was introduced to explicitly support adaptation in requirements engineering for self-adaptive systems. In the Requirements Analysis a goal model is defined containing the functional as well as the adaptation requirements of the system. In the System Design an executable process and a plan to execute adaptation is generated out of the stated functional and adaptive goals. The Implementation layer finally executes the process and adaptation plans.

With this approach the adaptation is defined and executed in a single infrastructure allowing a continuous integration of adaptation and requirements traceability. This is the big advantage of this approach, as others using different technologies and approaches in the different layers have to transform the results
of the one layer to be used in the next one. By doing this, the requirements traceability between the layers is lost. Another positive thing about this goal-oriented approach is the intuitive way of modeling adaptation by using membership functions representing different adaptation strategies.

The disadvantages however are, that this three-in-one approach does not reuse other approaches and technologies already established for self-adaptive systems. With MAPE-K [9] there already exists an approach to define a plan how to execute adaptation and the Rainbow Framework [7] is an already established implementation of the third layer. Instead of using them, own approaches for the system design and implementation layer are defined, which are in an early state so far. Further, the opportunity to define a test and verification layer in this three-in-one approach to extend it to an four-in-one approach was omitted.

References