# Software Engineering Processes for Self-adaptive Systems

Jesper Andersson[1], Luciano Baresi[2], Nelly Bencomo[3], Rogério de Lemos[4], Alessandra Gorla[5], Paola Inverardi[6] and Thomas Vogel[7]

[1] Department of Computer Science, Linnaeus University, Växjö, Sweden
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano, Italy
[3] INRIA Paris, Rocquencourt, France
[4] University of Kent, UK
[5] Faculty of Informatics, University of Lugano, Switzerland
[6] Dipartimento di Informatica, Università dell'Aquila, Italy
[7] Hasso Plattner Institute at the University of Potsdam, Germany

**Abstract.** In this paper, we discuss how for self-adaptive systems some activities that traditionally occur at development-time are moved to run-time. Responsibilities for these activities shift from software engineers to the system itself, causing the traditional boundary between development-time and run-time to blur. As a consequence, we argue how the traditional software engineering process needs to be reconceptualized to distinguish both development-time and run-time activities, and to support designers in taking decisions on how to properly engineer such systems. Furthermore, we identify a number of challenges related to this required reconceptualization, and we propose initial ideas based on process modeling. We use the Software and Systems Process Engineering Meta-Model (SPEM) to specify which activities are meant to be performed off-line and on-line, and also the dependencies between them. The proposed models should capture information about the costs and benefits of shifting activities to run-time, since such models should support software engineers in their decisions when they are engineering self-adaptive systems.

## 1 Introduction

Traditional software engineering research primarily focuses on development activities for high-quality software, rather than maintenance or evolution [29]. Meanwhile, the software engineering community has accepted that software must continuously adapt and evolve according to ever changing requirements to remain useful for the user [26, 27]. This awareness has led to iterative, incremental, and evolutionary software engineering processes [4, 9, 19, 20, 23, 35, 43], rather than strictly sequenced phases of requirements engineering, design, implementation, and testing, as perceived by the *waterfall model* [38].

However, such approaches to change software do not meet the requirements of many modern context and self-aware, mission-critical, or ultra-large-scale software systems [17, 31]. Context and self-aware systems require *timely* changes in response to changing environments, changes in the system itself, or in its goals. The inherent delay in traditional change processes is for these systems unsatisfactory. Mission-critical systems have to operate continuously. In traditional change processes, changes are deployed during scheduled down-times and, as a consequence, continuous operation is not possible. Ultra-large-scale systems are highly complex, which makes human-driven change activities difficult and expensive, or even infeasible in practice due to the size and inherent complexity that impede a complete shutdown of the system in order to change it. Thus, we may conclude that using a traditional change process for these kinds of systems holds the risk of the system failing to meet its specification with respect to timely reaction to changes and continuous operation.

These risks have led to the development of novel means of risk mitigation that change software in terms of *self-adaptation* [14]. Self-adaptive behavior implies that certain development and change activities are shifted from development-time to run-time, while reassigning the responsibility for these activities from software engineers or administrators to the system itself. The new time-of-change timeline [11] that is, when a change takes place, covers development-time, deployment-time and run-time. A consequence of this reconceptualization of the time-of-change is that the traditional boundary between development-time and run-time blurs, which requires a complete reconceptualization of the software engineering process [3, 7, 21, 22], where the traditional perspective that separates development-time and run-time is revisited. The rationale is that for self-adaptive software systems the typical software-process mapping from (1) *software life-cycle phases* [32] (e.g., development, deployment, operation, or evolution) and (2) *software engineering disciplines* [32] (e.g., requirements engineering, design, implementation, verification, etc.), and software-process activities (e.g., elicitation, prioritization and validation of requirements), onto a, (3) *time-of-change timeline* is not valid anymore. As an example, the disappearing boundary between development-time and run-time does not allow software changes to be decoupled from the running system anymore. Buckley et al. propose a taxonomy with a number of dimensions to classify software change [11]. Even though the proposed taxonomy appears to be sufficiently comprehensive, we argue that a more fine-grained perspective on the timeline is required for the class of self-adaptive software systems. This new perspective corresponds to a new dimension, which considers the blur and describes where, with respect to the self-adaptive system, change activities take place. With respect to the self-adaptive system, we refer to activities performed externally as *off-line activities* and to change activities performed internally as *on-line activities*[8].

The main contributions of this article is the identification and description of a number of challenges related to the required *reconceptualization of software engineering processes for self-adaptive systems*, and its impact on how to engineer such systems. We propose software process modeling as a corner-stone

[8] The notions of on-line and off-line we introduce are distinct from their traditional notions in the field of algorithm theory.

component to be used in a modeling and evaluation framework for self-adaptive software systems.

A reconceptualization of software engineering processes requires (1) *the definition of abstractions for off-line and on-line activities, the identification of the entities subject to change by these new activities and the dependencies in-between such activities* and, as a consequence, (2) *the complete understanding of the impact of design decisions related to off-line activities over on-line activities, and vice versa.* We argue that engineering self-adaptive software is about defining a software process which defines the scope for a system's self-adaptive behavior (by means of on-line activities) and proposes new variants to the mapping afore mentioned. However, scoping introduces a number of additional challenges.

We address these challenges with our second contribution, *process modeling for engineering self-adaptive software systems.* We propose an approach based on the *Software & Systems Process Engineering Meta-Model Specification* (SPEM) [32]. Explicit processes models manifest *how* a system is developed and evolved, which promotes a better understanding of self-adaptive software systems and the relationships to software engineering processes. In addition, it promotes communication, reuse, and reasoning that may in the long-term be automated [33]. Altogether this improves support for comprehension and decision making when engineering self-adaptive systems. The adoption of process modeling for self-adaptive software systems engineering involves a number of challenges: (1) The *new concepts* (on-line and off-line activities, and their dependencies) *have to be supported by the process modeling language.* (2) Likewise, *concepts and models must capture the relative value of activities (costs & benefits)* since they influence the scoping of a self-adaptive system's activities. Due to dependencies between the engineering process (off-line activities) and the self-adaptive system (on-line activities), we must develop and use (3) *new integrative design and modeling* techniques [6] that support both, process and product engineering. These techniques must support designers in scoping the self-adaptation mechanism, decisions that should be based on *value* and balance costs and benefits.

The remainder of this article is organized as follows. In Section 2, we introduce an illustrative example, which is used to define and motivate the problems this article is targeting. Section 3 discusses a number of challenges related to the problems in more depth. Based on these discussions, we outline a software process modeling approach for engineering self-adaptive software systems in Section 4. In Section 5, we discuss important challenges that remain to make process modeling a viable tool in self-adaptive software systems engineering. We discuss related work in Section 6, and conclude and outline a research agenda for the proposed direction in Section 7.

## 2 Revising the System Life-cycle

In this section, we introduce a specific perspective concerning self-adaptive software systems, which is the characterization of their life-cycle and the challenges

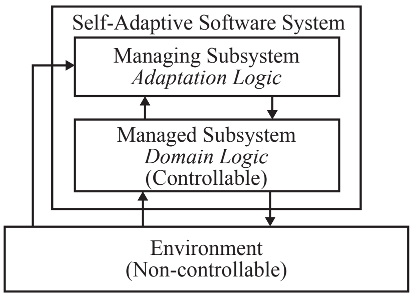


**Fig. 1.** A Conceptual Architecture for Self-Adaptive Software Systems [1]

to support this life-cycle by effective software processes. With self-adaptive software systems as our focal point, we revisit and revise the typical software system's life-cycle, pin-pointing explicit and implicit relationships between software engineering processes and a deployed self-adaptive software system.

The essential characteristic of a self-adaptive software system is its ability to autonomically evolve and adapt its behavior dynamically in response to changes to system requirements, the system itself, or the system's operational environment. The evolution and adaptation mechanisms should preserve the essence of the system behavior by continuously providing an acceptable implementation of the system's core requirements. The scope of this paper does not require a distinction between evolution and adaptation, the focus is whether *software changes* are enacted off-line or on-line and, as a consequence, a precise definition of software evolution and software adaptation is beyond the scope of this paper.

To frame evolution and adaptation mechanisms, Figure 1 depicts a conceptual architecture for self-adaptive software systems [1]. An important property of this architecture is the disciplined split, which promotes separation of concerns. A *Managing Subsystem* implements the *Adaptation Logic* that manages a *Managed Subsystem*, which implements the *Domain Logic*. The adaptation logic implements a control loop in line with the *monitor-analyze-plan-execute* (MAPE) loop [24], which evolves and adapts the domain logic. Domain functionality and the system's core requirements are implemented by the domain logic. The self-adaptive system operates in a non-controllable environment that may merely be observed by the adaptation logic, while the domain logic may both observe and affect the environment. Moreover, the architecture allows for additional managing subsystems that adapt adaptation logic in other managing subsystems. This may be used to describe, for example, the goal management layer in Kramer and Magee's layered architecture [25] or hierarchical control in autonomic computing [24]. This characterization of self-adaptive systems promotes evolution and adaptation mechanisms realized by the adaptation logic to first class computations that have to be supported side by side with domain logic computations, while seamlessly interleaving with the running system.

In a typical software process for conventional software systems, evolution and adaptation are performed after the initial development and deployment, and they usually encompass all process disciplines from requirements engineering to deployment. This indicates that timely changes are not prioritized. In a self-adaptive software system the situation is different. The evolution and adaptation activities realized by the adaptation logic are in place because requirements change frequently and timely reactions are essential. Self-adaptive software systems autonomically perform activities on-line and at run-time that originally have been carried out manually and off-line. This change to a process does not affect all process activities, and thus, understanding relationships and dependencies between (off-line) software process activities and (on-line) evolution and adaptation activities is a great challenge. Before systematically analyzing challenges concerning life-cycles and software processes for self-adaptive systems and the engineering of such systems, we present an illustrative example that we use throughout paper.

## 2.1 Illustrative Example: Automatic Workarounds

Automatic Workarounds (AW) is a technique that enhances applications with self-adaptation capabilities that deal with functional failures at run-time [12, 13]. When applications fail, either because there is a fault in the application itself or in one of the libraries used by the application, the AW technique attempts to mask the fault and thus, to avoid the corresponding failures, while providing the core domain functionality.

The technique is based on the hypothesis that software systems usually offer several "equivalent operations" that provide the same core functionalities in different realizations. If an operation fails, the AW mechanism exploits this intrinsic redundancy to automatically find workarounds and apply an alternative equivalent sequence of operations. Consider for example a container component that implements one operation to add a single element, and one operation to add several elements. To add two elements, it is possible to add either one element after the other, or to add them both at the same time. If one of these options causes a failure at run-time, the AW technique attempts to execute the equivalent sequence of operations as the alternative option to mask the fault.

We depict the AW technique's principal concepts and mechanisms in Figure 2. An application component invokes an operation provided by another component (*caller* and *called component*, respectively). If an invocation causes a failure, the AW technique implemented by the adaptation logic handles this failure at run-time by first looking for a sequence of operations which is equivalent to the failing invocation. Having found an equivalent sequence of operations, the adaptation logic enacts an adaptation that automatically invokes this sequence of operations. If the alternative execution does not cause a failure, a successful workaround has been found and the application proceeds as if the original failure never occurred. Otherwise, the adaptation logic continues testing other equivalent sequences until an alternative is either successfully executed or until all equivalent sequences have been tested unsuccessfully. In the latter case,

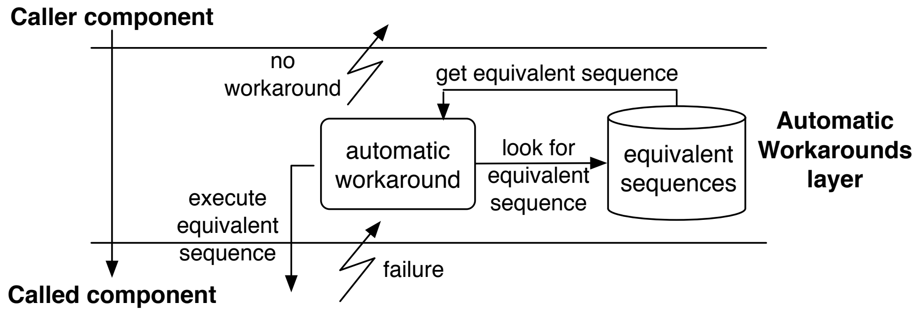


**Fig. 2.** The Automatic Workarounds technique

the failure is reported to the caller component. To address these reported failures, developers have to fix the fault either by fixing the faulty component or by manually identifying and providing valid workarounds to the AW adaptation logic.

In a typical software process, the bug report filled by a user would be the starting point of a long manual effort to deal with run-time failures. Developers would need to identify and analyze the root causes of the problem, identify and implement a patch for the fault, and finally deploy a patch or re-deploy the complete application. The AW technique aims at automating and shifting these activities at run-time to the adaptation logic. This will provide for more timely responses to failures without the need to stop and re-deploy the application.

## 2.2 A Refined Life-cycle Perspective

As mentioned above, a self-adaptive software system performs regular software process activities while the system is running. In Figure 3, we illustrate how a software process and its activities interact with a running self-adaptive software system. The left-hand side of the figure depicts a staged life-cycle model inspired by Rajlich and Bennett [35]. The stages cover the *initial development* of the self-adaptive system and traditional *evolution and adaptation* activities performed *off-line*. Off-line activities work on artifacts, such as design models or source code in a *product repository* and not directly on the running system. The final stage, *Phaseout* covers the shutdown and decommission of the self-adaptive system.

At first sight, and focusing on the left-hand side of the figure, this process looks identical to a traditional software process. However, note it interacts with the *running self-adaptive system*. This interaction takes place through *on-line* activities associated with *evolution and adaptation*, which constitute the self-adaptive system's adaptation logic. Using run-time representations of the self-adaptive system, on-line activities evolve and adapt the domain logic or other adaptation logic while the system is operational in providing services (illustrated by the right-hand side of Figure 3). Interactions and dependencies between off-line and on-line activities, depicted by bidirectional arrows, are specific for life-cycle models targeting self-adaptive systems.

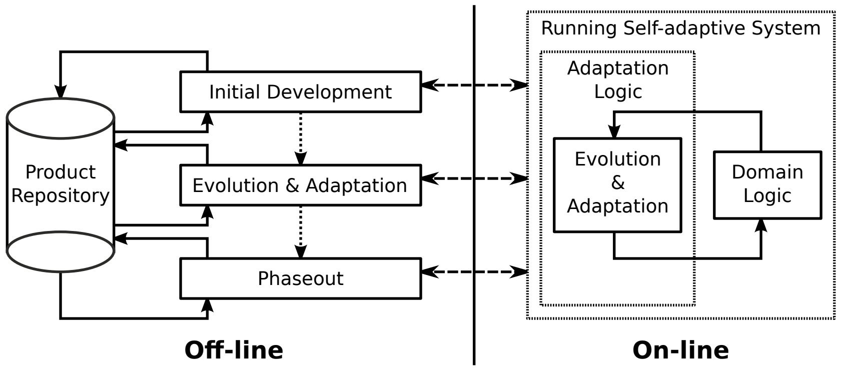


**Fig. 3.** A Life-cycle Model for Self-Adaptive Software System

In order to provide a more in-depth analysis of the interactions between a software process and a running self-adaptive system, more detailed descriptions of activities and their interactions are required. We introduce and discuss a timeline, illustrated in Figure 4, which represents a life-cycle instance for a self-adaptive system that uses an automatic workaround (AW) approach for the adaptation logic and its development process. The timeline view contains two graphs and *interaction points*. The top-most graph depicts the *activity level* (y-axis) in the development process and a number of specific *off-line activities* over *time* (x-axis). We see how the activity level varies over the life-cycle. The life-cycle is divided into the three distinct stages: initial development, evolution and adaptation, and phaseout. The bottom-most graph depicts the *service level* for the self-adaptive AW system over *time* (x-axis). The variations in the graph are due to events, external or internal to the system. For example, on-line activities initiated by the adaptation logic or the development processes (maintenance & evolution activities). The timeline in Figure 4 suggests that the running system acts as a stakeholder with a specific role in the development process, as it actively affects software development and maintenance [2]. However, for the case of self-adaptive systems, this is also true for on-line activities. We may now identify and characterize a number of scenarios where off-line and on-line activities interact, as depicted by the labeled *situations* in Figure 4.

The first stage, initial development, develops a first version of the self-adaptive software system by a number of off-line activities. In the context of the AW approach, software engineers develop the application's domain logic. To enhance the application with the AW technique, they have to provide an initial list of equivalent sequences for application operations with known workarounds, which is an off-line development activity. This also exemplifies how self-adaptation capabilities influence the initial development. Having completed the initial development, the system is ready for deployment. An initial deployment (cf. situation ① in Figure 4) puts the system into operation, which is illustrated by the step in the system's *service level*. The initial deployment activity is captured by the

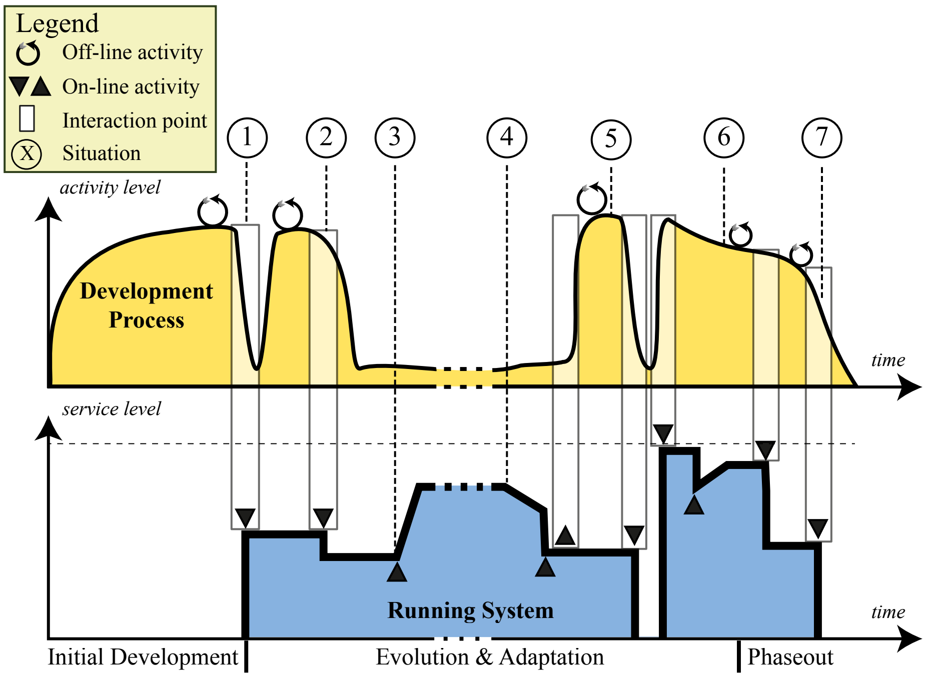


**Fig. 4.** Timeline View on a Process and a Running Self-Adaptive Software System

first *interaction point*. Interaction points indicate that off-line activities impact on-line activities or vice versa.

When the system instance is running, the evolution and adaptation stage starts. As we consider self-adaptive software systems, adaptations and evolution may be initiated and controlled by off-line (process) activities as well as on-line (adaptation logic) activities. This is illustrated by six additional situations following initial deployment (situation ①) in Figure 4.

Situation ② illustrates how changes by off-line evolution or adaptation activities are subsequently enacted to the running system by on-line activities. An on-line update deploys new or updated domain logic components. For these components, the lists of equivalent sequences and thus, the adaptation logic, have to be updated by the developer to preserve AW behaviors. In the scenario we describe, the system's service level is affected negatively after the new deployment due to probable faults in the new components.

However, if corresponding failures occur, they are handled by the AW technique as sketched by situation ③, thanks to previous off-line efforts to maintain the list of equivalent sequences. The AW technique monitors failures in the application, and it is able to deal with them by successfully applying workarounds. This brings the system into a state with improved service level. This situation exemplifies on-line adaptations, which are often enabled by preceding off-line activities.

Nevertheless, the AW technique might not be able to cope with arbitrary failures that continuously affect the system's service level negatively (cf. situation ④). This is the case when the AW technique does not find a valid workaround (first on-line activity in situation ④), i.e., all available equivalent sequences have been tested without success, and as a consequence the failure recurs. In this case, the second on-line AW activity in this situation notifies developers who enact an off-line process to deal with the failures, e.g., by manually correcting the fault and maintaining the list of equivalent sequences. This situation shows how on-line activities interact with and trigger off-line activities.

If the application is re-engineered off-line, an on-line update may be too complex, hence not feasible. Such radical changes to a system are captured in situation ⑤ by off-line evolution activities followed by a deployment. In this situation the running system is shutdown and the new release deployed (similar to situation ①), which affects the system's availability and thus, its service level. Situation ⑥ highlights the case when off-line activities evolve or adapt on-line (adaptation logic) activities followed by enacting these changes to the running system. In context of the AW adaptation logic, at any point in time developers may identify and specify new equivalent sequences, which is an off-line activity that tunes the AW mechanism. Through on-line activities, these new sequences are injected into the AW knowledge and these sequences may be used in subsequent adaptations of the domain logic. Finally, situation ⑦ illustrates the complete shutdown and decommission as part of the phaseout stage, since a decision has been made to discontinue the system. The shutdown and decommission activities, which are planned and initiated off-line, terminate the life-cycle of the system and with some delay the process life-cycle.

Evolution and adaptation activities performed in-between interaction points are in general carried out on-line if they are controlled by the adaptation logic. In contrast, they are carried out off-line if they are controlled by human-driven process activities on product repository artifacts from the initial development stage (cf. Figure 3). In this context, interaction points synchronize off-line and on-line activities or artifacts. As an example, situation ⑥ illustrates that on-line activities can be evolved and adapted off-line, and a subsequent dynamic update synchronizes these off-line changes to the corresponding on-line activities in the adaptation logic.

## 3 Processes for Self-adaptive Software Systems

As we discussed previously, software processes for self-adaptive systems have special characteristics due to their integration with the running system by automating a set of process activities in the system's adaptation logic. This automation and integration define the self-adaptation scope, i.e., the self-adaptation capabilities of the system. In the case of the automatic workarounds approach example, this set covers activities that handle functional run-time failures, but not activities that go beyond the specific idea of workarounds, such as repairing the faults causing the failures (cf. Section 2.1). Therefore, we distinguish between *on-line* activities, which are change activities realized and performed by the system's adaptation logic, and *off-line* activities, which are realized and performed externally to the self-adaptive system (cf. Figure 3). Self-adaptation does not make typical (off-line) activities redundant. In fact, a process has to support both off-line and on-line activities. Furthermore, it has to consider dependencies in between both kinds of activities.

Currently, software processes merely focus on the initial development and off-line evolution and adaptation of the system. The automation and integration require a dramatic change of the concepts associated with traditional software processes. The software process reconceptualization is, therefore, the first of the major challenges we have identified. We believe that a process reconceptualization is essential to effectively and efficiently engineer self-adaptive software systems. We provide a more detailed account for this challenge and its sub-challenges below in Section 3.1.

The outcome of the reconceptualization of software processes should lead to a life-cycle model that conveys a strong intertwining between the on-line and off-line activities and thus, between the self-adaptive system and its engineering process. We identify this as our second major challenge. Achieving effectiveness and efficiency in both is indeed a challenge that for instance includes deciding which activities should be performed on-line and which not. Design decisions of having activities either off-line or on-line should be motivated by the costs and the relative contribution to a product's *value*. In Section 3.2, we discuss related ideas and research challenges.

### 3.1 The Need of Reconceptualizing Software Processes

As mentioned above, traditionally software processes take the perspective that process activities can be either performed at development-time or at run-time. However, recently it has been advocated that due to the distinct features of self-adaptive systems, these type of systems require a reconceptualization of their software processes [3, 7, 21, 22]. We already took the initial steps to modify the perspective on process activities in Section 2 by distinguishing between on-line and off-line activities rather than development-time and run-time activities.

Although on-line and off-line activities perform in different contexts, they are not independent from each other, as it is illustrated by the interactions in Figure 4. Therefore, an effective process for self-adaptive software systems should support both kinds of activities together with the interactions and dependencies.

In order to consider on-line activities, they must be lifted to the abstraction level of software processes. Up to now, on-line activities are only explicitly addressed by self-adaptive systems' designs that describe the adaptation logic. Thus, they are represented in software design models, but not in process models. Hence, to fully capture a software process for self-adaptive systems, on-line activities must be first class entities of processes, and they must be explicitly reflected in process models. It is therefore important to integrate the process models with the self-adaptive system design models in order to seamlessly capture off-line and on-line activities in a process.

Alongside on-line activities, on-line roles and work products need to be addressed by processes too. The adaptation logic of the self-adaptive system is responsible for performing on-line activities, since it assigns on-line process roles to the system. In a self-adaptive system, on-line activities will manipulate work products that are representations of the executing system. Thus, the process has to consider the running system, and more specifically the abstractions that reify adaptation and domain logic as work products of a process.

Having a side by side process support for on-line and off-line activities requires that work products and roles as well as dependencies between them are explicit. For instance, this is required to use dependencies to enable interactions between on-line and off-line activities or to synchronize on-line and off-line work products. Neglecting dependencies would split the process into two subprocesses that may drift apart, what could prevent controlled coevolution of the system and process.

### 3.2 Engineering Self-adaptive Software System with Effective Process Support

The application of a reconceptualized process, in the way we have proposed above, requires support at the process level to effectively and efficiently engineer self-adaptive software systems. In fact, one of the consequences is that software engineers will have to face even more choices when engineering a self-adaptive system, as they have to decide on how to assign activities to off-line or on-line. Such engineering decisions have to be predictable and based on thorough value-based analysis of decision alternatives with well-understood consequences. Thus, we argue that decisions should be guided by models that represent both the costs and the benefits of having an activity performed either on-line or off-line together with the resulting dependencies. Such models should support engineers in making optimal decisions. In self-adaptive systems, some design decisions regarding the system's domain logic are delayed or revisited at run-time. The rationale for this delay is uncertainty, which in turn is a consequence of an information deficit [42]. For example, in the automatic workarounds approach, faults in the domain logic are not known at development-time, and thus, adaptation logic is integrated to cope with failures at run-time that are caused by such unknown faults. Thus, self-adaptation promotes shifting the tasks of traditional off-line activities, such as dealing with run-time failures, to on-line activities.

However, at the same time self-adaptation capabilities introduce additional uncertainty. When engineers leave design decision open to be resolved or revisited at run-time, it will be difficult to have all required information at hand that is required to make decisions with predictable consequences when designing a process and system. Thus, supporting the decision process and to deal with this type of uncertainty requires means to understand, specify, and reason about process activities, process roles, process work products, and dependencies. Such means must be provided at the process level, since any design decision concerning the self-adaptation scope affects the process, and any design decisions concerning the process influences the self-adaptation scope. Concretely, such decisions determine whether activities are performed on-line or off-line, and hence, whether they will become part of the adaptation logic or be handled off-line.

Moreover, such decisions must be based on the contribution to the product's value as perceived by its stakeholder using well-defined criteria that refer to activities, roles, work products, and dependencies. Such criteria specify costs and benefits for different design alternatives and support software engineers in making design decisions. For example in the automatic workarounds (AW) approach, using the AW technique to automatically deal with run-time failures induces additional costs for development (the equivalent sequences of operations must be specified by software developers), but it leads to the benefit of a more robust system, because the system is able to recover from run-time failures instead of crashing. The technique also introduces a performance penalty at run-time. Thus, a solid understanding of value, and useful ways to trade-off competing costs or benefits for the process and system together are required.

Besides issues concerning the design of self-adaptive systems and their processes, the supporting methods, techniques, and tools utilized by the activities are an additional parameter in the equation. Shifting activities from off-line to on-line requires that the underlying methods, techniques, and tools are adapted, optimized, or newly developed in order to be applicable on-line. This will result in a plethora of support alternatives for activities, and each alternative will have an associated value, possibly unique, for every given project and stakeholder. For example, an incremental solution to validation and verification can be efficient enough to be used on-line, but it might not provide the same degree of accuracy as a solution designed for off-line usage. Effective and efficient engineering requires that such value parameters are seamlessly integrated in the engineering process that codesigns on-line and off-line activities to define a process and the complementary adaptation logic in the managing system.

## 4 Process Modeling for Self-adaptive Software Systems

In this section, we discuss a framework, based on software process modeling, that may help in engineering self-adaptive software systems. This requires that process modeling languages support the new concepts, like on-line and off-line activities, that originate from the reconceptualization discussed above. Thereby, process modeling also helps in grasping and understanding the reconceptualization of processes because process models make these concepts explicit. In addition, we discuss a number of remaining research challenges, primarily concerned with the development and application of a process modeling language for engineering self-adaptive software systems' processes.

In general, a *process* is a mechanism to achieve a goal in a systematic way, like following instructions to assemble a product, or following office procedures. Likewise, performing engineering activities to develop and evolve a software product constitutes a process [33]. How a process is carried out is specified by a *process model* that defines a partially ordered set of *what* is done *when* and *where* and by

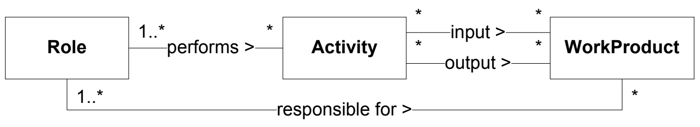


**Fig. 5.** A Conceptual View on the SPEM Meta-Model

*whom* [16, 36]. Thus, assembly instructions, office procedures, or descriptions of software engineering activities are process models, and such models materialize the corresponding processes. This materialization enables human understanding, coordination, and communication, and it supports the analysis, improvement, reuse, execution, or in general the management of processes [16, 33, 36].

To leverage such benefits of process modeling in the software engineering field, several modeling languages have been proposed to describe software processes [16]. One example of such modeling languages is the *Software & Systems Process Engineering Meta-Model Specification* (SPEM) [32]. In this paper, we have adopted SPEM for illustrating the concepts being discussed due to its flexibility, extensibility, and suitability for model-based engineering. The SPEM specification explicitly defines a modeling language by means of a meta-model for describing software development processes. Having an explicit definition of a modeling language, it is possible to discuss and extend the language. As discussed and demonstrated below, extending the language is required for addressing the concepts that originate from the reconceptualization of processes for the case of self-adaptive systems. We are not aware of any process modeling language that supports these concepts as first class elements and provides a rigorous underpinning for model-based engineering approaches.

### 4.1 SPEM-based Process Modeling

By defining a meta-model, SPEM provides a modeling language to specify software development methods and processes, and offers an initial support for configuring and enacting processes in concrete projects. However, the language is generic, since it is meant to support the modeling of processes that span from the waterfall model to agile approaches. Thus, the language supports only the basic and abstract concepts that are present in any development approach. These abstract concepts are depicted in Figure 5, which shows a conceptual and partial view on the SPEM meta-model.

Using this meta-model, we are able to describe a workflow of *Activities* that are performed by *Roles* and that have *WorkProducts* as input or output. Activities can be related to other activities by passing work products along activities, i.e., an output of one activity is the input for another activity. Finally, responsibilities for work products can be assigned to roles. Moreover, the SPEM language provides several elements to represent phases, iterations, and milestones. Finally, SPEM allows to specify different forms of dependencies, e.g., between activities or between work products, primarily to cover relations between activities or

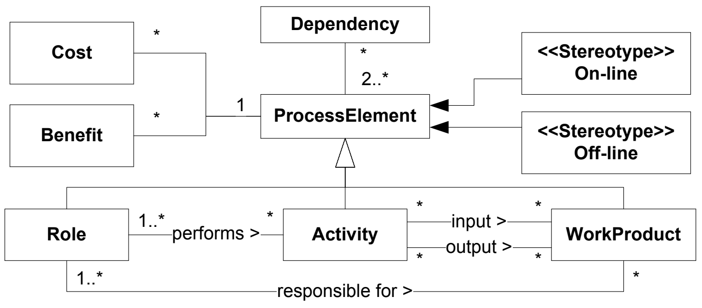


**Fig. 6.** Extended SPEM Meta-Model for Processes of Self-adaptive Systems

composition and impact relations between work products. However, we do not further describe such advanced elements, as they are not critical for the specific challenges that this paper is addressing.

Below, we discuss how we have extended the basic concepts defined by the SPEM language with additional concepts that originate from the reconceptualization of software processes for the specific case of self-adaptive systems. This makes the additional concepts explicit in process models, and it helps to tackle challenges in engineering self-adaptive software systems (cf. Section 5). In the following section, we use our extended version of the SPEM language to model the process of the automatic workarounds approach, and we show how these models can support and manifest the reconceptualization of a software process for self-adaptive systems.

## 4.2 Reconceptualization of SPEM-based Process Modeling

As discussed above, the reconceptualization of software processes to address self-adaptive systems requires a new dimension for classifying activities. This dimension allows for a distinction between on-line and off-line situations, which, however, requires to make dependencies between on-line and off-line situations explicit and manageable. Moreover, costs and benefits of alternative on-line and off-line activities must be considered, as they guide the engineering and influence the designs of processes and systems. Therefore, as depicted in Figure 6, we extended the basic meta-model defined by SPEM with additional concepts. In the scope of this paper, these extensions, as well as the meta-models should be considered at the conceptual level, and not at the technical level as a definitive and fully specified modeling language. Thus, we do not discuss how the extensions could be best implemented or realized within the complete meta-model defined in the SPEM specification [32].

To keep the extensions to the original meta-model (cf. Figure 5) simple and generic, we have added the *ProcessElement* as a common super meta-class for the role, activity, and work product meta-classes. All further extensions refer to this *ProcessElement* and thus, they refer to all three concepts of role, activity, and work product. First of all, the *On-line* and *Off-line* stereotypes have been defined for process elements to clearly define whether any process element occurs on-line or off-line. More precisely, if the stereotype is associated with a role, it indicates whether the role is part of the self-adaptive system (on-line) or not (off-line). For a work product, it indicates whether such a work product is produced or generally used on-line or off-line. Likewise, applying the stereotypes to activities, process models may clearly distinguish whether any activity is performed on-line or off-line.

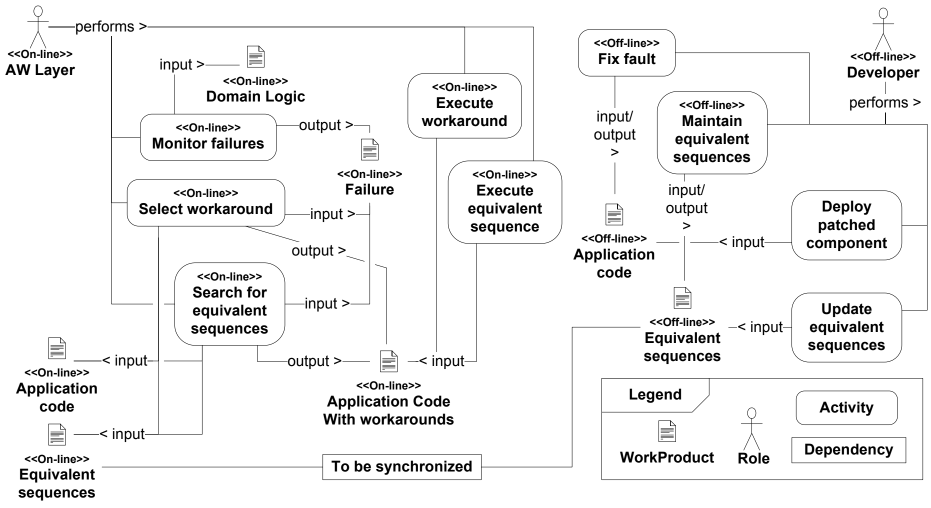
Moreover, we extended the SPEM modeling language with the concept of *Dependency* that relates two or more arbitrary process elements. This notion of dependency is more amenable and flexible for conceptual discussions than the specific possibilities provided by SPEM to cover, e.g., dependencies between activities or between work products. However, to implement these extensions within the SPEM meta-model, the already existing means to specify dependencies should be considered. Providing a generic notion of such concepts makes arbitrary dependencies, such as the different forms of interactions between on-line and off-line activities shown in Section 2, explicit in process models. As an example, being able to perform an activity on-line might require that another activity is in place off-line, which constitutes a dependency.

Finally, *Costs* and *Benefits* can be associated with any process element. This supports design decisions concerning the scoping of on-line and off-line activities, and should give answers to questions such as: “What are the costs and benefits of performing this activity on-line, in contrast to performing it off-line, and what other activities are affected or even required for the on-line and off-line variants?”

We now provide an example of how the extended SPEM language can be used to model the process of an application that relies on the AW approach for achieving self-adaptation, as described in Section 2.1. Figure 7 provides a high-level, structural view of the approach. This view includes all the core concepts of the SPEM language, which are the *Roles*, the *Activities*, and the *WorkProducts*.

Roles are depicted by actor icons, activities by rounded rectangles, and work products by document artifacts. As discussed above, these process elements can be stereotyped with *Off-line* or *On-line* to mark whether they belong to the off-line or on-line part of the process, respectively. Finally, dependencies are represented by rectangles connected to the interdependent process elements.

As shown in Figure 7, there are two roles in the AW approach: the *AW Layer* and the *Developer*. Both roles perform activities that have work products as input or output. The AW layer as the adaptation logic is part of the running self-adaptive system, and thus it is an on-line role. It monitors the execution of the application, i.e., the *Domain Logic*, on-line. If a *Failure* occurs, and it has been detected, the AW layer is in charge of either selecting a workaround, if any is known from previous executions, or searching for equivalent sequences, if no workaround is known. Thereby, the *Application Code with workarounds* is either selected or created by integrating the promising *Equivalent sequences* into the domain logic's *Application code*. Finally, the AW layer enacts the adjusted application code and thus the adaptation by executing a known workaround or the equivalent sequences in an attempt to find a valid workaround.



<<On-line>> AW Layer
performs >
input >
<<On-line>> Domain Logic
<<On-line>> Monitor failures
output >
<<On-line>> Failure
<<On-line>> Execute workaround
<<On-line>> Execute equivalent sequence
<<On-line>> Select workaround
input >
output >
<<On-line>> Search for equivalent sequences
input >
output >
< input
<<On-line>> Application code
< input
<<On-line>> Equivalent sequences
< input
< input
<<On-line>> Application Code With workarounds
To be synchronized
<<Off-line>> Fix fault
input/ output >
<<Off-line>> Application code
<<Off-line>> Maintain equivalent sequences
input/ output >
<<Off-line>> Developer
performs >
Deploy patched component
< input
<<Off-line>> Equivalent sequences
< input
Update equivalent sequences
Legend
WorkProduct
Role
Activity
Dependency

**Fig. 7.** *Roles*, *Activities*, and *WorkProducts* in the Automatic Workarounds approach

The developer is an off-line role that maintains the list of equivalent sequences for the components of the domain logic. If the AW layer does not manage to find a valid workaround automatically, the developer fixes the related faults in the failing components and deploys the patched component to the running system. This requires that the maintained list of equivalent sequences for this component is updated in the AW layer for future on-line use. This update synchronizes the list of *Equivalent sequences* maintained off-line by the developer with the list of *Equivalent sequences* used on-line by the AW layer. This tackles the *To be synchronized* dependency between these two work products. Instead of hiding such dependencies in activities, they should be made explicit in the process models. Otherwise, they might get lost when the process and its activities change or evolve.

In addition to structural aspects, as depicted in Figure 7, the process behavior also needs to be specified. The original SPEM language allows to integrate external languages for behavior modeling, like *UML Activity* diagrams (cf. [32]). Likewise, the extended SPEM language we are proposing does not define its own behavior modeling formalism but uses UML activity diagrams. For our AW approach, Figure 8 depicts a UML activity diagram representing the workflow of activities for the case when a failure occurs and needs to be resolved.

The activity diagram represents the evolution and adaptation stage in the process timeline (cf. Figure 4) for a system that relies on the AW approach. The model consists of two partitions, one for each role, namely the AW layer and the developer. Each partition contains the activities performed by the corresponding role, and the model defines the workflow of activities within and across partitions respectively roles. The roles and activities used in the activity diagram are the same as in the structural process view depicted in Figure 7.

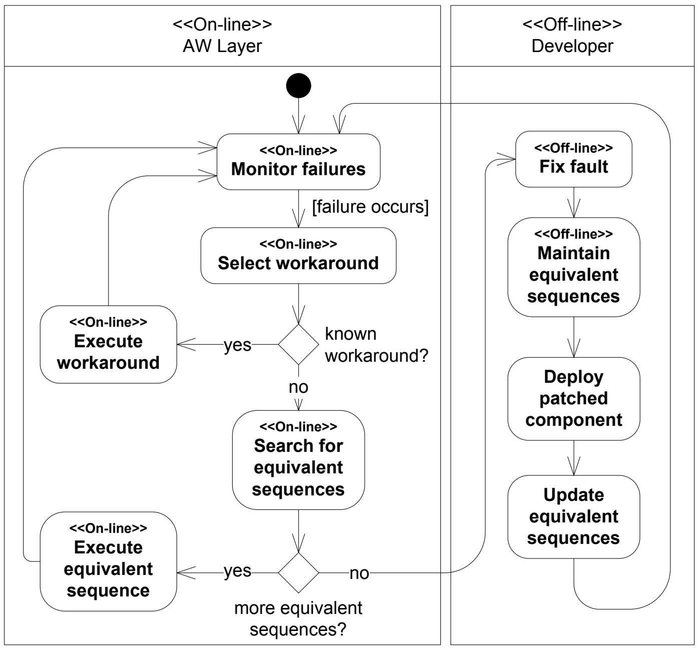


**Fig. 8.** Workflow of Activities in the Automatic Workarounds approach

The AW layer monitors the status of the application to detect failures. When a failure occurs, it selects a workaround if any is already available from previous executions. If a workaround is available, the AW layer executes it immediately on-line. If no workaround is known, then the AW layer looks for equivalent sequences, and once it selects one, the selected sequence is executed. If the execution of the workaround or the equivalent sequence causes another failure, the loop continues until either one equivalent sequence does not cause any failure, or until there are no more equivalent sequences to try. In the last case, the developer has to fix the fault and maintain the list of equivalent sequences, which is followed by deploying the patched component and updating the list of equivalent sequences to make the off-line changes available to the AW layer in the running system.

In this section, we have shown that modeling processes for self-adaptive systems using an extended SPEM language lifts the concepts that originate from the reconceptualized life-cycle to the abstraction level of software processes. This makes it possible to take a software process perspective on engineering self-adaptive systems, which is helpful in tackling the challenges related to the engineering of self-adaptive software systems.

## 5 Engineering Challenges

With the modeling aspect of the framework in place, we may shift focus to *design*, *decision making*, and *reasoning*. One of the key challenges we have identified in the engineering of self-adaptive systems is to partition the process activities effectively between off-line and on-line activities. As mentioned in the previous section, these activities may have costs and benefits associated to help in defining a process for each specific self-adaptive system that brings *value* to its stakeholders. *Value* here has a broad meaning that should encompass a number of aims: the system goals (quantitative and qualitative), the uncertainty that characterizes the execution environment (that defines the scope of adaptation) [42], the resource constraints of the execution environment (to support on-line activities), and the availability of accessing remote resources (to support off-line activities). This requires quantitative reasoning capabilities at the process definition level that shall suitably be complemented with stochastic reasoning to properly take into account the uncertainty dimension of the problem. As an illustration of the principles and practices in such reasoning support, we use *Value-Based Software Engineering* (VBSE) [6]. Biffl et al. argue that VBSE supports better software engineering decisions, providing an economic perspective where value bridges separation of concerns employed to manage complexity, thus allowing for achieving global optimums. The output of a software engineering process, the software system, has a number of goals associated. The purpose of an engineering process is to derive a solution, which optimizes the value (related to goals) of the product under current conditions. Engineering processes are characterized by their predictable outcomes, i.e., decisions made in a process have well-known consequences. Another characteristic is the continuous search for alternative solutions and an exhaustive evaluation of alternative solutions to provide sufficient knowledge on which decisions will be based.

The reconceptualization of software process activities, which allows some activities to migrate from off-line to on-line, dissolves a previously crisp boundary that separated software processes and the running system. This implies that the two may not be treated as separate concerns by an engineering process. Applying a value-based perspective on engineering a self-adaptive software system requires an understanding of value and that value is a main driver for the design and implementation of software processes for self-adaptive software systems and the systems themselves. Differently from how commonly intended in VBSE, cost in our context mainly concerns the impact of the activity in terms of resource consumption of the system's execution environment, e.g., computational time, memory use, etc. Benefit is the measure of the impact of the activity in terms of the system's goal, e.g., verification, graceful adaptation in presence of faults, etc. Value is the measure of the degree of satisfaction of the system's goals that can be achieved with the defined costs/benefits process tradeoffs.

VBSE is centered around Theory W [8] that aims at making all stakeholders in a project winners. VBSE suggests four supporting theories to "achieve and maintain a win-win state" [6, p. 19]; *dependency theory*, *utility theory*, *decision theory*, and *control theory*. Theory W and VBSE are developed with a process model which maintains watertight partitions between software processes and the running system. Our hypothesis is that the reconceptualization discussed above will impact VBSE in a fundamental way. However, the proposed approach where activities are modeled in a uniform way paves the way for customizing VBSE for developing self-adaptive software systems.

The first step in VBSE and Theory W is the identification of *success-critical stakeholders* (SCS). SCS are highly-important stakeholders and a project "will succeed if and only if it makes winners of *[the project's]* success-critical stakeholders" [6, p. 18]. It should be clear from the above discussion that in engineering a self-adaptive software system, the system itself is a SCS. Indeed, the extension proposed to the process modeling framework supports that the system is an SCS by making run-time roles, activities, and work-products explicit. At the same time, the execution environment characteristics, its resources and operational constraints, including its potential uncertain variability, represent another SCS. Depending on the self-adaptive system, the users of the system may represent another crucial SCS, they may, for example, define the acceptable behavioral variability of the system. The next step in VBSE identifies what is required to make a SCS a winner. Utility theory will play an important role here and may be used to define *value* on a level of detail where individual activities may define their win conditions. However, roles, activities, and work-products are not for free. We discussed the issue of relative cost above, and annotating process entities with costs will be essential in the next step where SCS should agree upon realization plans that will make all SCS winners. Eventually these plans will be implemented, a procedure which should be controlled to guarantee that the final products make all SCS winners.

The research challenge in this area is to formulate a VBSE theory and process for engineering self adaptive systems. We have described above some specific extensions to the SPEM meta-model that support value-based engineering. The *ProcessElement* in Figure 6 are annotated with collections of *Cost* and *Benefit* attributes. These concepts will represent the relative contribution of a specific *Role*, *Activity*, or *WorkProduct* to a *value*. The underlying idea with assigning costs and benefits to process elements is of course to use this knowledge in engineering activities. With the extended SPEM language, engineers are provided with the means to model and reason about how to design evolution and adaptation activities in a system. Distinguishing off-line activities and on-line activities opens up a design space where engineers may have several alternatives and eventually select the alternative that contributes relatively the most to the stakeholders' values. We illustrate some specific research challenges below using references from the automatic workaround example.

The first research challenge is to *provide the means for expressing value, that is the costs and benefits*. In the model we proposed, we describe off-line and on-line activities in an analogous way including cost and benefit attributes. This is an extremely simple approach to model *value*. We must develop ways for expressing *stakeholder specific values* in a way that they are useful for reasoning, evaluation, and eventually decision making. For example, the automatic workaround mechanism replaces a number of roles, activities, and work products. However it is not clear how to annotate these with costs, benefits, or any other type of value, neither for the process elements in the automatic workaround nor for its traditional, equivalent, off-line realization.

If we succeed in defining *a value-framework* for engineering self-adaptive software systems, engineers can reason about design alternatives, evaluate, and make predictable decisions about the relative contribution to the overall value. However, it is not clear how to reason about and evaluate alternatives in a structured manner. This takes us to the second challenge, *we need to design new reasoning and evaluation techniques*, potentially based on the large body of existing value-based design methods, for instance [15]. The design of a system's adaptation subsystem will require that engineers decide if and which activity should be performed off-line or on-line. Consider for example the *Maintain equivalent sequences* activity in Figure 7. It is performed by a *Developer*. However, it is not unlikely that future evolution of the AW mechanism provides for additional alternative realizations of this activity, for instance, by means of other automatic on-line activities. In that situation, engineers have a selection of alternatives to choose from in order to select the combination with the greatest relative contribution to the stakeholders' values.

Another challenge associated with processes for self-adaptive software systems is the fact that processes need to be generated dynamically at run-time since changes affecting the system, its context and goals may require their adaptation. This may imply that depending on the system's operational conditions, different processes can be generated by changing their activities or workflows. Moreover, since off-line and on-line activities might influence each other, it is important to consider how the initial development-time design rationale can affect the processes being generated at run-time, and vice versa. Finally, it is also crucial to incorporate into off-line activities the decisions being made during run-time since they would provide insightful knowledge about the operational profile of the system.

## 6 Related Work

Different researchers like Finkelstein and Blair et al. [7] or Inverardi and Tivoli [21, 22] have also identified the need for new software engineering paradigms suggesting a reconceptualization of software processes. Among others, this is motivated by the blurring boundary between development-time and run-time as discussed in [3, 7]. This is inline with the motivation for our work in this paper on revising the life-cycle and processes for self-adaptive software systems.

Challenges for software evolution are also discussed by Mens et al. who specifically state that "*[I]*t is important to investigate how the notion of software change can be integrated into the conventional software development process models" [30, p. 17]. They consider agile or in general iterative and incremental development processes as promising approaches to integrate support for change in the life-cycle. In contrast to this paper, they do not focus on life-cycle or process issues related to changes by means of self-adaptation or related to the blurring boundary between development-time and run-time. Likewise, Buckley et al. [11] or McKinley et al. [28] clearly distinguish between changes performed statically at development-time or dynamically at run-time. This is based on a traditional view on a system life-cycle, while we promote a refined view that primarily considers on-line and off-line changes or in general on-line and off-line process activities. In this context, by on-line and off-line we refer to different ways changes are carried out, but not whether the running system's domain logic provides service or not while being changed (cf. *availability* dimension in [11]).

Salehie and Tahvildari [39] discuss research challenges for the specific case of self-adaptive software, but not from a process view. They briefly consider a developing phase and an operating phase for self-adaptive systems, while the developing phase determines the adaptation capabilities in the operational phase. However, these phases are not used for discussing the challenges and in particular, this distinction into these two phases is similar to the traditional view on life-cycles exclusively separating development-time and run-time.

Gacek et al. [18] view evolution and adaptation as processes that include roles, artifacts etc., which is similar to our work. They propose a self-adaptation reference process that consists of two iteratively interacting processes. The inner adaptation process addresses the component control and change management layers of the reference architecture by Kramer and Magee [25], while the outer evolution process relates to the goal management layer. The authors conceptually discuss how a manual or partially automated evolution process guides an automated adaptation process by interactions between these two processes. These interactions seem to be similar to the interaction points between on-line and off-line process activities that we discussed in the context of Figure 4. However, Gacek et al. focus on discussing the co-existence of self-adaptation and traditional change management or evolution, but they do not discuss implications on software engineering processes or system life-cycles as this paper does.

Other approaches investigating software processes for self-* systems and especially for adaptive multi-agent systems cover only the development of such systems and not the whole life-cycle [34, 37]. This means that the adaptation mechanisms are exclusively considered as part of the system to be developed, but not as a part of the life-cycle process itself. Thus, both approaches [34, 37] describe processes or methodologies that specify the development of the systems including the development of the adaptation mechanisms. In contrast, besides considering adaptation mechanisms as part of a self-adaptive system, we also lift the adaptation mechanisms to the process level by treating the adaptation logic as a process role and the tasks performed by the adaptation logic as process activities. Consequently, we address processes that describe the whole system life-cycle comprising the development as well as the adaptation and evolution of the system. Nevertheless, one commonality between our work and [34, 37] is the usage of the same process modeling language, namely SPEM, though we conceptually extended SPEM due to the required reconceptualization of software processes.

Our work is also motivated by the fact that approaches to modern or self-adaptive software systems do not design comprehensive software processes spanning on-line and off-line activities for their approaches, and they just shift specific typical process activities to the system in order to be performed on-line. For example, Brenner et al. [10] equips components with mechanisms to test them on-line, which shifts typical validation and verification activities and efforts to the run-time. Another example is the work of Bencomo et al. [5] who consider a self-adaptive system as a dynamic software product line that determines product configurations on-line and at run-time, while for traditional product lines the configurations are determined off-line and usually before deployment. Such approaches can benefit from our work since we provide initial means to model and analyze processes that cover both on-line and off-line activities. This might help other approaches to engineer their systems with effective process support.

Another initiative associated with processes for self-adaptive software systems is the dynamic generation of plans at run-time [40, 41]. A key factor motivating this work was how to deal with the uncertainty related to changing goals, unexpected resource conditions, and unpredictable environments when managing the adaptation of software systems. This has shown particularly relevance when applied to the generation of plans for managing the integration testing of self-adaptive systems [41], which is a process that involves to calculate integration order, generate stubs and test cases, and perform the actual tests. Although this work is restricted to on-line activities, it would be interesting to consider how off-line activities could affect the automatic generation of plans, and how cost and benefit could be integrated with the decision making of selecting the most appropriate plan.

## 7 Conclusion and Future Work

The actual support for self-adaptation throughout the entire life-cycle of self-adaptive software systems requires a reconceptualization of the way they are engineered. Therefore, we presented a first integrated view of the problem, suitable abstractions for off-line and on-line process activities, and details of major challenges concerning the reconceptualization of software processes for self-adaptive software systems. Moreover, for tackling the challenges related to the engineering of self-adaptive software systems, we proposed an approach based on process modeling and value-based software engineering. An essential part of this approach is the intertwining of a self-adaptive software system and its software process.

As future work, we plan to elaborate the reconceptualization of software processes for self-adaptive systems, e.g., by investigating the impact of the on-line/off-line perspective on state-of-the-art approaches, methods, and techniques to design software processes and to engineer self-adaptive systems. Having more profound knowledge about reconceptualized software processes, we can work on formalizing the modeling language to fully capture a process and its system. A formal language is the prerequisite for automated analysis that follows the theory of value-based software engineering. Therefore, we have to adapt this theory to address specifics of self-adaptive systems and processes for such systems. For instance, we need to think about benchmarks and special-purpose metrics to assess processes and the corresponding self-adaptive systems as well as their values based on costs and benefits. How can we say that a given process is better than another one at identifying, designing, implementing, and running the on-line and off-line process activities for a system?

Besides these initial directions, a possible research agenda should in particular comprise the following elements. We need to better understand how to elicit functional and non-functional requirements of self-adaptive systems, especially, on how these should be associated with on-line and off-line activities and their dependencies. This could lead to a model-driven solution for the development, deployment, adaptation, and evolution of these system. We also need to better understand the dependencies between self-adaptation and software evolution since the former does not imply replacing the latter. This requires clear definitions of (self-)adaptation and evolution, and how both can be seamlessly integrated in a self-adaptive system's process. For example, there might be the need for understanding how to evolve the system based on its run-time adaptations. Experiences from the adaptations performed in the past may offer useful knowledge for the evolution of the system.

All these research directions promote our ultimate goal of effectively and efficiently engineering self-adaptive systems with proper software process support. Thereby, the process perspective should leverage systematic approaches to engineering self-adaptive systems, which have predictable outcomes concerning effectiveness and efficiency.

## Acknowledgment

This paper is the result of stimulating discussions among the authors and other participants, especially Bojan Cukic, Oscar M. Nierstrasz, Sooyong Park, and Dennis B. Smith, during the seminar on *Software Engineering for Self-Adaptive Systems* at Schloss Dagstuhl in October 2010 (`http://www.dagstuhl.de/10431`).

## References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Reflecting on self-adaptive software systems. In: Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'09). pp. 38–47. IEEE Computer Society (2009)
2. Bai, X., Huang, L., Zhang, H.: On scoping stakeholders and artifacts in software process. In: Munch, J., Yang, Y., Schafer, W. (eds.) New Modeling Concepts for Today's Software Processes. LNCS, vol. 6195, pp. 39–51. Springer (2010)
3. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proc. of the FSE/SDP workshop on Future of software engineering research (FoSER'10). pp. 17–22. ACM, New York (2010)

4. Beck, K.: Embracing Change with Extreme Programming. IEEE Computer 32(10), 70–77 (1999)
5. Bencomo, N., Sawyer, P., Blair, G., Grace, P.: Dynamically adaptive systems are product lines too: Using model-driven techniques to capture dynamic variability of adaptive systems. In: Thiel, S., Pohl, K. (eds.) Proc. of the 12th International Software Product Line Conference (SPLC'08), Second Volume (Workshops). pp. 23–32. Lero Int. Science Centre, University of Limerick, Ireland (2008)
6. Biffl, S., Aurum, A., Boehm, B., Erdogmus, H., Grünbacher, P. (eds.): Value-Based Software Engineering. Springer (2006)
7. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. IEEE Computer 42(10), 22–27 (2009)
8. Boehm, B.W., Ross, R.: Theory-W Software Project Management Principles and Examples. IEEE Trans. Softw. Eng. 15(7), 902–916 (1989)
9. Boehm, B.W.: A Spiral Model of Software Development and Enhancement. IEEE Computer 21(5), 61–72 (1988)
10. Brenner, D., Atkinson, C., Malaka, R., Merdes, M., Paech, B., Suliman, D.: Reducing verification effort in component-based software engineering through built-in testing. Information Systems Frontiers 9(2), 151–162 (2007)
11. Buckley, J., Mens, T., Zenger, M., Rashid, A., Kniesel, G.: Towards a taxonomy of software change. Journal of Software Maintenance and Evolution: Research and Practice 17(5), 309–332 (2005)
12. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds for web applications. In: Proc. of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'10). pp. 237–246. ACM, New York (2010)
13. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08). pp. 17–24. ACM, New York (2008)
14. Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Serugendo, G.D.M., 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: A Research Roadmap. In: Cheng, B.H., Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer (2009)
15. Clements, P., Kazman, R., Klein, M.: Evaluating Software Architectures: Methods and Case Studies. Addison-Wesley, Boston, MA (2001)
16. Curtis, B., Kellner, M.I., Over, J.: Process modeling. Commun. ACM 35(9), 75–90 (1992)
17. Gabriel, R.P., Northrop, L., Schmidt, D.C., Sullivan, K.: Ultra-large-scale systems. In: OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications. pp. 632–634. ACM, New York, NY, USA (2006)
18. Gacek, C., Giese, H., Hadar, E.: Friends or foes?: a conceptual analysis of self-adaptation and it change management. In: Proc. of the ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS'08). pp. 121–128. ACM, New York (2008)
19. Gilb, T.: Evolutionary development. SIGSOFT Softw. Eng. Notes 6(2), 17 (1981)
20. Gilb, T.: Evolutionary Delivery versus the waterfall model. SIGSOFT Softw. Eng. Notes 10(3), 49–61 (1985)

21. Inverardi, P.: Software of the Future Is the Future of Software? In: Montanari, U., Sannella, D., Bruni, R. (eds.) Trustworthy Global Computing. LNCS, vol. 4661, pp. 69–85. Springer (2007)
22. Inverardi, P., Tivoli, M.: The Future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) Software Engineering. LNCS, vol. 5413, pp. 1–31. Springer (2009)
23. Jacobson, I., Booch, G., Rumbaugh, J.: The unified process. IEEE Software 16(3), 96–102 (1999)
24. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. IEEE Computer 36(1), 41–50 (2003)
25. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: Future of Software Engineering (FOSE'07). pp. 259–268. IEEE Computer Society (2007)
26. Lehman, M.M.: Software's Future: Managing Evolution. IEEE Software 15(01), 40–44 (1998)
27. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc., San Diego, CA, USA (1985)
28. McKinley, P., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.C.: Composing Adaptive Software. IEEE Computer 37(7), 56–64 (2004)
29. Mens, T.: Introduction and Roadmap: History and Challenges of Software Evolution, chap. 1. Software Evolution, Springer (2008)
30. Mens, T., Wermelinger, M., Ducasse, S., Demeyer, S., Hirschfeld, R., Jazayeri, M.: Challenges in software evolution. In: Proc. of the 8th International Workshop on Principles of Software Evolution (IWPSE'05). pp. 13–22. IEEE Computer Society (2005)
31. Northrop, L., Feiler, P.H., Gabriel, R.P., Linger, R., Longstaff, T., Kazman, R., Klein, M., Schmidt, D.: Ultra-Large-Scale Systems: The Software Challenge of the Future. Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (2006)
32. Object Management Group (OMG): Software & Systems Process Engineering Meta-Model Specification (SPEM), Version 2.0 (2008)
33. Osterweil, L.J.: Software processes are software too. In: Proc. of the 9th International Conference on Software Engineering (ICSE'87). pp. 2–13. IEEE Computer Society, Los Alamitos (1987)
34. Puviani, M., Serugendo, G.D.M., Frei, R., Cabri, G.: Methodologies for self-organising systems: A spem approach. In: Proc. of the IEEE/WIC/ACM International Joint Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'09) - Volume 02. pp. 66–69. IEEE Computer Society (2009)
35. Rajlich, V.T., Bennett, K.H.: A Staged Model for the Software Life Cycle. IEEE Computer 33(7), 66–71 (2000)
36. Rolland, C.: Modeling the requirements engineering process. In: Markus, A.F., Jaakkola, H., Tadahiro, K., Kangassalo, H. (eds.) Information Modelling and Knowledge Bases V: Principles and Formal Techniques: Results of the 3rd European-Japanese Seminar, Held in Budapest, Hungary, May 31-June 3, 1993. pp. 85–96. IOS Press (1994)
37. Rougemaille, S., Migeon, F., Millan, T., Gleizes, M.P.: Methodology fragments definition in spem for designing adaptive methodology: A first step. In: Luck, M., Gomez-Sanz, J. (eds.) Agent-Oriented Software Engineering IX, LNCS, vol. 5386, pp. 74–85. Springer, Berlin (2009)
38. Royce, W.: Managing the Development of Large Software Systems: Concepts and Techniques. In: Proc. IEEE WESTCON. IEEE Computer Society Press (1970),

reprinted in Proc. of the 9th International Conference on Software Engineering (ICSE'87), pages 328-338, IEEE Computer Society

39. Salehie, M., Tahvildari, L.: Self-adaptive software: Landscape and research challenges. ACM Trans. Auton. Adapt. Syst. 4(2), 1–42 (2009)
40. da Silva, C.E., de Lemos, R.: Using dynamic workflows for coordinating self-adaptation of software systems. In: Proceedings of the 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2009). pp. 86–95. IEEE Computer Society, Washington, DC, USA (2009)
41. da Silva, C.E., de Lemos, R.: Dynamic plans for integration testing of self-adaptive software systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011). pp. 148–157. ACM, New York, NY, USA (2011)
42. Welsh, K., Sawyer, P.: Understanding the scope of uncertainty in dynamically adaptive systems. In: Wieringa, R., Persson, A. (eds.) Requirements Engineering: Foundation for Software Quality. LNCS, vol. 6182, pp. 2–16. Springer (2010)
43. Yau, S.S., Colofello, J.S., MacGregor, T.: Ripple effect analysis of software maintenance. In: Proc. of the 2nd International Conference on Computer Software and Applications (COMPSAC'78). pp. 60–65. IEEE Computer Society (1978)