Software Engineering for Self-Adaptive Systems: A Second Research Roadmap

Chapter · January 2013

DOI: 10.1007/978-3-642-35813-5_1

42 authors, including:

Holger Giese
Hasso Plattner Institute
407 PUBLICATIONS **9,221** CITATIONS

SEE PROFILE

Hausi A. Müller
University of Victoria
229 PUBLICATIONS **8,392** CITATIONS

SEE PROFILE

Jesper Andersson.
Linnaeus University
56 PUBLICATIONS **3,170** CITATIONS

SEE PROFILE

Software Engineering for Self-Adaptive Systems: A Second Research Roadmap

Rogério de Lemos, Holger Giese, Hausi A. Müller, Mary Shaw, Jesper Andersson, Marin Litoiu, Bradley Schmerl, Gabriel Tamura, Norha M. Villegas, Thomas Vogel, Danny Weyns, Luciano Baresi, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Ron Desmarais, Schahram Dustdar, Gregor Engels, Kurt Geihs, Karl M. Göschka, Alessandra Gorla, Vincenzo Grassi, Paola Inverardi, Gabor Karsai, Jeff Kramer, Antónia Lopes, Jeff Magee, Sam Malek, Serge Mankovskii, Raffaela Mirandola, John Mylopoulos, Oscar Nierstrasz, Mauro Pezzè, Christian Prehofer, Wilhelm Schäfer, Rick Schlichting, Dennis B. Smith, João Pedro Sousa, Ladan Tahvildari, Kenny Wong, and Jochen Wuttke

r.delemos@kent.ac.uk, holger.giese@hpi.uni-potsdam.de, hausi@cs.uvic.ca, mary.shaw@cs.cmu.edu

Abstract. The goal of this roadmap paper is to summarize the state-of-the-art and identify research challenges when developing, deploying and managing self-adaptive software systems. Instead of dealing with a wide range of topics associated with the field, we focus on four essential topics of self-adaptation: design space for self-adaptive solutions, software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification & validation for self-adaptive systems. For each topic, we present an overview, suggest future directions, and focus on selected challenges. This paper complements and extends a previous roadmap on software engineering for self-adaptive systems published in 2009 covering a different set of topics, and reflecting in part on the previous paper. This roadmap is one of the many results of the Dagstuhl Seminar 10431 on *Software Engineering for Self-Adaptive Systems,* which took place in October 2010.

1 Introduction

The complexity of current software systems has led the software engineering community to investigate innovative ways of developing, deploying, managing and evolving software-intensive systems and services. In addition to the ever increasing complexity, software systems must become more versatile, flexible, resilient, dependable, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changes that may occur in their operational contexts, environments and system requirements. Therefore, *self-adaptation* — systems that are able to modify their behavior and/or structure in response to their perception of the environment and the system itself, and their goals — has become an important research topic in many diverse application areas.

R. de Lemos et al. (Eds.): Self-Adaptive Systems, LNCS 7475, pp. 1–32, 2013.

It is important to emphasize that in all the many initiatives to explore self-adaption, the common element that enables its provision is usually software. Although control theory provides 60 years of experience and software the necessary flexibility to attain self-adaptability, the proper engineering and realization of self-adaptation in software still remains a formidable intellectual challenge. Moreover, only recently have the first attempts been made to establish suitable software engineering approaches for the provision of self-adaptation. In the long run, we need to establish the foundations that enable the systematic development, deployment, management and evolution of future generations of self-adaptive software systems.

The goal of this roadmap paper is to summarize the state-of-the-art and identify research challenges when developing, deploying, managing and evolving self-adaptive software systems. Specifically, we focus on development methods, techniques, and tools that we believe are required when dealing with software-intensive systems that are self-adaptive in their nature. In contrast to merely speculative and conjectural visions and ad hoc approaches for systems supporting self-adaptability, the objective of this paper is to establish a roadmap for research and identify the key research challenges.

The intent of this new roadmap paper is not to supersede the previous paper on software engineering self-adaptive systems [15], but rather to complement and extend it with additional topics and challenges. The research challenges identified in the previous paper are still valid. Moreover, it is too early to reassess the conjectures made in that paper. In order to provide a context for this roadmap, in the following, we summarize the most important challenges identified in the first roadmap paper [15].

- *Modeling dimensions* — the challenge was to define models that can represent a wide range of system properties. The more precise the models are, the more effective they should be in supporting run-time analyses and decision processes.
- *Requirements* — the challenge was to define a new language capable of capturing uncertainty at an abstract level. Once we consider uncertainty at the requirements stage, we must also find means of managing it. Thus, the need to represent the trade-offs between the flexibility provided by the uncertainty and the assurances required by the application.
- *Engineering* — the challenge was to make the role of feedback control loops more explicit. In other words, feedback control loops must become first-class entities throughout the lifecycle of self-adaptive systems. Explicit modeling of feedback loops will ease reifying system properties to allow their query and modification at run-time.
- *Assurances* — the challenge was how to supplement traditional V&V methods applied at requirements and design stages of development with run-time assurances. Since system context changes dynamically at run-time, systems must manage contexts effectively, and its models must include uncertainty.

Similar to previous research roadmap paper, instead of dealing with a wide range of topics associated with the field, this paper focuses on four essential topics

of self-adaptation: design space of self-adaptive solutions, software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification and validation (V&V) for self-adaptive systems. The presentations of each of the topics do not cover all related aspects, instead focused theses are used as a means to identify challenges associated with each topic. The four identified theses are the following.

- *Design space* – the need to define what is the design space for self-adaptive software systems, including the decisions the developer should address.
- *Processes* – the need to define innovative generic processes for the development, deployment, operation, maintenance, and evolution of self-adaptive software systems.
- *Decentralization of control loops* – the need to define a systematic engineering approach for control loops for software adaptation of varying degree of centralization and decentralization of the loop elements.
- *Practical run-time verification and validation* – the need to investigate V&V methods and techniques for obtaining inferential and incremental assessments for the provision of confidence and certifiable trust in self-adaptation.

Although the topics covered by the two roadmap papers may appear related, the issues covered are quite distinct since the topics covered different theses: design spaces is related to the former modeling dimensions topic but taking a broader more top-down rather than bottom up perspective, processes is a completely new topic, decentralization of control loops looks into the control loop addressed by the former engineering topic with the particular focus on decentralization, and practical run-time V&V refines the related former assurances topic looking in particular into techniques that can be effectively applied at run-time.

In order to motivate and present a new set of research challenges associated with the engineering of self-adaptive software systems, the paper is divided into four parts, each related to one of the new topics identified for this research roadmap. For each topic, we present an overview, suggest future directions, and focus on selected challenges. The four topics are: design space for adaptive solutions (Section 2), towards software engineering processes for self-adaptive systems (Section 3), from centralized to decentralized control (Section 4), and practical run-time verification and validation (Section 5). Finally, Section 6 summarizes our findings.

2 Design Space

Designing self-adaptive software systems involves making design decisions about observing the environment and the system itself, selecting adaptation mechanisms, and enacting those mechanisms. While most research on self-adaptive systems deals with some subset of these decisions, to the best of our knowledge, there has been neither a systematic study of the overall design space for such systems nor an enumeration of the decisions the developer should address.

2.1 Design Space Definitions

The *design space* of a system is the set of decisions, together with the possible choices, the developer must make. A *representation of a design space* is a static textual or graphical form of a design space, or a subset of that space. Intuitively, a design space is a Cartesian space with dimensions representing the design decisions and values along those dimensions representing the possible choices. Points in the space represent concrete designs. In practice, most interesting design spaces are too rich to represent in their entirety, so representations of the design space capture only the principal decisions, the ones that are significant for the task at hand. Typically, the design dimensions are not independent, so making one decision may preclude, restrict, or make irrelevant, other decisions [9,49].

Several partial methodologies for identifying and representing design spaces have emerged. For example, Andersson *et al.* [1] defined a set of modeling dimensions for self-adaptive software systems. The identified dimensions were organized into four categories: the self-adaptive goals of the system, the causes or triggers of self-adaptation, the mechanisms used to adapt, and the effects of those mechanisms on the system. Kramer and Magee [33] outline three tiers of decisions the developer must make — ones that pertain to goal management, change management, and component control. Dobson *et al.* [17] identify four aspects of self-adaptive systems around which decisions can be organized: collect, analyze, decide, act. Finally, Brun *et al.* [10] discuss the importance of making the adaptation control loops explicit during the development process and outline several types of control loops that can lead to adaptation. Specific design spaces have also been proposed in the form of taxonomies. For example, Brake *et al.* [8] introduce (and Ghanbari *et al.* [22] later refine) a taxonomy for performance monitoring of self-adaptive systems together with a method for discovering parameters in source code. Ionescu *et al.* [29] formally define controllability and observability for web services and show that controllability can be preserved in composition.

2.2 Key Design Space Dimensions

In this section, we outline a design space for self-adaptive systems with five principal dimensions — clusters of design decisions pertinent to self-adaptive systems. The clusters are: observation, representation, control, identification, and enacting adaptation. Each cluster provides additional structure in the form of questions a developer should consider when designing such a system. The *Observation* cluster answers questions related to what is monitored by the self-adaptive system, when and how often monitoring occurs, and how states are determined based on the monitored data. The *Representation* cluster is concerned with the run-time representation of adaptation targets, inputs, effects, system indicators, and states. The *Control* cluster is concerned with the mechanisms through which a solution is enacted. The *Identification* cluster is concerned with the possible solution instances that can achieve the adaptation targets. Finally, the cluster

of *Enacting Adaptation* concerns how adaptation can be achieved. While we hope our enumeration will help formalize and advance the understanding for self-adaptive system design, it is not intended to be complete and further work on expanding and refining this design space is necessary and appropriate.

To explain the concepts, we separate self-adaptive systems into two elements: the Managed System, which is responsible for the system's main functionality, and the Adaptation System, which is responsible for altering the Managed System as appropriate. The elements inherent to the managed system (that is, the things that would exist even if it were not adaptively managed) such as the inputs and the environment are captured and used by the Adaptation System. The Adaptation System produces adaptations that impact the Managed System.

Observation. The observation cluster is concerned with design decisions regarding what information is observed by the self-adaptive system and when such observations are made.

A key design decision about self-adaptive systems is "what information will the system observe?" In particular, "what information about the external environment and about the system itself will need to be measured or estimated?" To make these measurements, the system will need a set of sensors; these determine what the system *can* observe. Some of the measurements can be made implicitly (e.g., by inferring them from the state of the system or success or failure of an action). Choices include different aspects of goals, domain knowledge, environment, and the system itself necessary to make decisions about adaptation toward meeting the adaptation goals.

Given the set of information the system observes, another important design decision is "how will the system determine that information?" The system could make direct measurements with sensors, infer information from a proxy, extrapolate based on earlier measurements, aggregate knowledge, etc.

Given a way to observe, there are two important decisions that relate to timing: "what triggers observation?" and "what triggers adaptation?" The system could be continuously observing or observation could be triggered by an external event, a timer, an inference from a previous observation, deviation or error from expected behavior, etc. Thus, the observation can happen at a fixed delay, on-demand, or employ a best-effort strategy. The same decisions relate to when the adaptation triggers, which is also relevant to the control cluster.

Handling uncertainty in the measurements is another decision related to observation. Filtering, smoothing, and redundancy are just some of the solutions to dealing with noise and uncertainty.

Representation. The representation cluster is concerned with design decisions regarding how the problem and system are represented at run-time. Uncertainty is intrinsic to many self-adaptive systems, so some information may be available only at run-time. Therefore, to enable mechanisms of adaptation, key information about the problem and system may have to be made available at run-time. This cluster is closely related to the observation cluster, and has ties to the remaining clusters that make use of this representation.

A key decision in this cluster is "what information is made available to the components of the self-adaptive system?" Answers include different aspects of the adaptation targets, existing domain knowledge, and observations of the environment and Managed System that are found necessary and sufficient for a self-adaptive system to operate.

Another design decision in this cluster relates to choices of internal representation. "How is this information represented?" is meant to guide the designer to the representation that best matches the adaptation targets and the nature of the problem. Choices include explicit representations such as graph models, formulae, bounds, objective functions, etc., or implicit representations in the code.

Control. The control cluster is concerned with the system's run-time decision making toward self-adaptation.

"How to compute how much change to enact forms one design decision in this cluster?" Possible choices include the change being a predefined constant value or proportional to the deviation from the desired behavior. The PID technique adds three values to determine the amount of change: a value proportional to the control error, a value proportional to the derivative of the error, and a value proportional to the integral of the error.

Feedback loops play an integral role in adaptation decisions. Thus, key decisions about a self-adaptive system's control are: "what control loops are involved?" and "how do those control loops interact?" The choices depend on the structure of the system and the complexity of the adaptation goals. Control loops can be composed in series, parallel, multi-level (hierarchical), nested, or independent patterns. Brun *et al.* [10] have further discussed the choices and impact of control loops on the design of self-adaptive systems.

What aspects of the system can be adapted from another design decision? Systems can change parameters, representations, and resource allocations, choose among pre-constructed components and connectors, synthesize new components and connectors, and augment the system with new sensors and actuators.

The possible adaptations those aspects can undergo form another design decision. Choices include aborting, modifying data, calling procedures, starting new processes, etc.

The design decision from the observation cluster that deals with what triggers adaptation is closely related to the control cluster.

Identification. At every moment in time, the self-adaptive system is in one instantiation. The self-adaptation process consists of traversing the space of such instantiations. The identification cluster is concerned with identifying those instantiations the system may take on at run-time. Instantiations can describe system structure, behavior, or both.

For each adaptation target, there is a decision about which instantiations could satisfy that target. The main concern of this decision is enumerating concrete sets of possible structures, behaviors, states, parameter values, etc. It is

likely that not all identified instantiations will be supported at run-time. Selecting those that will be supported is another design decision.

Identifying the relevant domain assumptions and contexts for each instantiation is yet another design decision in this cluster. The system can then recognize the context and enact the relevant instantiations.

Finally, identifying the transition cost between instantiations informs the system of the run-time costs of certain types of self-adaptation.

Enabling Adaptation. The choice of adaptation mechanisms the self-adaptive system employs, how are they triggered and supported and how failure is handled are design decisions included in the Enabling Adaptation cluster.

The mechanisms can be represented explicitly or implicitly in the system. For example, self-managing systems with autonomic components typically have explicit adaptation mechanisms. Meanwhile, self-organizing systems often exhibit self-adaptation as an emergent property and do not explicitly define the adaptation mechanisms. The decision concerning control loops from the control cluster is closely related to this decision. Some forms of control can be explicitly expressed in the design, whereas other forms are emergent. It is also possible to create hybrid explicit-implicit self-adaptive systems.

Support of the self-adaptation forms another design decision. Support can be enacted through plugin architectures, component substitution, web services, etc. Related to this decision is what to do when adaptation fails. Choices include trying again, trying a different adaptation mechanism or strategy, observing the environment and the system itself to update internal representations, etc.

In selecting the adaptation mechanisms, it is important to consider the states or events that can trigger adaptation. Examples triggers include not satisfying the adaptation targets that relate to non-functional requirements (e.g., response time, throughput, energy consumption, reliability, fault tolerance), behavior, undesirable events, state maintenance, anticipated change, and unanticipated change.

2.3 Research Challenges

The design space described above can help formalize and advance the understanding of self-adaptive system design. However, it is incomplete, so further exploration and expansion are necessary to aid self-adaptive system developers. A more complete list can help ensure designers avoid leaving out critical decisions.

The main benefit of understanding the design space is infusing a systematic understanding of the options for self-adaptive control into the design process. The developer should understand the trade-offs among different options and the quantitative and qualitative implications of the design choices. To do this effectively, we need to understand the effects of these design decisions, and their order, on the quality of the resulting system.

Each cluster we outlined above can be further expanded and refined. Further, validation of the alternatives against real-world examples can serve as the

framework for describing options. Dimensions in the self-adaptive design space are not independent and the interactions between the decisions in those clusters need to be explored. Understanding the decision relationships can narrow the search space and reduce the complexity of the design and the design process.

An important challenge to consider is bridging the gap between the design and the implementation of self-adaptive systems. Frameworks and middleware (e.g., [18,38]) can help bridge that gap, providing developers with automatically generated code and reusable models for their specific design decisions. This challenge is even more difficult in the case of reengineering existing non-self-adaptive systems or integrating self-adaptive and non-self-adaptive systems.

Finally, of particular importance is the understanding of interactions of control loops and self-adaptation mechanisms. If we are to build complex systems, and systems-of-systems with self-adaptive components, we must understand how these mechanisms and their relevant control loops interact and affect one another.

3 Processes

Software engineering (SE) research has primarily focused on principles for developing high quality software systems, while the maintenance and the evolution of such systems have received less attention [40]. Meanwhile, it has been commonly accepted that software, which implements real world applications, must continually evolve. If software does not evolve, it will not fulfill the continuously changing requirements and thus, it will become outdated earlier than expected [35,36]. This awareness has impacted software process models to better address the inherent need for change and evolution by introducing iterative, incremental, and evolutionary approaches to software development as an alternative to strictly separating sequenced disciplines of requirements engineering, design, implementation, and testing [34,40].

In the last decade, software maintenance and evolution have emerged as a key research field in SE [40] that separates the time before and the time after the software is delivered, or in other words, divides the software lifecycle into development-time, deployment-time, and run-time. Post-delivery changes are typically realized by re-entering the regular development disciplines, which eventually results in a new version of a software product or a patch that is then released to replace or enhance the currently running version [32]. Such releases are usually performed during scheduled down-times of the system compromising the system's availability. Thus, the whole maintenance process is mainly performed off-line guided by human-driven change management activities and decoupled from the running system.

However, such a lifecycle does not meet the requirements of *self-adaptive software* [15] that we are considering in this work. A self-adaptive software system operates in a highly dynamic world and must adjust its behavior automatically in response to changes in its environment, in its goals, or in the system itself. This vision requires shifting the human role from operational to strategic. Humans define adaptations, that is, how a system should react to changes and the

system then performs adaptations autonomously at run-time. The implication is that activities previously performed during development-time will be shifted to run-time. Several researchers [4,6,27,28] argue that, as a consequence, we have to reconceptualize SE processes to fit modern software systems better. In particular, to fit self-adaptive software systems.

The problem being addressed concerns the timing of software activities [12] in a particular process regarding the software lifecycle. This problem has three dimensions:

- *Software lifecycle phases* [43] (i.e., development, deployment, operation, and maintenance and evolution).
- *Software engineering disciplines* [43] (i.e., requirements, design, implementation, validation, verification, etc.), and activities included in the disciplines (e.g., requirements elicitation, prioritization, and validation).
- *Software activities timeline* [12], that is, when activities take place (i.e., development-time, deployment-time, run-time).

The motivation for our work is that lifecycle activities in a self-adaptive software system are not bound to a traditional timeline (e.g., development-time), but may be shifted to run-time. However, such shifts have uncharted consequences, for instance, they may introduce new process requirements in a different phase (e.g., that additional activities have to be performed during development or deployment of the system to enable shifts of other activities to run-time). Moreover, these consequences must be identified, analyzed, and possibly mitigated, thus resulting in a more dynamic view on software processes. One example of changed timing for activities in self-adaptive systems is verification and validation. The dynamic nature of a running self-adaptive system and its environment requires continuous verification and validation (V&V) to assess the system at run-time. V&V are traditionally performed at development-time and shifting it to run-time requires new and efficient techniques (cf. Section 5). The consequence is a different and more dynamic SE process for self-adaptive systems that needs to be understood and elaborated. Our main challenge is to provide means for engineering processes for self-adaptive systems that will cover the complete software lifecycle. Engineering processes implies support for reasoning about costs and benefits of shifting activities in a process, a prerequisite for engineers to make informed decisions.

3.1 Example: Migrating Evolution Activities

To illustrate the specifics of SE processes for self-adaptive software systems and their differences to traditional software development and evolution activities, we compare the traditional approach to corrective maintenance [50] with the *automatic workarounds* approach [13,14]. Automatic workarounds aim to mask functional faults at run-time by automatically looking for and executing alternative ways to perform actions that lead to failures.

Besides the implementation of new or changing requirements, the evolution of software systems may include corrective maintenance activities [50]. Traditionally, users experience failures and report them to developers who are then in

charge of analyzing the failure report, identifying the root cause of the problem, implementing the changes, and releasing the new fixed version of the software.

In contrast, the automatic workarounds mechanism exploits the intrinsic redundancy of "equivalent operations" usually offered by software systems for different needs, but for obtaining the same functionality. Consider for example a container component that implements an operation to *add a single element*, and another operation to *add several elements*. To add two elements, it is possible to add one element after another, or as an equivalent alternative to add them both at the same time using the other operation. If adding two elements in sequence causes a failure at run-time, the automatic workarounds mechanism tries to execute the equivalent operation instead, as an attempt to avoid the problem.

Thus, the automatic workarounds approach partially moves corrective maintenance activities to run-time. Once the user reports a failure, the automatic workarounds mechanism tries to find a workaround based on that information. It checks whether the failure has been experienced by other users and a workaround is already known. If so, it first attempts to execute the workaround known to be valid. If no workaround is known or the known workaround no longer works, the mechanism scans the list of equivalent operations and checks whether they may serve as workarounds.

The automatic workaround mechanism exemplifies how activities, previously performed decoupled from running system instances by software developers similar to development activities, are now performed at run-time by a managing subsystem in a self-adaptive software system. Another example is the *failure analysis* activity, where failure causes are analyzed. In traditional maintenance, the failure report is analyzed by developers while in a self-adaptive software system, the managing subsystem analyzes the failure to find alternative workarounds. In general, compared to traditional SE processes, adding a managing subsystem affects how activities in lifecycle phases are defined and connected. The automatic workarounds approach exemplifies three effects:

- *Migrating activities from one phase to another* — the analyzing failure reports activity is (partially) moved from development-time (maintenance) phase to run-time. This (partially) delegates the developer's responsibility for this activity to a self-adaptation mechanism in the self-adaptive system and it is an example of the effects on the activities' timeline.
- *Introducing new activities in other lifecycle phases* — introducing the automatic workaround mechanism requires that additional activities are performed in the development and maintenance phases. One example is the identification of equivalent operations. Whenever some behavior is "assigned" to the automatic workaround mechanism, equivalent operations for this behavior must be identified.
- *Defining new lifecycle phase inter-dependencies* — the automatic workaround mechanism searches for equivalent operations, executes them, and lets the users evaluate the results. This is repeated either until the user approves the results, and thus the workaround, or until no more equivalent operations could be found. If this is the case, the mechanism is not able to provide a

solution to the problem. The only fallback available is to generate a failure report and send it to the maintenance organization where it will be dealt within the traditional maintenance activity. This exemplifies how traditional maintenance activities integrate with run-time activities, for instance, as information providers or as fallback activities if run-time activities do not succeed.

3.2 Understanding a Self-Adaptive Software System's Lifecycle

Understanding how software is best developed, operated, and maintained is a pervasive research challenge in the SE field. During the last two decades we have witnessed the development of ultra-large-scale, integrated, embedded, and context-aware software systems that have introduced new challenges concerned with system development, operation, and maintenance. For instance, dynamic environments may change systems' goals, the systems' inherent complexity makes it difficult for external parties to be responsible for the operation, and finally, the vast number of systems makes the operations task too complex for a single centralized machine or a system operator. One answer to these advances is to instrument software systems with *managing systems* that make them more autonomous. This autonomy means that systems take over some of the responsibilities previously performed by other roles in the software lifecycle, such as sensing failures and automatically recovering from them.

An SE process is a workflow of activities that are performed by roles and that use and produce artifacts in order to develop, operate, and evolve a software system. In general, we conceive two extreme poles of SE processes [27,28]. One pole corresponds to a traditional, off-line lifecycle process where the system itself has no on-line process activities, that is, no activities are performed by the running system. In contrast, the other pole describes a process with almost all activities performed on-line by the system at run-time. The distinction between off-line and on-line process activities is pivotal for the design of self-adaptive software systems as it enables engineers to design more sophisticated self-adaptation capabilities. In practice, a process for a self-adaptive software system is positioned in between these two extreme poles, due to cost and benefits trade-offs.

The research we envision has as its goal a generic process engineering framework for self-adaptive software systems that provides reasoning support based on the relative costs and benefits for individual design decisions. The framework should include a library of reusable process elements (i.e., activity, role, and artifact definitions). With its built-in support for reasoning, the framework associates value (costs and benefits) with these process elements. These will guide and support engineers in understanding, specifying, analyzing, tuning, and enacting an SE process for a concrete self-adaptive system. The framework is based on process modeling, where models specify processes. Such process descriptions materialize how a self-adaptive software system is developed and how its managing system behaves at run-time. By reifying a process in models, a framework will promote discussions about the process and its design. In the long term, it will promote reuse and even automated analysis of processes [45], which will further support a better understanding of a self-adaptive system's lifecycle.

The key research challenge is the design of the process engineering framework for self-adaptive software systems, which includes three corner-stone components: (i) A library containing definitions of reusable process elements; (ii) Support for specification of concrete self-adaptive software systems' processes; and (iii) Support for reasoning, analysis, and tuning of such process specifications based on their relative costs and benefits.

Definitions of process elements for the library as well as process specifications should be based on an existing framework, such as the *Software & Systems Process Engineering Metamodel Specification* (SPEM) [43]. SPEM provides a modeling language for process specifications including, among others, lifecycle phases, milestones, roles, activities, and work products. Research needs to identify required extensions to SPEM in order to model specifics of processes for self-adaptive systems, like the phases when process elements are employed and their inter-relationships to elements in other lifecycle phases. For example, in the automatic workarounds approach, we identified the analyze failure report activity as one activity that may be performed as part of a regular maintenance phase or at run-time. Another example is to model dependencies between phases (e.g., an activity can only be performed at run-time if another activity has been performed at development-time). In addition, research on how to integrate notions of value, that is, costs and benefits, into SPEM concepts is key. Extensions to SPEM will provide a language for the process engineering framework. A language to define process elements for the library, to model concrete processes for self-adaptive software systems, and to analyze and tune these processes.

The first framework component, defining reusable elements for the generic library, requires a solid understanding of SE processes, self-adaptive systems, and the influential factors such as benefits and costs to a self-adaptive system. This understanding is materialized by those elements that define processes, activities, roles, or artifacts, and is persisted and shared as knowledge, such as best-practices, in the library. Thus, the library supports the understanding and specification of concrete processes by reusing the library's knowledge and element definitions, which is addressed by the second component.

Starting with an abstract conceptual model of the self-adaptive software to be developed and the goals and the environments of the system, an engineer instantiates the library to create a process model for the specific product. The process engineering framework provides methods for decision support and product/process analysis that will assist in the instantiation task. Self-adaptive behavior introduces a complicated bi-directional dependency relation between process modeling and software design. The framework's methods will have to take several factors into consideration including the type of adaptation required at run-time, the associated costs and benefits, and the consequences for other lifecycle activities. In our example, there is a design decision (to use the automatic workaround mechanism) that introduces additional activities as part of the development activities (defining the scope of the mechanism, i.e., which operations should be covered by the mechanism, and identifying equivalent operations for this defined scope).

The third framework component explicitly addresses the product/process analysis and tuning to obtain an enactable process specification that appropriately fits the specific product and the product's goals and environments. A typical sensitivity point is the degree of adaptation and evolution support at run-time. Any design decision concerned with self-adaptive behavior must analyze, for instance, the overhead it introduces. Is the overhead acceptable or not? If not, are precomputed adaptations possible to tune the process by reducing the overhead? As stated in [4], run-time validation and verification may not match the requirements of efficiency to allow the systems to timely react according to changes. This exemplifies that software design and process analysis/tuning are not isolated activities, and it promotes the continuous integration of design decisions and process analysis/tuning throughout a self-adaptive software system's lifecycle.

Finally, it is likely that an engineer uses the three components of the process engineering framework iteratively and concurrently rather than sequentially. For example, while specifying a process, an engineer does not find a suitable process element definition in the library, and thus, new definitions will be created and possibly added to the library. Or during product/process analysis, an engineer identifies the need for process optimization, and searches the library for more suitable process element definitions that could be used to tune the process. Like software development processes, the process of using the framework itself is characterized by incremental, iterative, and evolutionary principles.

Another dimension that should be considered from the beginning is the degree of automation. An absolute requirement is that the process is based on and uses models throughout the lifecycle. Since the system evolves at run-time, these models may also have to evolve (model evolution) and thus, models need to be accessible at run-time, either on-line or off-line [6]. The availability of run-time models makes it possible to use them as interfaces for monitoring [57] and adapting [56] a running system, and to perform what-if analyses and consultations [6] (e.g., to test adaptations at the level of models before actually adapting the running system). In addition, process activities must be based on up-to-date models. Changes in a run-time model allow to some extent for the dynamic derivation of new capabilities to respond to situations unforeseen in the original design. Not all need to be new, we envisage the use of a library of model transformation strategies [4] to derive system models as well as keeping the process up-to-date with respect to the running system and vice versa. As an initial step, model synchronization techniques have already been applied at run-time to keep multiple system models providing different views on a running system up-to-date and consistent to each other [56,57].

3.3 Research Challenges

The different problems and dimensions highlighted in the previous sections can be summarized as research challenges in process comprehension, process modeling, and process design.

First, dynamic environments change the system's goals. As a consequence we need proper means to fully comprehend the characteristics of self-adaptive software systems and the key characteristics of their lifecycles to enhance design & modeling, optimization, and enactment of such systems and processes. For example, more autonomy calls for the capability of self-reacting to anomalous situations. Both probing and reacting must be properly planned, designed, and implemented, and they also require that some activities, which were traditionally performed before releasing the system, be shifted to run-time.

To fully comprehend how software processes change when developing a self-adaptive system also requires that influential factors are identified and understood. Identification of these factors is essential. Factors are costs and benefits related to self-adaptation capabilities. Less complex capabilities may be supported even in a primarily off-line process while more advanced, complex self-adaptation capabilities call for processes where a majority of the activities are performed on-line.

These two challenges impose a proper formalization of the software processes to allow involved parties to fully understand the roles, activities, and artifacts at each stage, but also to increase knowledge and foster reuse. Since some solutions for process definition already exists and SPEM is imposing as one of the most interesting/promising solutions, one should analyze it to understand what can be defined through the standard model, and identify required extensions of this model to take the specifics of processes for self-adaptive systems into account.

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 processes to adapt. However, to deal effectively with the variability associated with software adaptation, it is also necessary to adapt the processes that actually manage the dynamic generation of processes for handling the uncertainty of the changes. This calls for the need to have reflective processes in which a process is adapted by reflecting on itself and its goals. Since off-line and on-line activities might influence each other, another challenge that is identified is the need to consider how the initial development-time design rationale can affect the processes being generated at run-time. The reverse is also crucial, there is the need 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.

A SPEM-like solution is the enabler for defining a suitable library of generic, reusable process elements. The availability of these elements would turn the definition of suitable software processes, for the different self-adaptive systems, into the assembly of pre-existing blocks with a significant gain in terms of quality, speed, and accuracy of delivered solutions. Orthogonally, it would also allow for the analysis and tuning of designed processes to obtain enactable solutions that appropriately fit different products given their specific stakeholders' goals and environments in which they operate. Accurate analysis and optimization capabilities are mandatory to oversee the actual release of these processes, but they are also important to govern evolution since it is foreseeable that these processes

must evolve in parallel with developed systems. Processes must remain aligned and consistent with the corresponding systems and with the environments in which these systems operate. Adequate design support for self-adaptive systems and their lifecycle processes, where value and trade-offs are central, is a remaining grand challenge for engineering self-adaptive software systems.

4 Decentralization of Control Loops

Control loops have been identified as crucial elements to realize the adaptation of software systems [17,30,48]. As outlined in the the former road map [15], a single centralized control component may realize the adaptation of a software system, or multiple control components may realize the adaptation of a composite of software systems in a decentralized manner. In a decentralized setting, the overall system behavior emerges from the localized decisions and interactions. These two cases of self-adaptive behavior, in the form centralized and decentralized control of adaptation are two extreme poles. In practice, the line between the two is rather blurred, and development may result in a compromise. We illustrate this with a number of examples.

Adaptation control can be realized by a simple sequence of four activities: monitor, analyze, plan, and execute (MAPE). Together, these activities form a feedback control system from control theory [47]. A prominent example of such adaptation control is realized in the Rainbow framework [19]. Hierarchical control schemes allow management or the complexity of adaptation when multiple concerns (self-healing, self-protection, etc.) have to be taken into account. In this setting, higher level adaptation controllers determine the set values for the subordinated controllers. A prominent example of a hierarchical control schema is the IBM architectural blueprint [25]. In a fully decentralized adaptation control schema, relatively independent system components coordinate with one another and adapt the system when needed. An example of this approach is discussed in [21] in which component managers on different nodes automatically configure the system's components according to the overall architectural specification.

These examples show that a variety of control schemas for self-adaptive systems are available. Our interest in this section is twofold: first, we are interested in understanding the drivers to select a particular control schema for adaptation; and second, we are interested in getting better insight in the possible solutions to control adaptation in self-adaptive systems. Both the drivers and solutions are important for software engineers of self-adaptive system to choose the right solution concerning centralized or decentralized control. In the remainder of this section, we report on our findings concerning this endeavor and outline some of the major research questions we see to achieve that a systematic engineering approach for designing centralized or decentralized control schemes for software adaptation.

4.1 Distribution versus Decentralization

Before we elaborate on the problems and possible solutions of different control schemas in self-adaptive systems, we first clarify terminology. In particular, we want to clarify the terms distribution and decentralization, two terms that are often mixed by software engineers in the community of self-adaptive systems, leading to a lot of confusion.

Textbooks on distributed systems (e.g., [51]) typically differentiate between centralized data (in contrast to distributed, partitioned, and replicated data), centralized services (in contrast to distributed, partitioned, and replicated services) and centralized algorithms (in contrast to decentralized algorithms).

Our main focus with respect to decentralization is on the algorithmic aspect. In particular, with *decentralization* we refer to a particular type of control in a self-adaptive software system. With control, we mean the decision making process that results in actions that are executed by the self-adaptive system. In a decentralized system there is no single component that has the complete system state information, and the processes make adaptation decisions based only on local information. In a centralized self-adaptive system on the other hand, decisions regarding the adaptations are made by a single component.

With *distribution*, we refer to the deployment of a software system to the hardware. Our particular focus of distribution here is on the deployment of the managed software system. A distributed software system consists of multiple software components that are deployed on multiple processors that are connected via some kind of network. The opposite of a distributed software system is a system consisting of software that is deployed on a single processor.

From this perspective, control in a self-adaptive software system can be centralized or decentralized, independent of whether the managed software is distributed. In practice, however, when the software is deployed on a single processor, the adaptation control is typically centralized. Similarly, decentralized control often goes hand in hand with distribution of the managed software system.

The existing self-adaptive literature and research, in particular those with a software engineering perspective, have by and large tackled the problem of managing either local or distributed software systems in a centralized fashion (e.g., [19,25,44]). While promising work is emerging in decentralized control of self-adaptive software (e.g., [11,21,39,58,59]), we believe that there is a dearth of practical and effective techniques to build systems in this fashion.

It is important to highlight that the adaptation control schema we consider here (from centralized to decentralized control) is just one dimension of the design space of a distributed self-adaptive system. Other aspects of the design space include the actual distribution of the MAPE components, the distribution of the data and supporting services required to realize adaptation, the mechanisms for communication and coordination, etc.

4.2 Drivers for Selecting a Control Schema for Adaptation

Two key drivers for selecting the right control schema for adaptation in self-adaptive systems are the characteristics of the domain and the requirements of the problem at hand.

Domain Characteristics. Specific properties of the domain may put constraints on the selection of a particular control schema for adaptation. We give a number of example scenarios.

- In open systems, it might be the case that no trustworthy authority exists that can realize central control.
- When all information that is required for realizing adaptations is available at the single node, a centralized control schema may be easy to realize. However, in other settings, it might be very difficult or even unfeasible to get centralized access to all the information that is required to perform an adaptation.
- The communication network may be unreliable causing network disruptions that require decision making for adaptations based on local information only.

Requirements of the Problem at Hand. Stakeholder requirements may exclude particular solutions to realize adaptations.

If optimization is high on the priority list of requirements, a centralized approach may be easier to develop and enables optimization to be rather straightforward. On the other hand, in a decentralized approach, meeting global goals is known to be a complex problem. Hence, we have to compromise on the overall optimality in most cases.

For systems in which guarantees about system wide properties are crucial, fully decentralized solutions can be very problematic. Decentralized control imposes difficult challenges concerning consistency, in particular in distributed settings with unreliable network infrastructures. However, if reaction time is a priority, exchanging all monitored data that is required for an adaptation may be too slow (or too costly) in a centralized setting.

When scalability is a key concern, a decentralized solution may be preferable. Control systems with local information scale well in terms of size, and also regarding performance as the collection of information and control implementation are local. In contrast, scalability in a centralized setting is limited as all control information must be collected and processed at the single control point.

A central control scheme is also less robust as it results in a single point of failure. In a decentralized setting, when subsystems get disconnected, they may be able to operate and make decisions based on the local information only, hence increasing robustness.

4.3 Patterns for Interacting Control Loops

Ideally, we would like to have a list of problem characteristics/requirements and then match solutions against these. However, in practice, as stakeholders

typically have multiple, often conflicting requirements, any solution will imply trade-offs.

We have identified different solutions in the form of patterns of interacting control loops in self-adaptive systems. Patterns are an established way to capture design knowledge fostering comprehension of complex systems, and serving as the basis for engineering such systems. Each pattern can be considered as a particular way to orchestrate the control loops of complex self-adaptive software systems, as we explained in Section 2.2.

In order to describe the different patterns, we consider the interactions among the different phases of control loops realized by the MAPE components. Typically only the M and E phases interact with the managed system (to observe and adapt the system respectively). Furthermore, we consider possible peer interactions among phases of any particular type (e.g., interactions between P phases), and interactions among phases that are responsible for subsequent phases (e.g., an A phase interacts with a P phase, or a P phase that interacts with an E phase). According to the different interaction ways we have identified five different patterns that we briefly illustrate in the following.

Pattern 1: Hierarchical Control. In the hierarchical control pattern, the overall system is controlled by a hierarchical control structure where complete MAPE loops are present at all levels of the hierarchy. Generally, different levels operate at different time scales. Lower levels loops operate at a short time scale, to guarantee timely adaptation concerning the part of the system under their direct control. Higher levels operate at a longer time scale and with a more global vision. MAPE loops at different levels interact with each other by exchanging information. The MAPE loop at a given level may pass to the level above information it has collected, possibly filtered or aggregated, together with information about locally planned actions, and may issue to the level below directives about adaptation plans that should be refined into corresponding actions.

This pattern naturally fits systems with a hierarchical architecture. However, independently of the actual system architecture, hierarchical organization of the control system has been proposed (e.g., in [33]) to get a better separation of concerns among different control levels.

Pattern 2: Master/Slave. The master/slave pattern creates a hierarchical relationship between one master that is responsible for the analysis and planning part of the adaptation and multiple slaves that are responsible for monitoring and execution. Figure 1 shows a concrete instance of the pattern with two slaves.

In this case, the monitor components M of the slaves monitor the status of the local managed subsystems and possibly their execution environment and send the relevant information to the analysis component A of the master. A, in turn, examines the collected information and coordinates with the plan component P, when a problem arises that requires an adaptation of the managed system. The plan component then puts together a plan to resolve the problem and coordinates with the execute components (E) on the slaves to execute the actions to the local managed subsystems.

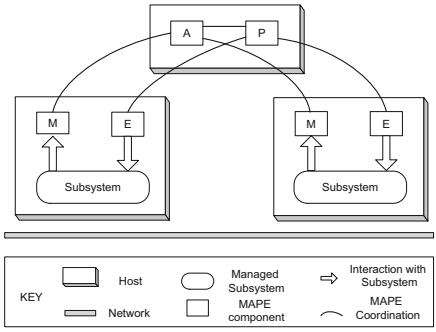

Fig. 1. Master-slave pattern

The master/slave pattern is a suitable solution for application scenarios in which slaves are willing to share the required information to allow centralized decision making. However, sending the collected information to the master node and distributing the adaptation plans may impose a significant communication overhead. Moreover, the solution may be problematic in case of large-scale distributed systems where the master may become a bottleneck.

Pattern 3: Regional Planner. In the regional planner pattern, a (varying) number of local hosts are hierarchically related to a single regional host. The local hosts are responsible for monitoring, analyzing and executing, while the regional host is in charge of the planning part. In this case, the monitor component M of each local host monitors the status of the managed subsystem and possibly its execution environment, and the local analysis component A analyzes the collected information, and reports the analysis results to the associated regional plan component P. P collects this information from all the hosts under its direct supervision, thus acquiring a global knowledge of their status. The regional P is in charge to evaluate the need of adaptation of the managed system and, in case, to elaborate an adaptation plan to resolve the problem, coordinating its decisions with other peer regional plan components. The plan can then be put in action activating the execute components E on the local hosts involved in the adaptation.

Regional planner is a possible solution to the scalability problems with master/slave. Regions may also map to ownership domains where each planner is responsible for the planning of adaptations of its region.

Pattern 4: Fully Decentralized. In this pattern, each host implements a complete MAPE loop, whose local M, A, P and E components coordinate their operation with corresponding peer components of the other hosts. Ideally, this should lead to a flexible sharing of information about the status of the managed systems, as well as the results of the analysis. The triggering of possible adaptation actions is then agreed on and managed by the local P components, which then activate their local E components to execute the actions to the local managed subsystems. In practice, achieving a globally consistent view on the system

status, and reaching a global consensus about suitable adaptation actions is not an easy task. In this case, it could be preferable to limit the interaction among peer control components to get some partial information sharing and some kind of lose coordination. Generally, this may lead to sub-optimal adaptation actions, from the overall system viewpoint. However, depending on the system at hand and the corresponding adaptation goals, even local adaptation actions based on partial knowledge of the global system status may lead to achieve globally optimal goals (TCP adaptive control flow is a typical example of this).

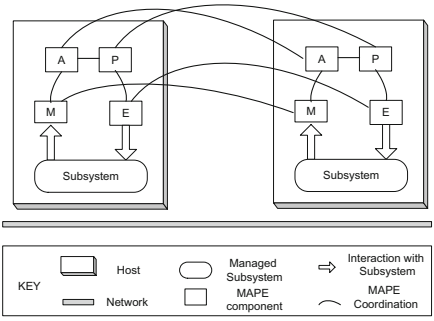

Fig. 2. Decentralized pattern

Pattern 5: Information Sharing. In this pattern, each host owns local M, A, P and E components, but only the monitor components M communicates with the corresponding peer components. Therefore the information collected about the status of the managed systems is shared among the various monitors, while the analysis of the collected data and the decision about possible adaptation actions taken by the plan components P are performed without any coordination action with the other hosts.

Information sharing is for example useful in peer-to-peer systems where peers can perform local adaptations but require some global information. One possible approach to share such global information is by using a gossip approach.

4.4 Outlook

So far, the research community on self-adaptive and autonomic systems has spent little effort in studying the interactions among components of MAPE loops. Our position of making the control loops explicit underlines the need for a disciplined engineering practice in this area. Besides the consolidation of architecture knowledge in the form of different MAPE configurations as patterns, we also need practical interface definitions (signatures and APIs), message formats, and protocols. The necessity of such definitions has partially already been appreciated in the past, e.g., in [37] the authors standardize the communication from the A to the P component by using standard BPEL (Business Process Execution

Language) as the data exchange format, but no comprehensive approach exists so far.

In terms of future research, there are a number of interesting challenges that need to be investigated when considering different self-adaptive control schemes, including:

- *Pattern applicability* — in what circumstances and for what systems are the different patterns of control applicable? Which quality attribute requirements hinder or encourage which patterns? What styles and domains of software are more easily managed with which patterns?
- *Pattern completeness* — what is the complete set of patterns that could be applied to self-management?
- *Quality of service analysis* — for decentralized approaches, what techniques can we use to guarantee system-wide quality goals? What are the coordination schemes that can enable guaranteeing these qualities?

We already mentioned the need for studying other aspects of the design space of adaptation in self-adaptive software systems, including distribution of the MAPE components, distribution of the data and supporting services required to realize adaptation, etc.

Finally, there may be a relationship between the architecture of the managed system and the architecture of the management system. How do we characterize this relationship and help us to choose the appropriate management patterns for the appropriate systems?

5 Practical Run-Time Verification and Validation

In a 2010 science and technology research agenda for the next 20 years, US Air Force (USAF) chief scientist Werner Dahm identified *control science* as a top priority for the USAF [60]. Control science can be defined as a systematic way to study certifiable validation and verification (V&V) methods and tools to allow humans to trust decisions made by self-adaptive systems. According to Dahm, the major barrier preventing the USAF from gaining more capability from autonomous systems is the lack of V&V methods and tools. In other words run-time V&V methods and tools are critical for the success of autonomous, autonomic, smart, self-adaptive and self-managing systems.

While high levels of adaptability and autonomy result in obvious benefits to the stakeholders of software systems, realizing these abilities with confidence is hard. Designing and deploying certifiable V&V methods for self-adaptive systems (SAS) is one of the major research challenges for the software engineering community in general and the self-adaptive systems community in particular. It may take a large part of this decade, if not more, investigating these research challenges to arrive at effective and practical solutions [60].

The V&V roadmap chapter in this book, entitled "Towards Practical Run-time V&V of Self-Adaptive Software Systems," provides a vision of open challenges and discusses run-time V&V challenges from several perspectives:

(i) contrasting design-time and run-time V&V; (ii) defining adaptation properties and viability zone dynamics for SAS; (iii) making V&V explicit in the self-adaptation loops of SAS; (iv) characterizing run-time V&V enablers (i.e., requirements at run-time, models at run-time, and smart context); and (v) ensuring adaptive control.

5.1 Run-Time V&V Research Enablers

Foundational Questions and the Viability Zone. One systematic approach to control science for adaptive systems is to study V&V methods for the mechanisms that sense the dynamic environmental conditions and the target system behavior, and act in response to these conditions by answering the fundamental questions: (i) *what* to validate? (ii) *where* to measure the aspects to validate? and (iii) *when* to validate these aspects? The *what* refers to the system's requirements and adaptation properties that must be validated and verified. The *where* relates to the separation of concerns between the target system and the adaptation mechanism (where V&V must be applied). Finally, the *when* corresponds to the stages of the adaptation process in which V&V tasks are to be performed. The answers to these questions determine the V&V methods that are suitable to keep a particular adaptive system operating within its viability zone. We define the *viability zone* of an adaptive system as the set of possible states in which the system's requirements and desired properties are satisfied [3].

Dependency on Dynamic Context Monitoring. Viability zones are highly dependent on relevant context entities. Relevant context entities provide the attributes to characterize the dimensions of a viability zone.

Viability zones are dynamic. Every time the adaptation process modifies either the target system or the adaptation controller, new variables are added to, or existing ones are replaced by others in the viability zone. Changes in requirements or adaptation goals can affect also the viability zone. Therefore, dynamic context monitoring is an important requirement for run-time V&V tasks, since the coherence of the monitoring infrastructure with respect to the system goals can be compromised. Dynamic context monitoring exploits models and requirements at run-time to maintain an up-to-date and explicit relationship between system requirements and monitoring strategies. This explicit representation and monitoring allow SAS to recognize changes in requirements and then to trigger changes in monitoring strategies accordingly [52,54,55].

5.2 Run-Time V&V Research Directions

Software validation and verification (V&V) concerns the quality assessment of software systems throughout their lifecycle. The goal is to ensure that the software system satisfies its functional requirements and meets its expected quality attributes [7,26]. To establish "certifiable trust" in software systems that adapt themselves according to contextual and environmental changes at run-time, we need powerful and versatile V&V methods, techniques, and tools. A promising

research direction is to ease or relax the traditional software engineering approach, where we satisfy requirements outright, to a more control engineering approach, where we regulate the satisfaction of functional and particularly non-functional requirements using feedback loops [41]. To accomplish this, adaptive software assesses its own situation with respect to its goals continuously, and uses different adaptation strategies accordingly. Nevertheless, the system itself must ensure that its desired behavior is not compromised as a result of the adaptation process. This is particularly important for safety-critical applications.

Quality assessment of self-adaptive software involves both the immutable and the adaptive parts of the system. For the immutable parts, traditional V&V techniques are sufficient. However, for the adaptive parts, the engineering of self-adaptive software requires the development of new, or the tailoring of traditional V&V methods to be applied at run-time throughout the adaptation process. The *Models@run-time* and *Requirements@run-time* research communities provide valuable support for validating and monitoring run-time behavior with respect to the system's requirements [6,46]. The term control science is an appropriate term to characterize this research realm that combines self-adaptation with run-time V&V techniques to regulate the satisfaction of system requirements. It is critical for the SEAMS community to develop a control science involving design-time and run-time V&V methods and techniques for self-adaptive and self-managing systems with inferential, incremental and compositional characteristics that provide adequate confidence levels and certifiable trust in the self-adaptation processes of such systems.

An important first step towards practical run-time validation and verification of self-adaptive software systems is to make V&V tasks explicit in the elements of feedback adaptation loops. This means, for example, to add a V&V component to every phase of the MAPE-K loop [30]. V&V enablers (i.e., requirements at run-time, models at run-time, and dynamic context monitoring) provide effective support to materialize V&V assurances for self-adaptation. Models at run-time enable the validation and monitoring of run-time behavior by providing on-line abstractions of the system state and its relevant context [2,5]. Requirements at run-time provide V&V tasks with on-line representations of the system requirements and adaptation properties throughout the adaptation process [46]. Dynamic context monitoring enables run-time V&V with relevant monitoring mechanisms that keep track of aspects to validate, even when monitoring requirements change at run-time [54].

5.3 Research Challenges

We argue that the fundamental problems addressed by run-time V&V for self-adaptive systems are identical to those of traditional, design-time V&V [20]. That is, independent of the self-* adaptation goals, context awareness, and even uncertainty, V&V fundamentally aims at guaranteeing that a system meets its requirements and expected properties. One key differentiating factor between run-time and design-time V&V is that resource constraints such as time and computing power are more critical for run-time V&V. From these constraints,

non-trivial challenges arise, and to tackle them we should depart of course from traditional V&V methods and techniques. On the one hand, these formal V&V methods are often too expensive to be executed regularly at run-time when the system adapts due to their time and space complexity. On the other hand, context-dependent variables are unbound at design time, but bound at run-time. Thus, performing V&V on these variables at run-time is valuable to reduce the verification space significantly, even when the SAS system viability zone varies with context changes. From this perspective, it is crucial to determine precisely when in the adaptation process these V&V operations are to be performed to guarantee the system properties and prevent unsafe operation.

V&V Techniques: Desirable Properties. Even though traditional V&V techniques (e.g., testing, model checking, formal verification, static and run-time analysis, and program synthesis) have been used for the assessment of quality attributes such as those mapped to adaptation properties by Villegas *et al.* [53], an important challenge is their integration into the self-adaptation lifecycle (i.e., at run-time). This integration requires yet another kind of properties—properties on V&V techniques—including sensitivity, isolation, incrementality, and composability.

According to González *et al.*, sensitivity and isolation refer to the level of run-time testability that an adaptive software system can support [24]. On the one hand, sensibility defines the degree to which run-time testing operations interfere with the running system services delivery. That is, the degree in which run-time V&V may affect the accomplishment of system requirements and adaptation goals. Examples of factors that can affect run-time test sensitivity are (i) component state, not only because run-time tests are influenced by the actual state of the system, but because the state of the system could be altered as a result of test invocations; (ii) component interactions, as the run-time testability of a component may depend on the testability of the components it interacts with; (iii) resource limitations, because run-time V&V may affect non-functional requirements such as performance at undesirable levels; and (iv) availability, as run-time validation can be performed depending on whether testing tasks require exclusive usage of components with high availability requirements. On the other hand, González *et al.* also define isolation as the means to counteract run-time test sensitivity. Instances of techniques for implementing test isolation are (i) state separation (e.g., blocking the component operation while testing takes place, performing testing on cloned components); (ii) interaction separation (e.g., blocking component interactions that may be propagated due to results of test invocations); (iii) resource monitoring (e.g., indicating that testing must be postponed due to resources unavailability); and (iv) scheduling (e.g., planning testing executions when involved components are less used).

Requirements and Models at Run-Time. Requirements define the objectives of validation and verification for software systems. However, adaptive systems requirements are dynamic and subject to change at run-time. Thus, these systems require suitable V&V techniques to cope with the dynamics

after behavioral and structural changes. From this perspective, the application of run-time automatic testing techniques to enable adaptive software systems with self-testing capabilities seems to be a promising approach. An instance of this approach is the self-testing framework for autonomic computing systems proposed by King *et al.* [31]. This framework dynamically validates change requests in requirements using regression testing and customized tests to assess the behavior of the system under the presence of added components. For this, autonomic managers designed for testing are integrated into the current workflow of autonomic managers designed for adaptation. Two strategies support their validation process: (i) safe adaptation with validation, and (ii) replication with validation. In the first strategy, testing autonomic managers apply an appropriate validation policy during the adaptation process where involved managed resources are blocked until the validation is completed. If the change request is accepted, the corresponding managed resources are adapted. In the second strategy, the framework maintains copies of the managed resources for validation purposes. Thus, changes are implemented on copies, then validated, and if they are accepted, the adaptation is performed. Testing policies can also be defined by administrators and loaded into the framework at run-time.

This self-testing approach illustrates the blurred boundaries among the software lifecycle phases and the many implications of V&V for self-adaptive software systems. Some of these implications constitute challenges that arise from requirements engineering, and model-driven engineering. First, run-time V&V tasks rely on on-line representations of the system and its requirements. Second, requirements at run-time support requirements traceability to identify incrementally what to validate, the requirements subset that has changed, and when. Moreover, test case priority further contributes to refine this incremental validation. Third, for context-aware requirements, run-time models must explicitly define the mapping between environmental conditions that must be monitored at run-time, and corresponding varying requirements. Furthermore, models are useful to support the dynamic reconfiguration of monitoring strategies according to changes in requirements and the execution environment. The Requirements@run-time and Models@run-time research communities provide valuable foundations for run-time V&V of self-adaptive software systems [2,5,46].

Context Uncertainty. To cope with context uncertainty, some of the previously proposed approaches to manage unexpected context changes, fully automated or human-assisted, can be exploited. For instance, Murray *et al.* used feedback loops to cover, with respect to the system requirements, the broadest possible range of system states to transition among them by adaptation operations. Their strategy is to augment robustness by reducing context uncertainty [42]. The approach by Goldsby and Cheng uses state machines to model adaptive systems with transitions as system reconfiguration [23]. Inspired by the adaptability of living organisms, they model systems using UML diagrams and apply digital evolution techniques to generate not only one, but several target states for a given transition, and then assist the user to select the one most appropriate. Thus, they address context uncertainty by generating several possible

target system states with qualitatively different QoS characteristics, all of them satisfying the required QoS conditions.

On the side of exhaustive V&V methods, model checking has been used at design time to verify desired properties or conditions on software systems to overcome the limitations of testing techniques, based on a correctness specification. The well known practical problem of this method is the state explosion, which implies the representation of all of the states of the system behavior. In self-adaptive software, this problem is augmented given its changing nature. In effect, the software structure of this kind of systems is subject to re-configuration operations (e.g., adding/removing components and their interconnections) in response to context changes at run-time. Thus, in contrast to the checking requirements of structural static configuration of traditional software, in self-adaptive systems model checking must be applied to each of the possible configurations produced by adaptation mechanisms.

The validation and verification of self-adaptive software systems at run-time is an urgent necessity, and a huge challenge to establish "certifiable trust" in practical adaptation mechanisms. However, despite the development of run-time V&V methods is necessary and plays an important role in the quest towards achieving effective run-time V&V, they are insufficient. To reason effectively and provide assurances on the behavior of self-adaptive systems at run-time, a promising approach is to combine control and software engineering techniques. We aptly termed this combination of foundational theories and principles for run-time V&V methods *control science.*

This section discussed important challenges and possible roadblocks for run-time validation and verification of self-adaptive systems. First, the traceability of evolving requirements, and run-time representations of system aspects are crucial for the identification of what to validate and when. Concrete issues concerning the answers to these questions appear when deciding in which phase of the adaptation loop to implement run-time V&V techniques. Second, these techniques must exhibit desirable properties thus increasing their complexity. Third, dynamic instrumentation such as dynamic monitoring is also required to realize run-time V&V techniques to be implemented throughout the adaptation process.

The assessment of research approaches on self-adaptive software systems constitutes an important starting point for the development of standardized and objective certification methods. For this, we believe that the evaluation framework proposed by Villegas *et al.* provides useful guidance and insights [53]. The SEAMS community is ideally positioned to conduct ground-breaking control science research in our quest towards certifiable trust in self-adaptation.

6 Overall Challenges

In this section, we present the overall conclusions of the research roadmap paper in the context of the major ensuing challenges for our community. First and foremost, this exercise was not intended to be exhaustive. We decided to focus

on four major topics identified as key to engineering of self-adaptive software systems: design space of self-adaptive solutions, software engineering processes for self-adaptive systems, from centralized to decentralized control, and practical run-time verification and validation (V&V) for self-adaptive systems. We now summarize the most important challenges for each topic.

- *Design space* — a major challenge associated with design space is to infuse a systematic understanding of the alternatives for adaptive control into the design process. Since the alternatives could be represented as clusters of design decisions, another challenge should be the detailed refinement of dimensions that characterize these clusters in order to provide a complete set of choices to the developer. Moreover, since dimensions should not be dependent, the search space for the solution can be reduced by identifying the dependencies between the different dimensions. Another identified challenge is how to map a generalized design space into an implementation.
- *Processes* — there are two key challenges related to software processes for self-adaptive systems, first, to have a full understanding of the nature of system, its goals and lifecycle in order to establish appropriate software processes, and second, to understand how processes changes and what are the factors affecting these changes. Another major challenge is the formalization of processes for understanding the roles, activities, and artifacts at each stage of the process. This formalization would enable the definition of a library of generic and reusable entities that could be used across different self-adaptive software systems, and would also facilitate the analysis and tuning of processes according to the system.
- *Decentralization of control loops* — since the direction taken in this topic was the identification of patterns for capturing the interaction of control loops in self-adaptive systems, most of the challenges identified are associated with patterns. For example, concerning pattern applicability, what are the circumstances that decide the applicability of patterns, and what application domains or architectural styles that are better managed by patterns? Also there is the challenge of identifying a complete set of patterns that could be applied to the management of self-adaptive systems. Outside the context of patterns, when considering decentralized approach, a major challenge would be to identify techniques that can be used for guaranteeing system-wide quality goals, and the coordination schemes that enable guaranteeing these qualities.
- *Practical run-time verification and validation* — three key challenges related to the run-time verification and validation of self-adaptive software systems were identified. The first challenge is associated with the need to trace the evolution of requirements in order to identify what and when to validate, and the V&V method to be employed. The second challenge is to control the inevitable complexity that is expected from run-time V&V techniques, and final challenge is related to the need of providing appropriate dynamic monitoring when employing run-time V&V techniques.

There are several topics related to software engineering for self-adaptive systems that we did not cover, some of which we now mention, and which can be considered key challenges on their own. First, how to design in an integrated way self-adaptive system in order to enable them to handle expected and unexpected changes? For example, when composing systems designs should provide some elasticity in order to improve their robustness when reacting to changes. Another issue related to system design is whether adaptation should be reactive or proactive. Further, how should competition and cooperation be managed? How to integrate development-time and run-time V&V in order to provide the necessary assurances before deploying a self-adaptive system? Still related to run-time V&V, what kind of processes and how these should be deployed in order to manage the collection, structuring and analysis of evidence? One of the key activities of feedback control loops in self-adaptive software systems is decision making, and its associated adaptation techniques and criteria for balancing, for example, quality of services, over-provisioning, and cost of ownership. Underpinning all the above issues is the question what shape should take a comprehensive system theory, or theories, for self-adaptive software systems [16]? We also did not cover technologies like model-driven development, aspect-oriented programming, and software product lines. These technologies might offer new opportunities and approaches in the development of self-adaptive software systems. Finally, we did not discuss exemplars — canonical problems and accompanying self-adaptive solutions — which are a likely stepping stone to the necessary benchmarks, methods, techniques, and tools to solve the challenges of engineering self-adaptive software systems.

The four topics discussed in this paper outline challenges that our community must face in engineering self-adapting software systems. All these challenges result from the dynamic nature of self-adaptation, which brings uncertainty to the forefront of system design. It is this uncertainty that challenges the applicability of traditional software engineering principles and practices, but motivates the search for new approaches for developing, deploying, managing and evolving self-adaptive software systems.

References

1. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Modeling Dimensions of Self-Adaptive Software Systems. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 27–47. Springer, Heidelberg (2009)
2. Aßmann, U., Bencomo, N., Cheng, B.H.C., France, R.B.: Models@run.time (Dagstuhl Seminar 11481). Dagstuhl Reports 1(11), 91–123 (2012), http://drops.dagstuhl.de/opus/volltexte/2012/3379
3. Aubin, J., Bayen, A., Saint-Pierre, P.: Viability Theory: New Directions. Springer, Heidelberg (2011), http://books.google.ca/books?id=0YpZNVBXNK8C
4. Baresi, L., Ghezzi, C.: The disappearing boundary between development-time and run-time. In: Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research (FoSER 2010), pp. 17–22. ACM, New York (2010), http://doi.acm.org/10.1145/1882362.1882367

5. Bencomo, N., Blair, G., France, R., Muñoz, F., Jeanneret, C.: 4th International Workshop on Models@run.time. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 119–123. Springer, Heidelberg (2010)
6. Blair, G., Bencomo, N., France, R.B.: Models@run.time: Guest Editors' Introduction. IEEE Computer 42(10), 22–27 (2009)
7. Bourque, P., Dupuis, R.: Guide to the Software Engineering Body of Knowledge (SWEBOK). IEEE Computer Society (2005), http://www.computer.org/portal/web/swebok/home
8. Brake, N., Cordy, J.R., Dancy, E., Litoiu, M., Popescu, V.: Automating discovery of software tuning parameters. In: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-managing Systems, SEAMS 2008, pp. 65–72. ACM, New York (2008), http://doi.acm.org/10.1145/1370018.1370031
9. Brooks, F.P.: The Design of Design: Essays from a Computer Scientist, 1st edn. Addison-Wesley Professional (2010)
10. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 48–70. Springer, Heidelberg (2009)
11. Brun, Y., Medvidovic, N.: An architectural style for solving computationally intensive problems on large networks. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2007), Minneapolis, MN, USA (May 2007)
12. 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), http://dx.doi.org/10.1002/smr.319
13. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds for web applications. In: FSE 2010: Proceedings of the 2010 Foundations of Software Engineering Conference, pp. 237–246. ACM, New York (2010)
14. Carzaniga, A., Gorla, A., Pezzè, M.: Self-healing by means of automatic workarounds. In: SEAMS 2008: Proceedings of the 2008 International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 17–24. ACM, New York (2008)
15. Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J., Andersson, J., Becker, B., Bencomo, N., Brun, Y., Cukic, B., Di Marzo Serugendo, G., Dustdar, S., Finkelstein, A., Gacek, C., Geihs, K., Grassi, V., Karsai, G., Kienle, H.M., Kramer, J., Litoiu, M., Malek, S., Mirandola, R., Müller, H.A., Park, S., Shaw, M., Tichy, M., Tivoli, M., Weyns, D., Whittle, J.: Software Engineering for Self-Adaptive Systems: A Research Roadmap. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Self-Adaptive Systems. LNCS, vol. 5525, pp. 1–26. Springer, Heidelberg (2009)
16. Dobson, S., Sterritt, R., Nixon, P., Hinchey, M.: Fulfilling the vision of autonomic computing. Computer 43(1), 35–41 (2010)
17. Dobson, S., Denazis, S., Fernández, A., Gaïti, D., Gelenbe, E., Massacci, F., Nixon, P., Saffre, F., Schmidt, N., Zambonelli, F.: A survey of autonomic communications. ACM Transactions on Autonomous and Adaptive Systems (TAAS) 1, 223–259 (2006)
18. Elkhodary, A., Esfahani, N., Malek, S.: FUSION: A framework for engineering self-tuning self-adaptive software systems. In: Proceedings of the 18th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2010), Santa Fe, NM, USA, pp. 7–16 (2010)

19. Garlan, D., Cheng, S.W., Huang, A.C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. IEEE Computer 37, 46–54 (2004)
20. Gat, E.: Autonomy software verification and validation might not be as hard as it seems. In: Proceedings 2004 IEEE Aerospace Conference, pp. 3123–3128 (2004)
21. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: 1st Workshop on Self-Healing Systems. ACM, New York (2002)
22. Ghanbari, H., Litoiu, M.: Identifying implicitly declared self-tuning behavior through dynamic analysis. In: International Workshop on Software Engineering for Adaptive and Self-Managing Systems, pp. 48–57 (2009)
23. Goldsby, H.J., Cheng, B.H.C.: Automatically Generating Behavioral Models of Adaptive Systems to Address Uncertainty. In: Czarnecki, K., Ober, I., Bruel, J.-M., Uhl, A., Völter, M. (eds.) MODELS 2008. LNCS, vol. 5301, pp. 568–583. Springer, Heidelberg (2008), http://dx.doi.org/10.1007/978-3-540-87875-9_40
24. González, A., Piel, E., Gross, H.G.: A Model for the Measurement of the Runtime Testability of Component-Based Systems. In: Proceedings of 2009 International Conference on Software Testing Verification and Validation Workshops, pp. 19–28. IEEE (2009), http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4976367
25. IBM: An architectural blueprint for autonomic computing. Tech. rep. IBM (January 2006)
26. IEEE: Industry implementation of international standard ISO/IEC 12207:95, standard for information technology-software life cycle processes. Tech. rep. IEEE (1996)
27. Inverardi, P.: Software of the Future Is the Future of Software? In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 69–85. Springer, Heidelberg (2007), http://dx.doi.org/10.1007/978-3-540-75336-0_5
28. Inverardi, P., Tivoli, M.: The Future of Software: Adaptation and Dependability. In: De Lucia, A., Ferrucci, F. (eds.) ISSSE 2006-2008. LNCS, vol. 5413, pp. 1–31. Springer, Heidelberg (2009), http://www.springerlink.com/content/g624t1466m9v5647/
29. Ionescu, D., Solomon, B., Litoiu, M., Iszlai, G.: Observability and controllability of autonomic computing systems for composed web services. In: 6th IEEE International Symposium on Applied Computational Intelligence and Informatics, SACI 2011 (2011)
30. Kephart, J.O., Chess, D.M.: The vision of autonomic computing. Computer 36(1), 41–50 (2003)
31. King, T.M., Ramirez, A.E., Cruz, R., Clarke, P.J.: An Integrated Self-Testing Framework for Autonomic Computing Systems. Journal of Computers 2(9), 37–49 (2007), http://academypublisher.com/ojs/index.php/jcp/article/view/361
32. Kitchenham, B.A., Travassos, G.H., von Mayrhauser, A., Niessink, F., Schneidewind, N.F., Singer, J., Takada, S., Vehvilainen, R., Yang, H.: Towards an ontology of software maintenance. Journal of Software Maintenance: Research and Practice 11(6), 365–389 (1999), http://dx.doi.org/10.1002/(SICI)1096-908X(199911/12)11:6<365::AID-SMR200>3.0.CO;2-W
33. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, pp. 259–268. IEEE Computer Society, Washington, DC (2007)

34. Larman, C., Basili, V.R.: Iterative and Incremental Development: A Brief History. IEEE Computer 36(6), 47–56 (2003), http://doi.ieeecomputersociety.org/10.1109/MC.2003.1204375
35. Lehman, M.M.: Software's Future: Managing Evolution. IEEE Software 15(01), 40–44 (1998)
36. Lehman, M.M., Belady, L.A. (eds.): Program evolution: processes of software change. Academic Press Professional, Inc., San Diego (1985)
37. Leymann, F.: Combining Web Services and the Grid: Towards Adaptive Enterprise Applications. In: Castro, J., Teniente, E. (eds.) First International Workshop on Adaptive and Self-Managing Enterprise Applications (ASMEA 2005) - CAiSE Workshop, pp. 9–21. FEUP Edi cões (June 2005), http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=INPROC-2005-123&engl=1
38. Malek, S., Edwards, G., Brun, Y., Tajalli, H., Garcia, J., Krka, I., Medvidovic, N., Mikic-Rakic, M., Sukhatme, G.: An architecture-driven software mobility framework. Journal of Systems and Software 83(6), 972–989 (2010)
39. Malek, S., Mikic-Rakic, M., Medvidovíc, N.: A Decentralized Redeployment Algorithm for Improving the Availability of Distributed Systems. In: Dearle, A., Savani, R. (eds.) CD 2005. LNCS, vol. 3798, pp. 99–114. Springer, Heidelberg (2005)
40. Mens, T.: Introduction and Roadmap: History and Challenges of Software Evolution. In: Software Evolution, ch.1. Springer (2008), http://www.springerlink.com/content/978-3-540-76439-7
41. Müller, H.A., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Proceedings of Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), pp. 23–27. ACM/IEEE (2008)
42. Murray, R.M., Åström, K.J., Boyd, S.P., Brockett, R.W., Stein, G.: Future Directions in Control in an Information Rich World. IEEE Control Systems 23, 20–33 (2003)
43. Object Management Group (OMG): Software & Systems Process Engineering Meta-Model Specification (SPEM), Version 2.0 (2008)
44. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems 14, 54–62 (1999), http://dx.doi.org/10.1109/5254.769885
45. Osterweil, L.J.: Software processes are software too. In: Proceedings of the 9th International Conference on Software Engineering (ICSE 1987), pp. 2–13. IEEE Computer Society Press, Los Alamitos (1987), http://portal.acm.org/citation.cfm?id=41765.41766
46. Sawyer, P., Bencomo, N., Whittle, J., Letier, E., Finkelstein, A.: Requirements-Aware Systems. A Research Agenda for RE For Self-Adaptive Systems. In: 18th International Requirements Engineering Conference (RE 2010), pp. 95–103. IEEE (2010)
47. Seborg, D.E., Edgar, T.F., Mellichamp, D.A., Doyle III, F.J.: Process Dynamics and Control, 3rd edn. John Wiley & Sons (1989)
48. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
49. Shaw, M.: The role of design spaces in software design (2011) (submitted for publication)

50. Swanson, E.B.: The dimensions of maintenance. In: Proceedings of the 2nd International Conference on Software Engineering (ICSE 1976), pp. 492–497. IEEE Computer Society Press (1976), http://portal.acm.org/citation.cfm?id=800253.807723
51. Tanenbaum, A.S., van Steen, M.: Distributed Systems: Principles and Paradigms, 2nd edn. Prentice-Hall, Inc., Upper Saddle River (2006)
52. Villegas, N.M., Müller, H.A.: Context-driven Adaptive Monitoring for Supporting SOA Governance. In: 4th International Workshop on a Research Agenda for Maintenance and Evolution of Service-Oriented Systems (MESOA 2010). CMU/SEI-2011-SR-008, Pittsburgh: Carnegie Mellon University (2011), http://www.sei.cmu.edu/library/abstracts/reports/11sr008.cfm
53. Villegas, N.M., Müller, H.A., Tamura, G., Duchien, L., Casallas, R.: A Framework for Evaluating Quality-driven Self-Adaptive Software Systems. In: Proceedings 6th International Symposium on Software Engineering for Adaptive and Self-managing Systems (SEAMS 2011), pp. 80–89. ACM, New York (2011), http://doi.acm.org/10.1145/1988008.1988020
54. Villegas, N.M., Müller, H.A., Tamura, G.: Optimizing Run-Time SOA Governance through Context-Driven SLAs and Dynamic Monitoring. In: 2011 IEEE International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA 2011), pp. 1–10. IEEE (2011)
55. Villegas, N.M., Müller, H.A., Muñoz, J.C., Lau, A., Ng, J., Brealey, C.: A Dynamic Context Management Infrastructure for Supporting User-driven Web Integration in the Personal Web. In: 2011 Conference of the Center for Advanced Studies on Collaborative Research (CASCON 2011), pp. 200–214. IBM Corp., Markham (2011), http://dl.acm.org/citation.cfm?id=2093889.2093913
56. Vogel, T., Giese, H.: Adaptation and Abstract Runtime Models. In: Proceedings of the 5th ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2010), pp. 39–48. ACM (2010), http://portal.acm.org/citation.cfm?id=1808984.1808989
57. Vogel, T., Neumann, S., Hildebrandt, S., Giese, H., Becker, B.: Incremental Model Synchronization for Efficient Run-Time Monitoring. In: Ghosh, S. (ed.) MODELS 2009. LNCS, vol. 6002, pp. 124–139. Springer, Heidelberg (2010), http://www.springerlink.com/content/1518022k168n5055/
58. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS 2011), Honolulu, Hawaii (2011)
59. Weyns, D., Malek, S., Andersson, J.: On decentralized self-adaptation: lessons from the trenches and challenges for the future. In: Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems, SEAMS 2010, pp. 84–93. ACM, New York (2010), http://doi.acm.org/10.1145/1808984.1808994
60. Dahm, W.J.A.: Technology Horizons a Vision for Air Force Science & Technology During 2010-2030. Tech. rep., U.S. Air Force (2010)