Software Engineering for Self-Adaptive Systems: A Research Roadmap

Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, and Jeff Magee
(Dagstuhl Seminar Organizer Authors)

Jesper Andersson, Basil Becker, Nelly Bencomo, Yuriy Brun, Bojan Cukic, Giovanna Di Marzo Serugendo, Schahram Dustdar, Anthony Finkelstein, Cristina Gacek, Kurt Geihs, Vincenzo Grassi, Gabor Karsai, Holger M. Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi A. Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, and Jon Whittle
(Dagstuhl Seminar Participant Authors)

`r.delemos@kent.ac.uk, holger.giese@hpi.uni-potsdam.de`

Abstract. The goal of this roadmap paper is to summarize the state-of-the-art and to identify critical challenges for the systematic software engineering of self-adaptive systems. The paper is partitioned into four parts, one for each of the identified essential views of self-adaptation: modelling dimensions, requirements, engineering, and assurances. For each view, we present the state-of-the-art and the challenges that our community must address. This roadmap paper is a result of the Dagstuhl Seminar 08031 on "Software Engineering for Self-Adaptive Systems," which took place in January 2008.

1 Introduction

The simultaneous explosion of information, the integration of technology, and the continuous evolution from software-intensive systems to ultra-large-scale (ULS) systems require new and innovative approaches for building, running, and managing software systems [1]. A consequence of this continuous evolution is that software systems must become more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, and self-optimizing by adapting to changing operational contexts, environments or system characteristics. Therefore, *self-adaptation* - systems that are able to adjust their behaviour in response to their perception of the environment and the system itself – has become an important research topic.

It is important to emphasize that in all the many initiatives to explore self-adaptive behaviour, the common element that enables the provision of self-adaptability is usually software. This applies to the research in several application areas and technologies such as adaptable user interfaces, autonomic computing, dependable computing, embedded systems, mobile ad hoc networks, mobile and autonomous robots, multi-agent systems, peer-to-peer applications, sensor networks, service-oriented architectures, and ubiquitous computing. It also hold for many research fields, which have already investigated some aspects of self-adaptation from their own perspective, such as fault-tolerant computing, distributed systems, biologically inspired computing, distributed artificial intelligence, integrated management, robotics, knowledge-based systems, machine learning, control theory, etc. In all these case software's flexibility allows such heterogeneous applications; however, the proper realization of the self-adaptation functionality still remains a significant intellectual challenge and only recently have the first attempts in building self-adaptive systems emerged within specific application domains. Moreover, little endeavour has 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 of future generations of self-adaptive systems. Therefore it is worthwhile to identify the commonalities and differences of the results achieved so far in the different fields and look for ways to integrate them.

The goal of this roadmap paper is to summarize and point out the current state-of-the-art and its limitations, as well as to identify critical challenges for engineering self-adaptive software systems. Specifically, we intend to focus on development methods, techniques, and tools that we believe are required to support the systematic development of complex software systems with dynamic self-adaptive behaviour. 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 to identify the main research challenges for the systematic software engineering of self-adaptive systems.

To present and motivate these challenges, the paper divided into four parts, one for each of the four essential views of self-adaptation we have identified. For each view, we present the state-of-the-art and the challenges our community must address. The four views are: modelling dimensions (Section 2), requirements (Section 3), engineering (Section 4), and assurances (Section 5). Finally, we summarize our findings in Section 6.

2 Modelling Dimensions

Endowing a system with a self-adaptive property can take many different shapes. The way self-adaptation has to be conceived depends on various aspects, such as, user needs, environment characteristics, and other system properties. Understanding the problem and selecting a suitable solution requires precise models for representing important aspects of the self-adaptive system, its users, and its environment. A cursory review of the software engineering literature attests to the wide spectrum of software systems that are argued to be self-adaptive. Indeed, there is a lack of consensus among researchers and practitioners on the points of variation among such software systems. We refer to these points of variations as modelling dimensions.

In this section, we provide a classification of modelling dimensions for self-adaptive systems. Each dimension describes a particular aspect of the system that is relevant for self-adaptation. Note that it is not our ambition to be exhaustive in all possible dimensions, but rather to give an initial impetus towards defining a framework for modelling self-adaptive systems. The purpose is to establish a baseline from which key aspects of different self-adaptive system can be easily identified and compared. A more elaborated discussion of the ideas presented in this section can be found in [2].

In the following, we present the dimensions in term of four groups. First, the dimensions associated with self-adaptability aspects of the system goals, second, the dimensions associated with causes of self-adaptation, third, the dimensions associated with the mechanisms to achieve self-adaptability, and fourth, the dimensions related to the effects of self-adaptability upon a system. The proposed modelling framework is presented in the context of an illustrative case from the class of embedded systems, however, these dimensions can be equally useful in describing the self-adaptation properties, for example, of an IT change management system.

2.1 Illustrative Case

As an illustrative scenario, we consider the problem of obstacle/vehicle collisions in the domain of unmanned vehicles (UVs). A concrete application could be the DARPA Grand Challenge contest [3]. Each UV is provided with an autonomous control software system (ACS) to drive the vehicle from start to destination along the road network. The ACS takes into account the regular traffic environment, including the traffic infrastructure and other vehicles. The scenario we envision is the one in which there is a UV driving on the road through a region where people and animals can cross the road unexpectedly. To anticipate possible collisions, the ACS is extended with a self-adaptive control system (SCS). The SCS monitors the environment and controls the vehicle when a human being or an animal is detected in front of the vehicle. In case an obstacle is detected, the SCS manoeuvres the UV around the obstacle negotiating other obstacles and vehicles. Thus, the SCS extends the ACS with self-adaptation to avoid collisions with obstacles on the road.

2.2 Overview of Modelling Dimensions

We give overview of the important modelling dimensions per group. Each dimension is illustrated with an example from the illustrative case.

Goals. Goals are objectives the system under consideration should achieve. Goals could either be associated with the lifetime of the system or with scenarios that are related to the system. Moreover, goals can either refer to the self-adaptability aspects of the application, or to the middleware or infrastructure that supports that application. In the context of the case study mentioned above, amongst several possible goals, we consider, as an example, the following goal:

the system shall avoid collisions. This goal could be expressed in a way in which quantities are associated with the different attributes, and partitioned into sub-goals, with each sub-goal related to one of the attributes.

Evolution. This dimension captures whether the goals can change within the lifetime of the system. The number of goals may change, and the goals themselves may also change as the system as a whole evolves. Hence, goal evolution ranges from static in which changes are not expected, to dynamic in which goals can change at run-time, including the number of goals, i.e., the system is able to manage and create new goals during its lifetime. In the context of the case study since the goal is related to the safety of the UVs it is expected for the goal to be static.

Flexibility. This dimension captures whether the goals are flexible in the way they are expressed. This dimension is related to the level of uncertainty associated with the goal specification, which may range over three values: rigid, constrained, and unconstrained. A goal is rigid when it is prescriptive, while a goal is unconstrained when its statement provides flexibility for dealing with uncertainty. An example of a rigid goal is "the system shall do this..." while an unconstrained goal is "the system might do this..." A constrained goal provides a middle ground, where there is flexibility as long as certain constraints are satisfied, such as, "the system may do this...as long as it does this..." In the context of the case study, the goal is rigid.

Duration. This dimension is concerned with the validity of a goal throughout the system's lifetime. It may range from temporary to persistent. While a persistent goal should be valid throughout the system's lifetime, a temporary goal may be valid for a period of time: short, medium and long term. A persistent goal may restrict the adaptability of the system because it may constrain the system flexibility in adapting to change. A goal that is associated with a particular scenario can be considered a temporary goal. In terms of duration, the goal of the illustrative case can be considered persistent since it is related to the purpose of the system.

Multiplicity. This dimension is related to the number of goals associated with the self-adaptability aspects of a system. A system can either have a single goal or multiple goals. As a general rule of thumb, a single goal self-adaptive system is relatively easier to realize than systems with multiple goals. As discussed in the next dimension, this is particularly true for system where the goals are related. The illustrative case is presented in the context of a single goal.

Dependency. In case a system has multiple goals, this dimension captures how the goals are related to each other. They can be either independent or dependent. A system can have several independent goals (i.e., they don't affect each other). When the goals are dependent, goals can either be complementary with respect to the objectives that should be achieved or they can be conflicting. In the latter case, trade offs have to be analyzed for identifying an optimal configuration of the goals to be met. In the illustrative case study there are no dependencies since there is a single goal.

Change. Changes are the cause of adaptation. Whenever the system's context changes the system has to decide whether it needs to adapt. In line with [4], we consider context as any information which is computationally accessible and upon which behavioural variations depend. Actors (entities that interact with the system), the environment (the part of the external world with which the system interacts [5]), and the system itself may contribute to the context that may influence the behaviour of the application. Actor-dependent, system-dependent, and environment-dependent variations can occur separately, or in any combination. We classify context-dependable changes of a self-adaptive system in terms of the place in which change has occurred, the type and the frequency of the change, and whether it can be anticipated. All these elements are important for identifying how the system should react to change that occurs during run-time. In the context of the illustrative case study, we consider the cause of adaptation the appearance of an obstacle in front of the ACS.

Source. This dimension identifies the origin of the change, which can be either external to the system (i.e., its environment) or internal to the system, depending on the scope of the system. In case the source of change is internal, it might be important to identify more precisely where change has occurred: application, middleware or infrastructure. The source of the change related to the ACS is external to the system.

Type. This dimension refers to the nature of change. It can be functional, non-functional, and technological. Technological refers to both software and hardware aspects that support the delivery of the services. Examples of the three types of change are, respectively: the purpose of the system has changed and services delivered need to reflect this change, system performance and reliability need to be improved, and the version of the middleware in which the application runs has been upgraded. In the illustrative case, since the change can lead ACS to collide against an obstacle the type of change is non-functional.

Frequency. This dimension is concerned with how often a particular change occurs, and it can range from rare to frequent. If for example a change happens quite often this might affect the responsiveness of the adaptation. Since the occurrence of obstacles is rare within the system lifetime, we consider changes are rare to occur.

Anticipation. This dimension captures whether change can be predicted ahead of time. Different self-adaptive techniques are necessary depending on the degree of anticipation: foreseen (taken care of), foreseeable (planned for), and unforeseen (not planned for) [6]. In the illustrative case study, the occurrence of obstacles should be foreseeable.

Mechanisms. This set of dimensions captures the system reaction towards change, which means that they are related to the adaptation process itself. The dimensions associated with this group refer to the type of self-adaptation that is expected, the level of autonomy of the self-adaptation, how self-adaptation is controlled, the impact of self-adaptation in terms of space and time, how responsive is self-adaptation, and how self-adaptation reacts to change.

Type. This dimension captures whether adaptation is related to the parameters of the system's components or to the structure of the system. Based on this, adaptation can be parametric or structural, or a combination of these. Structural adaptation could also be seen as compositional, since it depends on how components are integrated. In the illustrative case, to avoid collisions with obstacles, the SCS has to adjust the movements of the UV, and this might imply adjusting parameters in the steering gear.

Autonomy. This dimension identifies the degree of outside intervention during adaptation. The range of this dimension goes from autonomous to assisted. In the autonomous case, at run-time there is no influence external to the system guiding how the system should adapt. On the other hand, a system can have a degree of self-adaptability when externally assisted, either by another system or by human participation (which can be considered another system). In the illustrative case, for the foreseen type of changes the system is autonomous since the UV has to avoid collisions with animals without any human intervention.

Organization. This dimension captures whether adaptation is performed by a single component - centralized, or distributed amongst several components - decentralized. If adaptation is decentralized no single component has a complete control over the system. The SCS of the UV in the illustrative example seems to fit naturally with a weak organization.

Scope. This dimension identifies whether adaptation is localized or involves the entire system. The scope of adaptation can range from local to global. If adaptation affects the entire system then more thorough analysis is required to commit the adaptation. It is fundamental for the system to be well structured in order to reduce the impact that change might have on the adaptation. In the illustrative case, the adaptation is global to the UV since involves different components in the car, such as, steering gear and brakes.

Duration. This dimension refers to the period of time in which the system is self-adapting, or in other words, how long the adaptation lasts. The adaptation process can be for short (seconds to hours), medium (hours to months), or long (months to years) term. Note that time characteristics should be considered relative to the application domain. While scope dimension deals with the impact of adaptation in terms of space, duration deals with time. Considering that the time it takes for the UV to react to an obstacle is minimal compared with the lifetime of the system, the duration of the self-adaptation should be short term.

Timeliness. This dimension captures whether the time period for performing self-adaptation can be guaranteed, and it ranges from best-effort to guaranteed. For example, in case change occurs quite often, it may be the case that it is impossible to guarantee that adaptation will take place before another change occurs, in these situations best effort should be pursued. In the context of the case study, the upper bounds for the SCS to manoeuvre the UV should be identified for the timeliness associated with self-adaptation to be guaranteed.

Triggering. This dimension identifies whether the change that initiates adaptation is event-trigger or time-trigger. Although it is difficult to control how and when change occurs, it is possible to control how and when the adaptation should react to a certain change. If the time period for performing adaptation has to be guaranteed, then an event-trigger might not provide the necessary assurances when change is unbounded. Obstacles in the illustrative case appear unexpectedly and as such triggering of self-adaptation is event-based.

Effects. This set of dimensions capture what is the impact of adaptation upon the system, that is, it deals with the effects of adaptation. While mechanisms for adaptation are properties associated with the adaptation, these dimensions are properties associated with system in which the adaptation takes place. The dimensions associated with this group refer to the criticality of the adaptation, how predictable it is, what are the overheads associated with it, and whether the system is resilient in the face of change. In the context of the illustrative case study, a collision between an UV and an obstacle may ensue if the SCS fails.

Criticality. This dimension captures the impact upon the system in case the self-adaptation fails. There are adaptations that harmless in the context of the services provided by the system, while there are adaptations that might involve the loss of life. The range of values associated with this criticality is harmless, mission-critical, and safety-critical. The level of criticality of the application (and the adaptation process) is safety-critical since it may lead to an accident.

Predictability. This dimension identifies whether the consequences of self-adaptation can be predictable both in value and time. While timeliness is related to the adaptation mechanisms, predictability is associated with system. Since predictability is associated with guarantees, the degree of predictability can range from non-deterministic to deterministic. Given the nature of the illustrative case, the predictability of the adaptation should be deterministic.

Overhead. This dimension captures the negative impact of system adaptation upon the system's performance. The overhead can range from insignificant to system failure (e.g., thrashing). The latter will happen when the system ceases to be able to deliver its services due to the high-overhead of running the self-adaptation processes (monitoring, analyzer, planning, effecting processes). The overheads associated with the SCS should be insignificant, otherwise the UV might not be able to avoid the obstacle.

Resilience. This dimension is related to the persistence of service delivery that can justifiably be trusted, when facing changes [6]. There are two issues that need to be considered under this dimension: first, it is the ability of the system to provide resilience, and second, it is the ability to justify the provided resilience. The degree of resilience can range from resilient to vulnerable. In the context of the illustrative case study, the system should be resilient.

2.3 Research Challenges in Modelling Dimensions

In spite of the many years of software engineering research, construction of self-adaptive software systems has remained a very challenging task. While substantial progress has been made in each of the discussed modelling dimensions, there are several important research questions that are remaining, and frame the future research in this area. We briefly elaborate on those below. The discussion is structured in line with the four presented groups of modelling dimensions.

Goals. A self-adaptive software system often needs to perform a trade-off analysis between several potentially conflicting goals. Practical techniques for specifying and generating utility functions, potentially based on the user's requirements, are needed. One promising direction is to use preferences that compare situations under Pareto optimal conditions.

Change. Monitoring a system, especially when there are several different QoS properties of interest, has an overhead. In fact, the amount of degradation in QoS due to monitoring could outweigh the benefits of improvements in QoS to adaptation. More research on lightweight monitoring techniques is needed.

Mechanisms. Researchers and practitioners have typically leveraged a single tactic to realize adaptation based on the characteristics of the target application. However, given the unique benefits of each approach, we believe a fruitful avenue of future research is a more comprehensive approach that leverages several adaptation tactics simultaneously.

The application of the centralized control loop pattern to a large-scale software system may suffer from scalability problems. There is a pressing need for decentralized, but still manageable, efficient, and predictable techniques for constructing self-adaptive software systems. A major challenge is to accommodate a systematic engineering approach that integrates both control-loop approaches with decentralized agent inspired approaches.

Responsiveness is a crucial property in real-time software systems, hence the need for adaptation models targeted for real-time systems that treat the duration and overhead of adaptation as first class entities.

Effects. Predicting the exact behaviour of a software system due to run-time changes is a challenging task. More advanced and predictive models of adaptation are needed for systems that could fail to satisfy their requirements due to side-effects of change.

In highly dynamic systems, such as mobile systems, where the environmental parameters change frequently, the overhead of adaptation due to frequent changes in the system could be so high that the system ends up thrashing. The trade-offs between the adaptation overhead and the accrued benefits of changing the system needs to be taken into consideration for such systems.

3 Requirements

A self-adaptive system is able to modify its behaviour according to changes in its environment. As such, a self-adaptive system must continuously monitor changes in its context and react accordingly. But what aspects of the environment should the self-adaptive system monitor? Clearly, the system cannot monitor everything. And exactly what should the system do if it detects less than optimal conditions in the environment? Presumably, the system still needs to maintain a set of high-level goals that should be satisfied regardless of the environmental conditions. But non-critical goals could well be relaxed, thus allowing the system a degree of flexibility during or after adaptation.

These questions (and others) form the core considerations for building self-adaptive systems. Requirements engineering is concerned with what a system should do and within which constraints it must do it. Requirements engineering for self-adaptive systems, therefore, must address what adaptations are possible and what constrains how those adaptations are realized. In particular, questions to be addressed include: what aspects of the environment are relevant for adaptation? Which requirements are allowed to vary or evolve at run-time, and which must always be maintained? In short, requirements engineering for self-adaptive systems must deal with uncertainty because the information about future execution environments is incomplete, and therefore the requirements for the behavior of the system may need to change (at run-time) in response to the changing environment.

3.1 Requirements State-of-the-Art

Requirements engineering for self-adaptive systems appears to be a wide open research area with only a limited number of approaches yet considered. Cheng and Atlee [7] report on some previous work on specifying and verifying adaptive software, and on run-time monitoring of requirements conformance [8,9]. They also explain how preliminary work on personalized and customized software can be applied to adaptive systems (e.g., [10,11]). In addition, some research approaches have successfully used goal models as a foundation for specifying the autonomic behaviour [12] and requirements of adaptive systems [13].

One of the main challenges that self-adaptation poses is that when designing a self-adaptive system, we cannot assume that all adaptations are known in advance — that is, we cannot anticipate requirements for the entire set of possible environmental conditions and their respective adaptation specifications. For example, if a system is to respond to cyber-attacks, one cannot possibly know all attacks in advance since malicious actors develop new attack types all the time.

As a result, requirements for self-adaptive systems may involve degrees of uncertainty or may necessarily be specified as "incomplete." The requirements specification therefore should cope with:

– the incomplete information about the environment and the resulting incomplete information about the respective behaviour that the system should expose
– the evolution of the requirements at run-time

3.2 Research Challenges in Requirements

This subsection highlights a number of short-term and long-term research challenges for requirements engineering for self-adaptive systems. We start with shorter-term challenges and progress to more visionary ideas. As far as the authors are aware, there is little or no research currently underway to address these challenges.

A New Requirements Language. Current languages for requirements engineering are not well suited to dealing with uncertainty, which, as mentioned above, is a key consideration for self-adaptive systems. We therefore propose that richer requirements languages are needed. Few of the existing approaches for requirements engineering provide this capability. In goal-modelling notations such as KAOS [14] and i* [15], there is no explicit support for uncertainty or adaptivity. Scenario-based notations generally do not explicitly support adaptation either, although live sequence charts [16] have a notion of mandatory versus potential behaviour that could possibly be used to specify adaptive systems. Of course, the most common notation for specifying requirements in industry is still natural language prose. Traditionally, requirements documents make statements such as "the system shall do this..." For self-adaptive systems, the prescriptive notion of "shall" needs to be relaxed and could, for example, be replaced with "the system may do this...or it may do that ..." or "if the system cannot do this...then it should eventually do that..." This idea leads to a new requirements vocabulary for self-adaptive systems that gives stakeholders the flexibility to account for uncertainty in their requirements documents. For example:

Traditional RE:
– "*The system shall do this...*"
Adaptive RE:
– "*The system might do this...*"
– "*But it may do this... as long as it does this...*"

– "*The system ought to do this... but if it cannot, it shall eventually do this...*"

Such a vocabulary would change the level of discourse in requirements from prescriptive to flexible. There would need to be a clear definition of terms, of course, as well as a composition calculus for defining how the terms relate to each other and compose. Multimodal logic and perhaps new adaptation-oriented logic [17] need to be developed to specify the semantics for what it means to have the possibility of conditions [18,19]. There is also a relationship with variability management mechanisms in software product lines [20], which also tackle built-in flexibilities. However, at the requirements level, one ideally would capture uncertainty at a more abstract level than simply enumerating alternatives. Some preliminary results in defining a new adaptation requirements language along these lines are being developed [21].

Mapping to Architecture. Given a new requirements language that explicitly handles uncertainty, it will be necessary to provide systematic methods for refining models in this language down to specific architectures that support run-time adaptation. A variety of technical options exist for implementing reconfigurability at the architecture level, including component-based, aspect-oriented and product-line based approaches, as well as combinations of these. Potentially, there could be a large gap in expressiveness between a requirements language that incorporates uncertainty and existing architecture structuring methods. One can imagine, therefore, a semi-automated process for mapping to architecture where heuristics and/or patterns are used to suggest architectural units corresponding to certain vocabulary terms in the requirements.

Managing Uncertainty. In general, once we start introducing uncertainty into our software engineering processes, we must have a way of managing this uncertainty and the inevitable complexity associated with handling so many unknowns. Certain requirements will not change (i.e., invariants), whereas others will permit a degree of flexibility. For example, a system cannot start out as a transport robot and self-adapt into a robot chef [22]! Allowing uncertainty levels when developing self-adaptive systems requires a trade-off between flexibility and assurance such that the critical high-level goals of the application are always met [23,24,25].

Requirements Reflection. As we said above, self-adaptation deals with requirements that vary at run-time. Therefore it is important that requirements lend themselves to be dynamically observed, i.e., during execution. Reflection [26,27,28] enables a system to observe its own structure and behaviour. A relevant research work is the ReqMon tools [29] which provides a requirements monitoring framework, focusing on temporal properties to be maintained. Leveraging and extending beyond these complementary approaches, Finkelstein [22] coins the term "requirements reflection" that would enable systems to be aware of their own requirements at run-time. This capability would require an appropriate model of the requirements to be available online. Such an idea brings with it a host of interesting research questions, such as: Could a system dynamically observe its requirements? In other words, can we make requirements run-time objects? Future work is needed to develop technologies to provide such infrastructure support.

Online Goal Refinement. As in the case of design decisions that are eventually realized at run-time, new and more flexible requirement specifications like the one suggested above would imply that the system should perform the RE processes at run-time, e.g. goal-refinement [25].

Traceability from Requirements to Implementation. A constant challenge in all the topics shown above is dynamic traceability. For example, new operators of a new RE specification language should be easily traceable down to architecture, design, and beyond. Furthermore, if the RE process is performed at run-time we need to assure that the final implementation or behaviour of the system matches the requirements. Doing so is different from the traditional requirements traceability.

The above research challenges the requirements engineering (RE) community will face, as the demand for self-adaptive systems continues to grow, span RE activities during the development phases and run-time. In order to gain assurance about adaptive behaviour, it is important to monitor adherence and traceability to the requirements during run-time. Furthermore, it is also necessary to acknowledge and support the evolution of requirements at run-time. Given the increasing complexity of applications requiring run-time adaptation, the software artefacts with which the developers manipulate and analyze must be more abstract than source code. How can graphical models, formal specifications, policies, etc. be used as the basis for the evolutionary process of adaptive systems versus source code, the traditional artefact that is manipulated once a system has been deployed? How can we maintain traceability among relevant artefacts, including the code? How can we maintain assurance constraints during and after adaptation? How much should a system be allowed to adapt and still maintain traceability to the original system? Clearly, the ability to dynamically adapt systems at run-time is an exciting and powerful capability. The RE community, among other software engineering disciplines, need to be proactive in tackling these complex challenges in order to ensure that useful and safe adaptive capabilities are provided to the adaptive systems developers.

4 Engineering

The engineering of self-adaptive software systems is a major challenge, especially if predictability and cost-effectiveness are desired. However, in other areas of engineering and nature there is a well-known, pervasive notion that could be potentially applied to software systems as well: the notion of feedback.

The first mechanical system that regulated its speed automatically using feedback was Watt's steam engine that had a regulator implementing feedback control principles. Also in nature plenty examples for positive and negative feedback can be found that help to regulate processes.

Even though control engineering [30,31] as well as feedback found in nature are not targeting software systems, mining the rich experiences of these fields and applying principles and findings to software-intensive adaptive systems is a most worthwhile and promising avenue of research for self-adaptive systems. We further strongly believe that self-adaptive systems must be based on this feedback principle and we advocate in this section to focus on the 'control loop' when engineering self-adaptive systems.

In this section we first examine the generic control loop and then analyze the control loop's role in control theory, natural systems, and software engineering, respectively. Finally, we describe the challenges whose resolutions are necessary to enable the systematic engineering of self-adaptive systems. A more detailed elaboration of the perspective presented in this section can be found in [32].

4.1 Control Loop Model

Self-adaptation aspects of software-intensive systems can often be hidden within the system design. What self-adaptive systems have in common is that (1) typically design decisions are partially made at run-time, and (2) the systems reason about their state and environment. This reasoning typically involves feedback processes with four key activities: collect, analyze, decide, and act, as depicted in Figure 1 [33].

Here, we concentrate on self-adaptive systems with feedback mechanisms controlling their dynamic behaviour. For example, keeping web services up and running for a long time requires collecting of information about the current state of the system, analyzing that information to diagnose performance problems or to detect failures, deciding how to resolve the problem (e.g., via dynamic load-balancing or healing), and acting on those decisions.

The generic model of a control loop based on [33] (cf. Figure 1) provides an overview of the main activities around the control loop but ignores properties of the control and data flow around the loop. When engineering a self-adaptive system, questions about these properties become important. We now identify such questions and argue that in order to properly design self-adaptive software systems, these questions must be brought to the forefront of the design process.

The feedback cycle starts with the *collection* of relevant data from environmental sensors and other sources that reflect the current state of the system. Some of the engineering questions that Figure 1 ignores with respect to collection but that are important to the engineering process are: What is the required sample rate? How reliable is the sensor data? Is there a common event format across sensors?

Next, the system *analyzes* the collected data. There are many approaches to structuring and reasoning about the raw data (e.g., using applicable models, theories, and rules). Some of the important questions here are: How is the current state of the system inferred? How much past state may be needed in the future? What data need to be archived for validation and verification? How faithful is the

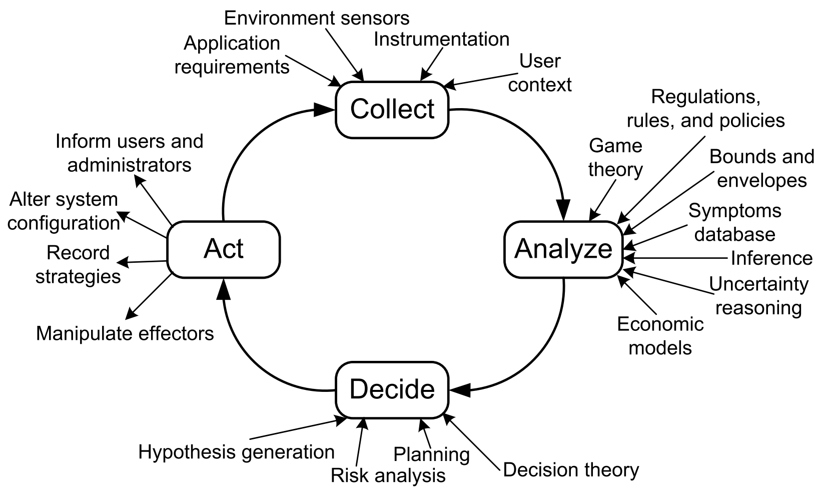

Fig. 1. Activities of the control loop

model to the real world? Can an adequate model be derived from the available sensor data?

Next, the system makes a *decision* about how to adapt in order to reach a desirable state. Approaches such as risk analysis can help make this decision. Here, the important questions are: How is the future state of the system inferred? How is a decision reached (e.g., with off-line simulation or utility/goal functions)? What are the priorities for adaptation across multiple control loops and within a single control loop?

Finally, to implement the decision, the system must *act* via available actuators and effectors. Important questions here are: When should the adaptation be safely performed? How do adjustments of different control loops interfere with each other? Does centralized or decentralized control help achieve the global goal? Does the control system have sufficient command authority over the process—that is, can the action be implemented using the available actuators and effectors?

The above questions—as well as others—regarding the control loop should be explicitly identified, recorded, and resolved during the development of the self-adaptive system.

4.2 Control Loops and Control Theory

The control loop is a central element of control theory, which provides well-established mathematical models, tools, and techniques to analyze system performance, stability, sensitivity, or correctness [34,35]. Researchers have applied results of control theory and control engineering to building self-adaptive systems. However, it is not clear if general principles of this discipline (e.g., open/closed-loop controller, observability, controllability, stability, or hysteresis) are applicable to self-adaptive software systems.

Control engineering has determined that systems with a single control loop are easier to reason about than systems with multiple loops. Unfortunately, the latter types of control loops are far more common. Good engineering practice calls for reducing multiple control loops to a single one, or making control loops independent of each other [36]. When such decoupling is impossible, the design must make the interactions of control loops explicit and expose how these interactions are handled.

Control engineering has also identified hierarchical organization of control loops as a fruitful way to decouple control-loop interactions. The different time scales of the different layers of the hierarchy can minimize the unexpected interference between control loops. This scheme is of particular interest if we distinguish between forms of adaptation such as change management and goal management [25] and can organize them hierarchically.

While mining control engineering for control-loop mechanisms applicable to software engineering can result in breakthroughs in engineering self-adaptive systems, one important obstacle is that different application areas of control engineering introduce distinct nomenclature and architectural diagrams for their realizations of the generic control loop depicted in Figure 1. It is useful to investigate how different application areas realize this generic control loop and to identify the commonalities in order to compare and leverage self-adaptive systems research from different application areas. For example, control engineering has developed standard approaches to model and reason about feedback such as the Model Reference Adaptive Control (MRAC) [31] and the Model Identification Adaptive Control (MIAC) [37].

Models such as MRAC and MIAC introduce well-defined elements such as controller, process, adjustment mechanism, and system identification or model reference along with prescribed dependencies among these elements. This form of separation of concerns suggests that these models are a solid starting point for the design of self-adaptive software-intensive systems. In fact, many participants of the Dagstuhl Seminar 08031 [38] presented self-adaptive systems that can be expressed in terms of standardized models from control engineering such as MRAC and MIAC. Examples of presented systems include a self-adaptive flight-control system that realizes a more robust aircraft control capable of handling multiple faults (e.g., change of aircraft dynamics due to loss of control surface, aileron, or stabilator) [39]; a system of autonomous shuttles that operate on demand and in a decentralized manner using a wireless network [40]; a multi-agent approach to an AGV transportation system that allows agents to flexibly adapt their behavior to changes in their context, realizing cooperative self-adaptation [41]; and the Rainbow system [42], whose architecture was mapped by Shaw to the classical control loop in control theory [43].

4.3 Control Loops and Natural Systems

In contrast to engineered self-adaptive systems, biologically or sociologically inspired systems do not often have clearly visible control loops. Furthermore, the systems are often decentralized in such a way that the agents do not have a sense of the global goal but rather it is the interaction of their local behaviour that yields the global goal as an emergent property.

Nature is full of self-adapting systems that leverage mechanisms and types of control loops far removed from those we use today when engineering self-adaptive systems. Mining this rich collection of systems and creating a catalogue of feedback types and self-adaption techniques is an important and likely fruitful endeavour our community must undertake.

Some software systems that leverage mechanisms found in nature already exist and promise a bright future for nature-inspired software engineering techniques. For example, in systems built using the crystal-growth-inspired tile architectural style [44], components distributed around the Internet come together to "self-assemble" and "self-organize" into a solution to an NP-complete problem. These systems can self-adapt to exhibit properties of fault and adversary tolerance [45]. The self-adaptation control loop is not easily evident in the nature's process of crystal growth, but it does exist and increasing our understanding of such control loops will increase our ability to engineer self-adaptive software systems.

In addition to discovering new self-adaptation mechanisms, mining natural systems and creating a catalogue can facilitate engineering of new novel mechanisms as the combinations of existing ones. For example, while many systems in nature use bottom-up adaptation mechanisms, it may be possible to unify the self-adaptive top-down and self-organizing (bottom-up) mechanisms via software architecture by considering metadata and policies with adaptation properties and control-loop reasoning explicitly, both at design-time and run-time [46].

4.4 Control Loops and Software Engineering

We have observed that control loops are often hidden, abstracted, or internalized when presenting the architecture of self-adaptive systems [43]. However, the feedback behaviour of a self-adaptive system, which is realized with its control loops, is a crucial feature and, hence, should be elevated to a first-class entity in its modelling, design, and implementation.

When engineering a self-adaptive system, the properties of the control loops affect the system's design and architecture. Therefore, besides the control loops, those control loops' properties must be made explicit as well. In one approach, Cheng et al. [47] advocate making self-adaptation external, as opposed to internal or hard-wired, to separate the concerns of system functionality from the concerns of self-adaptation.

Despite recent attention to self-adaptive systems (e.g., several ICSE workshops), development and analysis methods for such systems do not yet provide sufficient explicit focus on the control loops and their associated properties that almost inevitably control self-adaptation.

The idea of increasing the visibility of control loops in software architectures and software methods is not new. Over a decade ago, Shaw compared a software design method based on process control to an object-oriented design method [48].

She introduced a new software organization paradigm based on control loops, one with an architecture that is dominated by feedback loops and their analyses, rather than by the identification of discrete stateful objects.

4.5 Research Challenges in Engineering

We have argued that control loops are essential for self-adaptive systems. Therefore, control loops must become first-class entities when engineering self-adaptive systems. Understanding and reasoning about the control loops of a self-adaptive systems is integral to advancing the engineering of self-adaptive systems' maturation from an ad-hoc, trial-and-error endeavour to a disciplined approach. We identify the following issues as the current most critical challenges that must be addressed in order to achieve a disciplined approach to engineering self-adaptive systems:

Modelling. Making the control loops explicit and exposing self-adaptive properties to allow the designer to reason about the system modelling support for control loops.

Architecture. Developing reference architectures for adaptive systems that address issues such as structural arrangements of control loops (e.g., sequential, parallel, hierarchy, decentralized), interactions among control loops, data flow around the control loops, tolerances, trade-offs, sampling rates, stability and convergence conditions, hysteresis specifications, and context uncertainty.

Design. Compiling a catalogue of common control-loop schemes and characterizing control-loop elements, along with associated obligations in the form of patterns to help classify specific kinds of interacting control loops, e.g., for manual vs. automatic control or for decoupling control loops from one another. These control-loop schemes should come from exploring existing knowledge in control engineering, as well as other fields that use feedback, and from mining naturally occurring systems that use adaptation.

Middleware Support. Developing middleware support to "allow researchers with different motivations and experiences to put their ideas in practice, free from the painful details of low-level system implementation" [49] by supporting a framework and standardized interfaces for self-adaptive functionality.

Verification & Validation. Creating validation and verification techniques to test and evaluate control loops' behaviour and automatically detect unintended interactions.

Reengineering. Exploring techniques for evolving existing systems and injecting self-adaptation into such systems.

Human-Computer Interaction. Analyzing feedback types from human-computer interaction and devising novel mechanisms for exposing the control loops to the users, keeping the users of self-adapting systems "in the loop" to ensure their trust.

5 Assurances

The goal of system assurance is simple. Developers need to provide evidence that the set of stated functional and non-functional properties are satisfied during system's operation. While the goal is simple, achieving it is not. Traditional verification and validation methods, static or dynamic, rely of stable descriptions of software models and properties. The characteristics of self-adaptive systems create new challenges for developing high-assurance systems. Current verification and validation methods do not align well with changing goals and requirements as well as variable software functionality. Consequently, novel verification and validation methods are required to provide assurance in self-adaptive systems.

In this section, we present a generalized verification and validation framework which specifically targets the characteristics of self-adaptive systems. Thereafter, we present a set of research challenges for verification and validation methods implied by the presented framework.

5.1 Assurances Framework

Self-adaptive systems are highly context dependent. Whenever the system's context changes the system has to decide whether it needs to adapt. Whenever the system decides to adapt, this may prompt the need for verification activities to provide continual assessment. Moreover, depending on the dynamics of change, verification activities may have to be embedded in the adaptation mechanism.

Due to the uniqueness of such assessment process, we find it necessary to propose a framework for adaptive system assurance. This framework is depicted in Figure 2. Over a period of operation, the system operates through a series of operational modes. Modes, represented in Figure 2 by index j, represent known and, in some cases, unknown phases in the computational lifecycle. Examples of known modes in flight control include altitude hold mode, flare mode and touchdown mode. Sequences of behavioural adjustments in the known modes are known. But, continuing with the same example, if failures change the airframe dynamics, the application's context changes and software control needs to sense and adapt to the conditions unknown prior to the deployment.

Such adaptations are reflected in a series of context - system state (whatever this is for a self-adaptive system) configurations. $(C+S)_{j_i}$ denotes the i^{th} combination of context and system state in a cycle which is related to the requirements of the system mode j. At the level of configurations it is irrelevant whether the context or the system state changes (transition t_{j_0}), the result always is a new configuration.

Goals and requirements of a self-adaptive system may also change during runtime. We abstract from the subtle differences between goals and requirements

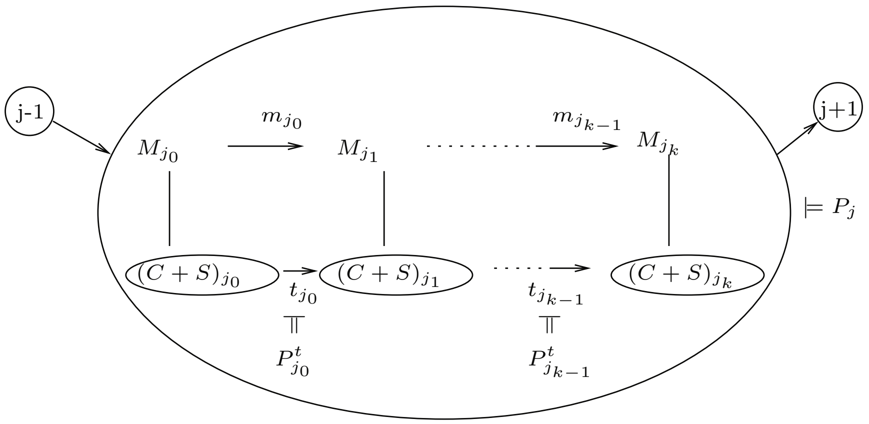

Fig. 2. V & V model

for the generalized framework and instead use the more generic term *properties*. In self-adaptive systems, properties may change over time in response to changes in the system context or the system state. Properties might be relevant in one context and completely irrelevant in some other. Properties might even be weighted against each other, resulting in a trade offs between properties and, thus, their partial fulfilment. Properties can also be related to each other. Global properties like safety requirements must be satisfied over the entire life time of a system, through all the system modes. Different local properties $P^t_{j_i}$ within a context might guarantee a global property. Similarly, a local property may guarantee that a global property is not invalidated by the changes.

Verification of the properties typically relies on the existence of models. System dynamics and changing requirements of self-adaptive systems make it impossible to build a steady model before system deployment and perform all verification tasks on such a model. Therefore, models need to be built and maintained at run-time. In Figure 2, M_{j_i} is the model that corresponds to configuration $(C+S)_{j_i}$. Each change in the system configuration needs to be reflected at model level as well, letting the model evolve accordingly from one configuration to the other, not necessarily linearly as depicted in Figure 2. Delays in model definition may also be inevitable. In Figure 2, the evolution of models from one configuration to the other is denoted by m_{j_i} . The complexity of such evolution moves along two dimensions. On one side the model must be efficiently updated to reflect the system changes, on the other it should still reflect an accurate representation of reality.

5.2 Research Challenges in Assurances

While verification and validation of properties in distributed systems is not a novel problem, a number of additional issues arise in the context of self-adaptation due to the nature of these applications. Self-adapting systems have to contend with dynamic changes in modes and contexts as well as the dynamic changes in user requirements. Due to this high dynamism, V&V methods traditionally applied at requirements and design stages of development must be supplemented with run-time assurance techniques.

Dynamic Identification of Changing Requirements. System requirements can change implicitly, as a result of a change in context. Since in dynamic environments all eventualities cannot be anticipated, self-adapting systems have to be able to identify new contexts. There will inevitably be uncertainty in the process of context identification. Once the context is identified, utility functions evaluate trade-offs between the properties (goals) aligned with the context. The adequacy of context identification and utility functions is subject to verification. It appears that failure detection and identification techniques from distributed fault tolerant computing are a special case of context identification. Given that all such techniques incorporate uncertainty, probabilistic approaches to assurance seem to be the most promising research direction.

Adaptation-Specific Model-Driven Environments. To deal with the challenges of adaptation we envisage a model-driven development, where models play a key role throughout the development [50]. Models allow the application of verification and validation methods during the development process and can support self-adaptation at run-time. In fact, models can support estimation of system's status, so that the impact of a context change can be predicted. Provided that such predictions are reliable, it should be possible to perform model-based adaptation analysis as a verification activity [51]. A key issue in this approach is to keep the run-time models synchronized with the changing system. Any model based verification, therefore, presumes the availability of accurate change tracking algorithms that keep system model synchronized with the run-time environment. Uncertain model attributes can be described, for example, using probability distribution functions, the attribute value ranges, or using the analysis of historical attribute values. These methods can take advantage of probability theory and statistics that helped solve stochastic problems in the past.

Agile Run-Time Assurance. In situations when models that accurately represent the dynamic interaction between system context and state cannot be developed, performing verification activities that address verification at run-time are inevitable. The key requirement for run-time verification is the existence of efficient agile solution algorithms which do not require high space/time complexity. Self-adaptive systems may change their state quickly to respond to context or property changes. An interesting class of verification techniques is that inspired by Proof-Carrying Code (PCC). PCC is a technique by which a host platform can verify that code provided that needs to be executed adheres to a predefined, still limited, set of safety rules. The limitation of the PCC paradigm is that executed code must contain a formal safety proof that attests to the fact that the code respects the defined safety policy. Defining such kind of proofs for code segments which are parameterized and undergo changes and for larger classes of safety properties is a challenge. When formal property proofs do not seem feasible, run-time assurance techniques may rely on demonstrable properties of adaptation, like convergence and stability. Adaptation is a transient behaviour and the fact that a sequence of observable states converge towards a stable state is always desirable. Transient states may not satisfy local or global properties (or we just cannot prove that they do). Therefore, the analysis of the rate of convergence may inspire confidence that system state will predictably quickly reach a desirable state. Here we intentionally use term "desirable" rather than "correct" state because we may not know what a correct adaptation is in an unforeseen context [52]. This problem necessitates investigation of scientific principles needed to move software assurance beyond current conceptions and calculations of correctness.

Liability and Social Aspects. Adaptive functionality in safety-critical systems is already a reality. Applications of adaptive computing in safety critical systems are on the rise [39,53]. Autonomous software adaptation raises new challenges in the legal and social context. Generally, if software does not perform as expected, the creator may be held liable. Depending on the legal theory, different issues will be relevant in a legal inquiry [54]. Software vendors may have a difficult time to argue that they applied the expected care when developing a critical application if the software is self-adaptive. Software may enter unforeseeable states that have never been tested or reasoned about. It can be also argued that current state-of-the-art engineering practices are not sufficiently mature to warrant self-adaptive functionality. However, certain liability claims for negligence may be rebutted if it can be show safety mechanisms could disable self-adaptive functionality in hazardous situations. Assurance of self-adaptive software is then not only a step to make the product itself safer, but should be considered a valid defence against liability claims.

6 Lessons and Challenges

In this section, we present the overall conclusions of the roadmap paper in the context of lessons learned and the major ensuing challenges for our community. First and foremost, we must point out that this exercise had no intention of being exhaustive. We made the choice to focus on the four major issues we identified as the key in the software engineering of self-adaptive systems process: modelling dimensions, requirements, engineering, and assurances.

The presentations of each of the four views intend not to cover all the related aspects, but rather focused theses as a means for identifying the challenges associated with each view. The four identified theses are:

– *modelling dimensions* - the need to enumerate and classify modelling dimensions for obtaining precise models to support run-time reasoning and decision making for achieving self-adaptability;

– *requirements* - the need to define a new requirements language for handling uncertainty to give self-adaptive systems the freedom to do adaptation;
– *engineering* - the need to consider feedback control loops as first-class entities during engineering of self-adaptive systems;
– *assurances* - the need to define novel verification and validation methods for the provision of assurances that cover the self-adaptation of systems.

We now summarize the most important challenges of each the views identified.

Modelling Dimensions. A major challenge in modelling dimensions is defining 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 analysis and decision process. However, at the same time, models should be sufficiently general and simple to keep synthesis feasible. Defining utility functions for supporting decision making is a challenging task, and we need practical techniques to specify and generate such utility functions.

Requirements. The major challenge in requirements is defining 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, we need to represent the trade offs between the flexibility provided by the uncertainty and the assurances required by the application. Since requirements might vary at run-time, systems should be aware of their own requirements, creating a need for requirements reflection and online goal refinement. It is important to note that requirements should not be considered in isolation and we must develop techniques for mapping requirements into architecture and implementation.

Engineering. In order to properly engineer self-adaptive software systems, the feedback control loop must become a first-class entity throughout the process. To allow this, there is the need for modelling support to make the loop's role explicit. Explicit modelling of the loops will ease reifying system properties to allow their query and modification at run-time. In order to facilitate reasoning between system properties and the feedback control loop, reference architectures must highlight key aspects of the loop, such as, number, structural arrangements, interactions, and stability conditions. In order to maintain users' trust, certain aspects of the control must be exposed to the users. Finally, in order to facilitate organized use and reuse of self-adaptation mechanisms, the community must compile a catalog of feedback control loops, explicitly explaining their properties, benefits and shortcomings, and interaction possible methods for interaction with other loops.

Assurances. The major challenge in assurances is supplementing traditional methods applied at requirements and design stages of development with run-time assurances. Since system context is dynamic at run-time, systems must identify new contexts dynamically. In order to handle the uncertainty associated with this process, models must include uncertainty via, e.g., probabilistic approaches. Further, adaptation-specific model-driven environments may facilitate modelling support of run-time self-adaptation; however, these environments must be lightweight, in order to allow run-time verification without impacting system performance. One approach to run-time verification of assurances is the labelling of such assurances as "desirable," rather than "required."

There are several aspects related to software engineering of self-adaptive systems that we did not cover. One of them is processes, which are an integral part of software engineering. Software engineering processes are essentially associated with design-time; however, engineering of self-adaptive systems will also require run-time processes for handling change. This may require re-evaluating how software should be developed for self-adaptive systems. For example, instead of a single process, two complementary processes may be required for coordinating the design-time and run-time activities of building software, which might lead to a whole new way of developing software. Technology should enable and influence the development of self-adaptive systems. Other aspects of software engineering related to self-adaptation are technologies like model-driven development, aspect-oriented programming, and software product lines. These technologies might offer new opportunities and offer new processes in the development of self-adaptive systems.

During the course of our work, we have learned that the area of self-adaptive systems is vast and multidisciplinary. Thus, it is important for software engineering to learn and borrow from other fields of knowledge that are working or have being working in the development and study of similar systems, or have already contributed solutions that fit the purpose of self-adaptive systems. We have already mentioned some of the fields, such as control theory and biology, but decision theory, non-classic computation, and computer networks may also prove to be useful. Finding a solution in one of these fields that fits our needs exactly is unlikely; however, studying a wide range of exemplars is likely to provide necessary knowledge of benchmarks, methods, techniques, and tools to solve the challenges of engineering self-adaptive software systems.

The four theses we have discussed in this paper outline new challenges that our community must face in engineering self-adapting software systems. These challenges all result from the dynamic nature of adaptation. This dynamic nature brings uncertainty that some traditional software engineering principles and techniques are the proper way to go about designing self-adaptive systems and will likely require novel solutions.

References

1. Feiler, P., Gabriel, R.P., Goodenough, J., Linger, R., Longstaff, T., Kazman, R., Klein, M., Northrop, L., Schmidt, D., Sullivan, K., Wallnau, K.: Ultra-large-scale systems: The software challenge of the future. Technical report, Software Engineering Institute (2006), `http://www.sei.cmu.edu/uls/`

2. Andersson, J., de Lemos, R., Malek, S., Weyns, D.: Towards a classification of self-adaptive software system. In: Cheng, B.H.C., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. LNCS, vol. 5525. Springer, Heidelberg (2009)
3. Seetharaman, G., Lakhotia, A., Blasch, E.P.: Unmanned Vehicles Come of Age: The DARPA Grand Challenge. Computer 39, 26–29 (2006)
4. Hirschfeld, R., Costanza, P., Nierstrasz, O.: Context-oriented programming. Journal of Object Technology 7, 125–151 (2008)
5. Jackson, M.: The meaning of requirements. Annals of Software Engineering 3, 5–21 (1997)
6. Laprie, J.C.: From dependability to resilience. In: International Conference on Dependable Systems and Networks (DSN 2008), Anchorage, AK, USA, pp. G8–G9 (2008)
7. Cheng, B.H.C., Atlee, J.M.: Research directions in requirements engineering. In: FOSE 2007: 2007 Future of Software Engineering, pp. 285–303. IEEE Computer Society, Minneapolis (2007)
8. Fickas, S., Feather, M.S.: Requirements monitoring in dynamic environments. In: IEEE International Symposium on Requirements Engineering (RE 1995), pp. 140–147 (1995)
9. Savor, T., Seviora, R.: An approach to automatic detection of software failures in realtime systems. In: IEEE Real-Time Technology and Applications Symposium, pp. 136–147 (1997)
10. Sutcliffe, A., Fickas, S., Sohlberg, M.M.: PC-RE a method for personal and context requirements engineering with some experience. Requirements Engineering Journal 11, 1–17 (2006)
11. Liaskos, S., Lapouchnian, A., Wang, Y., Yu, Y., Easterbrook, S.: Configuring common personal software: a requirements-driven approach. In: 13th IEEE International Conference on Requirements Engineering (RE 2005), pp. 9–18. IEEE Computer Society, Los Alamitos (2005)
12. Lapouchnian, A., Yu, Y., Liaskos, S., Mylopoulos, J.: Requirements-driven design of autonomic application software. In: CASCON 2006: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research, p. 7. ACM, New York (2006)
13. Goldsby, H.J., Sawyer, P., Bencomo, N., Hughes, D., Cheng, B.H.C.: Goal-based modeling of dynamically adaptive system requirements. In: 15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS) (2008)
14. Dardenne, A., van Lamsweerde, A., Fickas, S.: Goal directed requirements acquisition. In: Selected Papers of the Sixth International Workshop on Software Specification and Design (IWSSD), pp. 3–50 (1993)
15. Yu, E.S.K.: Towards modeling and reasoning support for early-phase requirements engineering. In: 3rd IEEE International Symposium on Requirements Engineering (RE 1997), Washington, DC, USA, p. 226 (1997)
16. Harel, D., Marelly, R.: Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine. Springer, Heidelberg (2005)
17. Zhang, J., Cheng, B.H.C.: Using temporal logic to specify adaptive program semantics. Journal of Systems and Software (JSS), Architecting Dependable Systems 79, 1361–1369 (2006)
18. Easterbrook, S., Chechik, M.: A framework for multi-valued reasoning over inconsistent viewpoints. In: Proceedings of International Conference on Software Engineering (ICSE 2001), pp. 411–420 (2001)

19. Sabetzadeh, M., Easterbrook, S.: View merging in the presence of incompleteness and inconsistency. Requirements Engineering Journal 11, 174–193 (2006)
20. Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Software: Practice and Experience 35, 705–754 (2005)
21. Whittle, J., Sawyer, P., Bencomo, N., Cheng, B.: A language for self-adaptive system requirement. In: SOCCER Workshop (2008)
22. Finkelstein, A.: Requirements reflection. Dagstuhl Presentation (2008)
23. Zhang, J., Cheng, B.H.C.: Model-based development of dynamically adaptive software. In: Proceedings of International Conference on Software Engineering (ICSE 2006), Shanghai,China (2006)
24. Robinson, W.N.: Monitoring web service requirements. In: Proceedings of International Requirements Engineering Conference (RE 2003), pp. 65–74 (2003)
25. Kramer, J., Magee, J.: Self-managed systems: an architectural challenge. In: FOSE 2007: 2007 Future of Software Engineering, Minneapolis, MN, USA, pp. 259–268. IEEE Computer Society, Los Alamitos (2007)
26. Maes, P.: Computional reflection. PhD thesis, Vrije Universiteit (1987)
27. Kon, F., Costa, F., Blair, G., Campbell, R.H.: The case for reflective middleware. Communications of the ACM 45, 33–38 (2002)
28. Coulson, G., Blair, G., Grace, P., Joolia, A., Lee, K., Ueyama, J.: A generic component model for building systems software. ACM Transactions on Computer Systems (2008)
29. Robinson, W.: A requirements monitoring framework for enterprise systems. Requirements Engineering 11, 17–24 (2006)
30. Tanner, J.A.: Feedback control in living prototypes: A new vista in control engineering. Medical and Biological Engineering and Computing 1(3), 333–351 (1963), `http://www.springerlink.com/content/rh7wx0675k5mx544/`
31. Dumont, G., Huzmezan, M.: Concepts, methods and techniques in adaptive control. In: Proceedings American Control Conference (ACC 2002), Anchorage, AK, USA, vol. 2, pp. 1137–1150 (2002)
32. Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litiou, M., Müller, H., Pezzè, M., Shaw, M.: Engineering self-adaptive systems through feedback loops. In: Cheng, B.H., de Lemos, R., Giese, H., Inverardi, P., Magee, J. (eds.) Software Engineering for Self-Adaptive Systems. Lecture Notes in Computer Science Hot Topics, vol. 5525 (2009)
33. 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 Autonomous Adaptive Systems (TAAS) 1(2), 223–259 (2006)
34. Burns, R.: Advanced Control Engineering. Butterworth-Heinemann (2001)
35. Dorf, R.C., Bishop, R.H.: Modern Control Systems, 10th edn. Prentice-Hall, Englewood Cliffs (2005)
36. Perrow, C.: Normal Accidents: Living with High-Risk Technologies. Princeton University Press, Princeton (1999)
37. Söderström, T., Stoica, P.: System Identification. Prentice-Hall, Englewood Cliffs (1988)
38. Schloss Dagstuhl Seminar 08031 Wadern, Germany: Software Engineering for Self-Adaptive Systems (2008), `http://www.dagstuhl.de/08031/`
39. Liu, Y., Cukic, B., Fuller, E., Yerramalla, S., Gururajan, S.: Monitoring techniques for an online neuro-adaptive controller. Journal of Systems and Software (JSS) 79(11), 1527–1540 (2006)

40. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool support for the design of self-optimizing mechatronic multi-agent systems. International Journal on Software Tools for Technology Transfer (STTT) 10 (2008) (to appear)
41. Weyns, D.: An architecture-centric approach for software engineering with situated multiagent systems. PhD thesis, Department of Computer Science, K.U. Leuven, Leuven, Belgium (2006)
42. Garlan, D., Cheng, S.W., Schmerl, B.: Increasing system dependability through architecture-based self-repair. In: de Lemos, R., Gacek, C., Romanovsky, A. (eds.) Architecting Dependable Systems. LNCS, vol. 2677. Springer, Heidelberg (2003)
43. Müller, H.A., Pezzè, M., Shaw, M.: Visibility of control in adaptive systems. In: Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), ICSE 2008 Workshop (2008)
44. 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 (2007)
45. Brun, Y., Medvidovic, N.: Fault and adversary tolerance as an emergent property of distributed systems' software architectures. In: Proceedings of the 2nd International Workshop on Engineering Fault Tolerant Systems (EFTS 2007), Dubrovnik, Croatia, pp. 38–43 (2007)
46. Di Marzo Serugendo, G., Fitzgerald, J., Romanovsky, A., Guelfi, N.: Metaself - a framework for designing and controlling self-adaptive and self-organising systems. Technical Report BBKCS-08-08, School of Computer Science and Information Systems, Birkbeck College, London, UK (2008)
47. Cheng, S.W., Garlan, D., Schmerl, B.: Making self-adaptation an engineering reality. In: Babaoğlu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A., van Steen, M. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 158–173. Springer, Heidelberg (2005)
48. Shaw, M.: Beyond objects. ACM SIGSOFT Software Engineering Notes (SEN) 20(1), 27–38 (1995)
49. Babaoglu, Ö., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A.P.A.: The self-star vision. In: Babaoglu, O., Jelasity, M., Montresor, A., Fetzer, C., Leonardi, S., van Moorsel, A. (eds.) SELF-STAR 2004. LNCS, vol. 3460, pp. 1–20. Springer, Heidelberg (2005)
50. Inverardi, P., Tivoli, M.: The future of software: Adaptation and dependability. In: ISSSE 2008, pp. 1–31 (2008)
51. Sama, M., Rosenblum, D., Wang, Z., Elbaum, S.: Model-based fault detection in context-aware adaptive applications. In: International Symposium on Foundations of Software Engineering (2008)
52. Liu, Y., Cukic, B., Gururajan, S.: Validating neural network-based online adaptive systems: A case study. Software Quality Journal 15(3), 309–326 (2007)
53. Hageman, J.J., Smith, M.S., Stachowiak, S.: Integration of online parameter identification and neural network for in-flight adaptive control. Technical Report NASA/TM-2003-212028, NASA (2003)
54. Kaner, C.: Software liability. Software QA 4 (1997)