Software Engineering for Self-Adaptive Systems: A Research Road Map (Draft Version)*

Betty H.C. Cheng, Rogério de Lemos, Holger Giese, Paola Inverardi, 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 Kienle, Jeff Kramer, Marin Litoiu, Sam Malek, Raffaela Mirandola, Hausi Müller, Sooyong Park, Mary Shaw, Matthias Tichy, Massimo Tivoli, Danny Weyns, Jon Whittle
(Dagstuhl Seminar Participant Authors)
Contact Emails: r.delemos@kent.ac.uk, holger.giese@hpi.uni-potsdam.de

ABSTRACT

Software's ability to adapt at run-time to changing user needs, system intrusions or faults, changing operational environment, and resource variability has been proposed as a means to cope with the complexity of today's software-intensive systems. Such self-adaptive systems can configure and reconfigure themselves, augment their functionality, continually optimize themselves, protect themselves, and recover themselves, while keeping most of their complexity hidden from the user and administrator. In this paper, we present research road map for software engineering of self-adaptive systems focusing on four views, which we identify as essential: requirements, modelling, engineering, and assurances.

Keywords

Software engineering, requirements engineering, modelling, evolution, assurances, self-adaptability, self-organization, self-management

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 requires new and innovative approaches for building, running and managing software systems [18]. A consequence of this continuous evolution is that software systems must become more versatile, flexible, resilient, dependable, robust, energy-efficient, recoverable, customizable, configurable, or self-optimizing by adapting to changing operational contexts and environments. The complexity of current software-based systems has led the software engineering community to look for inspiration in diverse related fields (e.g., robotics, artificial intelligence) as well as other areas (e.g., biology) to find new ways of designing and managing systems and services. In this endeavour, the capability of the system to adjust its behaviour in response to its perception of the environment and the system itself in form of self-adaptation has become one of the most promising directions.

The topic of self-adaptive systems has been studied within the different research areas of software engineering, including, requirements engineering, software architectures, middleware, component-based development, and programming languages, however most of these initiatives have been isolated and until recently without a formal forum for discussing its diverse facets. Other research communities that have also investigated this topic from their own perspective are even more diverse: fault-tolerant computing, distributed systems, biologically inspired computing, distributed artificial intelligence, integrated management, robotics, knowledge-based systems, machine learning, control theory, etc. In addition, research in several application areas and technologies has grown in importance, for example, 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 is important to emphasise that in all the above initiatives the common element that enables the provision of self-adaptability is software because of its flexible nature. 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 foun-

*This road map paper is a result of the Dagstuhl Seminar 08031 on "Software Engineering for Self-Adaptive Systems" in January 2008.

Dagstuhl Seminar Proceedings 08031
Software Engineering for Self-Adaptive Systems
http://drops.dagstuhl.de/opus/volltexte/2008/1500

dations 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 development of self-adaptive systems can be viewed from two perspectives, either top-down when considering an individual system, or bottom-up when considering cooperative systems. Top-down self-adaptive systems assess their own behaviour and change it when the assessment indicates a need to adapt due to evolving functional or non-functional requirements. Such systems typically operate with an explicit internal representation of themselves and their global goals. In contrast, bottom-up self-adaptive systems (self-organizing systems) are composed of a large number of components that interact locally according to simple rules. The global behaviour of the system emerges from these local interactions.[1] The global behaviour of the system emerges from these local interactions, and it is difficult to deduce properties of the global system by studying only the local properties of its parts. Such systems do not necessarily use internal representations of global properties or goals; they are often inspired by biological or sociological phenomena. The two cases of self-adaptive behaviour in the form of individual and cooperative self-adaptation are two extreme poles. In practice, the line between both is rather blurred, and compromises will often lead to an engineering approach incorporating representatives from these two extreme poles. For example, ultra large-scale systems need both top-down self-adaptive and bottom-up self-adaptive characteristics (e.g., the Web is basically decentralized as a global system but local sub-webs are highly centralized). However, from the perspective of software development the major challenge is how to accommodate in a systematic engineering approach traditional top-down approaches with bottom-up approaches.

The goal of this road map paper is to summarize and point out the current state-of-the-art, its limitations, and identify critical challenges for the software engineering of self-adaptive systems. Specifically, we intend to focus on development methods, techniques, and tools that seem to be 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 road map for research and identify the main research challenges for the systematic software engineering of self-adaptive systems.

To present and motivate these challenges, the paper is structured using the four views which have been identified as essential. Each of these views are roughly presented in terms of the state of the art and the challenges ahead. We first review the state of the art and needs concerning requirements (Section 2). Then, the relevant modelling dimensions are discussed in Section 3 before we discuss the engineering of self-adaptive systems in Section 4. The considerations are completed by looking into the current achievements and needs for assurance in the context of self-adaptive systems in Section 5. Finally, the findings are summarized in Section 6 in terms of lessons learned and future challenges.

[1]In context of biologically inspired systems usually self-organization rather than self-adaptation is used and similar to our initial characterization we distinguish "strong self-organizing systems," which are those systems where there is no explicit central control either internal or external (bottom up); from "weak self-organizing systems," which are those systems where, from an internal point of view, there is re-organization maybe under an internal (central) control or planning (top-down). Strong self-organizing systems are thus purely decentralized, access to global information is limited or impossible, interactions occur locally (among neighbours) and based on local information [13].

2. 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 a less than optimal pattern in the environment? Presumably, the system still needs to maintain a set of high-level goals that should be maintained 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 ought to 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 carried out. 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 runtime and which must always be maintained? In short, requirements engineering for self-adaptive systems must deal with uncertainty because the expectations on the environment frequently vary over time.

2.1 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 [19, 42]. They also explain how preliminary work on personalized and customized software can be applied to adaptive systems (e.g., [47, 31]). In addition, some research approaches have successfully used goal models as a foundation for specifying the autonomic behaviour [29] and requirements of adaptive systems [22].

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 runtime

2.2 Research Challenges

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 [11] and i* [51], there is no explicit support for uncertainty or adaptivity. Scenario-based notations generally do not provide any support either although live sequence charts (LSCs) [24] have a notion of mandatory versus potential behaviour which could possibly be used in specifying adaptive systems. Of course, the most common notation for specifying requirements in industry is still using 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 logics and perhaps new adaptation-oriented logics [53] need to be developed to specify the semantics for what it means to have the "possibility" of conditions [17, 40]. There is also a relationship with variability management mechanisms in software product lines [48], 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.

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 runtime adaptation. There are a variety of technical options 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 these 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! Allowing uncertainty levels when developing self-adaptive systems requires a tradeoff between flexibility and assurance such that the critical high-level goals of the application are always met [52, 39, 28].

Requirements reflection. As said above, self-adaptation deals with requirements that vary at runtime. Therefore it is important that requirements lend themselves to be dynamically observed, i.e., during execution. Reflection [34], [27], [10] enables a system to observe its own structure and behaviour. A relevant research work is the ReqMon tools [38] which provides a requirements monitoring framework, focusing on temporal properties to be maintained. Leveraging and extending beyond these complementary approaches, Finkelstein [20] coins the term "requirements reflection" that would enable systems to be aware of their own requirements at runtime. This 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 runtime objects? Future work is needed to examine how technologies may provide the infrastructure to do this.

Online goal refinement. As in the case of design decisions that are eventually realized at runtime, new and more flexible requirement specifications like the one suggested above would imply that the system should perform the RE processes at runtime, e.g. goal-refinement [28].

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 runtime 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.

2.3 Final Remarks

In this section, we have presented several important research challenges that the requirements engineering community will face as the demand for self-adaptive systems continues to grow. These challenges span RE activities during the development phases and runtime. In order to gain

assurance about adaptive behaviour, it is important to monitor adherence and traceability to the requirements during runtime. Furthermore, it is also necessary to acknowledge and support the evolution of requirements at runtime. Given the increasing complexity of applications requiring runtime adaptation, the software artifacts 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 artifact that is manipulated once a system has been deployed? How can we maintain traceability among relevant artifacts, 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 runtime 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.

3. MODELLING

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.

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. Some of these dimensions could equally be applied to the environment and the users of the system (in addition to other specific dimensions), but here we have focused on the system itself.

For the identification of the system modelling dimensions, two perspectives were considered: the abstraction levels associated with the system, and the activities associated with the adaptation. The first perspective refers to the requirements (e.g., goals), the design (e.g., architecture), and the code of the software system, and the second refers to the key activities of the feedback control loop, i.e., collect, analyse, decide, and act.

In the following, we present the dimensions in term of three groups. First, we introduce the modelling dimensions that can be associated with the adaptation activities of the feedback control loop, giving special emphasis to decision making. The other two groups are related to non-functional properties, i.e., timing and dependability, that are particularly relevant to some classes of self-adaptive systems. The proposed modelling framework is presented in the context of an illustrative case from the class of embedded systems, however, these dimensions were equally useful in describing the self-adaptation properties of an IT change management system.

3.1 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 [44]. 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-adaptable 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.

3.2 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.

Adaptation

The first group describes the modelling dimensions related to adaptation.

Type of adaptability. The type of adaptability refers to the particular kind of adaptation applied. The domain of type of adaptability ranges from parametric to compositional. Self-adaptivity can be realized by simple local parametric changes of a system component, for example, or it can involve major architectural level structural changes. 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.

Degree of automation. The automation dimension refers to the degree of human intervention required for self-adaptation. The domain of degree of automation ranges from autonomous to human-based. Adaptive systems may be fully automatic requiring no human intervention, or the system may require human decision making, or at least confirmation or approval. In the illustrative example, the UV has to avoid collisions with animals without any human intervention.

Form of organization. The form of organization refers to the type of organization used to realize self-adaptation. The domain of form of organization ranges from weak (or centralized) to strong (or decentralized). In a strong organization, the behaviour of components reflect their local environment, there is no global model of the system. Driven by changing requirements, the components change their structure or behaviour to self-adapt the system. This self-organizing form of self-adaptation can be collaborative, market-based, and so on. In a weak organization, adaptation is achieved through a global system model, which incorporates a feedback control loop, for example. A self-adaptive subsystem monitors the base system possibly maintaining an explicit representation of the system, and based on a set of high-level goals, the structure or behaviour of the system is adapted. Section 4 elaborates on the different forms of organization to realize self-adaptation. The SCS of the UV in the illustrative example seems to fit naturally with a weak organization.

Techniques for adaptability. Techniques for adaptability refer to the way self-adaptation is accomplished. The domain of techniques for adaptability ranges from data-oriented to process-oriented [46]. In a data-oriented approach, the system is characterised as acted upon, by providing the criteria for identifying objects, often by modelling the objects themselves. In a process-oriented approach, the system is characterised as sensed, by providing the means for producing or generating objects having the desired characteristics. In the illustrative case, the SCS will monitor the environment for obstacles that suddenly appear in front of the vehicle and subsequently guide the vehicle around the obstacle to avoid a collision. To realize this form of self-adaptability, the SCS senses the environment of the UV, and depending on the controller, which is part of the system model, it produces the appropriate system output.

Place of change. The place of change refers to the location where self-adaptation takes place. The domain of place of change includes the values application, middleware, or infrastructure. Self-adaptation can be realized by monitoring and adapting the application logic, the supporting middleware, or the infrastructure that defines the system. In the illustrative case, self-adaptation is realized by the SCS that is part of the application logic.

Abstraction of adaptability. This modelling dimension refers to abstraction level at which self-adaptation is applied. The domain of abstraction of adaptability refers to requirements, design, and implementation, and their respective products, for example, goals, architectures and code. An example of adaptation at the design level is the dynamic reconfiguration of the system architecture. Another example of adaptation at the design level can be the selection of an alternative algorithm. An example of adaptation at the level of code is dynamic weaving of additional code. To avoid collisions, the SCS may pass particular control information to the ACS which seems to fit best at the abstraction level of design.

Impact of adaptability. This modelling dimension refers to the impact that adaptation might have upon the system. The domain of impact of adaptability ranges from specific to generic. Adaptability is specific if it affects a particular component or part of the system. On the other hand, if the adaptability affects the whole system, its impact is generic. In the illustrative case, if the the steering gear fails, the self-adaptation would be generic since collision avoidance affects the overall system's behaviour.

Trigger of adaptability. This modelling dimension refers whether the agent of change is either internal or external to the system. A failure in a system component is considered as an internal trigger for reconfiguring the system structure or changes the services it provides, while the existence of an obstacle is an external trigger since the system has to change its behaviour in order to avoid a collision.

In addition to the above modelling dimensions that can be applied to the system as a whole, there are some dimensions related specifically to the key activities of the feedback control loop. In the following, we present two of that modelling dimensions that are related to decision making.

Degree of decision making. The degree of decision making expresses to what extent self-adaptation is defined in advance. The domain ranges from static (or pre-defined) to dynamic (or run-time). For static decision making, the scenarios of self-adaptation are exhaustively defined before the system is deployed. For dynamic decision making, the decision of self-adaptation will be made during execution based on a set of high-level goals. In the illustrative example, the SCS monitors the environment and decides at run-time when it has to take control over the ACS to avoid collisions.

Techniques for decision making. This modelling dimension refers to the procedures and methods used to determine when to apply self-adaptation. Values of the domain of techniques for decision making are utility functions, case-based reasoning, etc. The SCS will likely use a reasoning-like approach to determine when the vehicle is in collision range with an obstacle.

Timing

The second group describes modelling dimensions related to timing issues.

Responsiveness. The responsiveness of self-adaptation relates to the answering or replying of the self-adaptation. The domain ranges from guaranteed to best-effort. For critical scenarios, self-adaptation is required to be guaranteed, however, in less-critical situations, best-effort will suffice. In the illustrative example, the SCS must guarantee that the UV reacts effectively to avoid collisions with possibly a human being.

Performance. The performance dimension refers to the degree of predictability of self-adaptation. The domain ranges from predictable to degradable. In time-critical cases, the self-adaptable system often needs to act in a highly predictable manner. In other cases, a graceful degradation of the system is acceptable. In the illustrative case, when an obstacle appears, the SCS will manoeuvre the UV in such a way that a collision should be avoided. In order to accomplish this task predictably, other system tasks might have their performance affected.

Triggering. The triggering dimension of self-adaptation refers to the initiation of the adaptation process. The domain of triggering ranges from event to time. The cause for self-adaptation is event triggered when the process is initiated whenever there is a significant change in the state, i.e., an event. The cause for self-adaptation is time triggered when the process is initiated at predetermined points in time. Obstacles in the illustrative case appear unexpectedly and as such triggering of self-adaptation is event-based.

Dependability

The third and final group we consider describes modelling dimensions related to dependability, that is, the ability of a system to deliver a service that can justifiably be trusted [1].

Reliability, availability, confidentiality. Reliability, availability, and confidentiality are attributes of dependability. The domain of each of these properties ranges from high to low. In the illustrative case, the reliability of the SCS avoiding a collision is expected to be high.

Safety. The safety dimension refers to absence of catastrophic consequences on the user and the environment, which can be caused by the self-adaptation. The domain of safety ranges

from critical to non-critical. Self-adaptation in the illustrative example is clearly critical because of the catastrophic if there is a failure.

Maintainability. The maintainability modelling dimension refers ability to undergo modifications and repairs. The domain of maintainability ranges from autonomous to human-based. An fully autonomous self-adaptive system includes facilities to self-check and self-tune its abilities to adapt the system. Given the nature of the Grand Challenge contest is likely that the system should be autonomous as far as maintainability is concerned.

Data integrity. The data integrity dimension of self-adaptation refers to the improper alterations of the data. The domain of this modelling dimension ranges from short-term to long-term. Short-term data integrity applies to hard real-time systems in which the flow of time tends to invalidate the date, while long-term data integrity applies to transaction processing systems. Since the illustrative example is related to an embedded real-time system, the data integrity will be short-term.

3.3 Challenges Ahead

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 three presented groups of modelling dimensions.

Adaptation

Many types of adaptation techniques have been developed: architecture-based adaptation that is mainly concerned with structural changes at the level of software components, parametric-based adaptation that leverages policy files or input parameters to configure the software components, aspect-oriented-based adaptation that changes the source code of a running system via dynamic source-code weaving, and so on. 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.

There is a wide spectrum of the degree of automation supported by the current state of the art approaches. However, in general, there are significant hurdles facing the construction of fully automatic adaptive systems, many of which are reminiscent of the challenges that the AI community has faced over the past few decades. It is imperative for the software engineering community to develop better models that incorporate the AI techniques in solving the practical problems of automatic adaptive systems.

Most state of the art adaptive systems are built according to the centralized closed-control loop pattern. Thereby, if applied to a large-scale software system, almost all current techniques suffer from scalability problems. The field of multi-agent systems has developed a large body of knowledge on systems in which the system components adapt their structure or behaviour to changing requirements to realize system adaptation. Related are biologically inspired adaptation systems that tend to be more decentralized as well. However, practical experiences with these approaches in real-world settings is limited. Methods used in systems engineering, like hierarchical organization and coordination schemes could be applicable. 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.

Most state of the art techniques leverage a utility function to map the trade-offs among several conflicting goals of adaptability to a scalar value, which is then used for making decisions about adaptation. However, in practice, defining such a utility function is a challenging task. 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 *ceteris paribus* conditions.

Dynamic/run-time decision requires efficient mechanisms for gathering information about the running system and its environment. Principled approaches for efficient gathering of information at run-time are needed. An important research effort will be to understand how we can collect the necessary information for different domains and derive reusable engineering approaches.

Timing

Responsiveness is a crucial property in real-time software systems, which are becoming more prevalent with the emergence of embedded and cyber-physical systems. These systems are often required to respond deterministically within pre-specified (often short) time intervals, making it extremely difficult to adapt the system, while satisfying timing constraints. There is a need for adaptation models targeted for real-time systems that treat the duration and overhead of adaptation as first class entities.

Predicting the exact behaviour of a software system due to run-time changes is a challenging task. For example, while it may be possible to predict the new functionality that will become available as a result of replacing a software component, it may not be possible to determine what will be the impact of the replaced software component on the other components that are sharing the same resources (e.g., CPU, memory, and network). More advanced and predictive models of adaptation are needed for systems that could fail to satisfy their requirements due to side-effects of change.

Often, adaptation is triggered by the occurrence of a pattern in the data that is gathered from a running system. For example, the system is monitored to determine when a particular level of Quality of Service (QoS) is not satisfied, which then initiates the adaptation process. However, 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. We believe that more research on light-weight monitoring techniques, as well as more advanced models that take the monitoring overhead of the approach into account are needed.

Dependability

Adapting safety-critical software systems while ensuring the safety requirements have remained largely an out-of-reach goal for the practitioners and researchers. There is a need for verification and validation techniques that guarantee safe and sound adaptation of safety-critical systems, under all foreseen and unforeseen, see also section 5.

4. ENGINEERING

Building self-adaptive software systems cost-effectively and in a predictable manner is a major engineering challenge even though adaptive systems have a long history with huge successes in many different branches of engineering [49, 16]. Mining the rich experiences in these fields, borrowing theories from control engineering, and then applying the findings to software-intensive adaptive systems is a most worthwhile and promising avenue of research. Lehman's work on software evolution [30] has shown that "[t]he software process constitutes a multilevel, multiloop feedback system and must be treated as such if major progress in its planning, control, and improvement is to be achieved." Therefore, any attempt to automate parts of these processes such as self-adaptive systems necessarily also has to consider feedback loops. Therefore, we advocate to focus on the feedback loop—a concept that is elevated to a first-class entity in control engineering—when engineering self-adaptive software systems.

4.1 State of the Art & Feedback Loops

Self-adaptation in software-intensive systems comes in many different guises. What self-adaptive systems have in common is that design decisions are moved towards runtime and that the system reasons about its state and environment. The reasoning typically involves feedback processes with four key activities: collect, analyze, decide, and act as depicted in Figure 1 [14]. Here we concentrate on self-adaptive systems that are implemented using feedback mechanisms to control their dynamic behavior. For example, keeping web services up and running for a long time requires collecting of information that reflects the current state of the system, analyzing of 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 to effect the made decision.

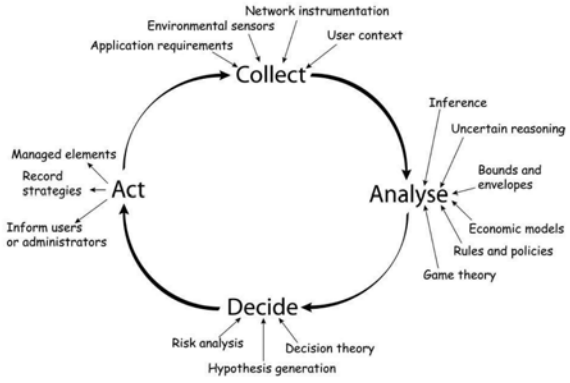

Figure 1: Activities of the control loop.

We have observed that feedback loops are often hidden, abstracted, or internalized when presenting the architecture of self-adaptive systems [36]. However, the feedback behavior 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 making the control loops explicit, the control loops' properties have to be made explicit as well. Cheng, Garlan and Schmerl also advocate to make self-adaptation external, as opposed to be internal or hard-wired, to separate the concerns of system functionality from the concerns of self-adaptation [8].

Generic Control Loop Model

The generic model of a control loop presented in Figure 1 provides a good overview of the main activities around the feedback loop, but ignores many properties of the control and data flow around the loop. When engineering a self-adaptive system, questions about these properties become important. The feedback cycle starts with the *collection* of relevant data from environmental sensors and other sources that reflect the current state of the system (see top of Figure 1). Some of the engineering questions that need be answered here 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 applicable 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? Next, a *decision* must be made about how to adapt the system in order to reach a desirable state. Approaches such as risk analysis can help to make a decision among various alternatives. 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 and can the adaptation be safely performed? How do adjustments of different feedback loops interfere with each other? Do centralized or decentralized control help achieve the global goal? The above questions — and many others — regarding the control loop should be explicitly identified, recorded, and resolved during the development of the self-adaptive system.

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 feedback loops—and their associated properties—that almost inevitably control the self-adaptations. 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 [45]. She introduced a new software organization paradigm based on control loops with an architecture that is dominated by feedback loops and their analysis rather than by the identification of discrete stateful objects. As pointed out in

[26] different forms of control loops may be employed for self-adaptive software and we may even go beyond adaptive control and use reconfigurable control where besides the parameters also structural changes are considered (cf. compositional adaptation [35]).

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 [6, 15]. Researchers have applied results of control theory and engineering when building self-adaptive systems. However, it is not clear if general principles of this discipline (e.g., open/closed-loop controller, observability, controlability, stability, or hysteresis) are applicable when reasoning about self-adaptive software systems. Systems with a single control loop are easier to reason about than systems with multiple loops-which are much more common. Good engineering practice calls for reducing multiple control loops to a single one, or making control loops independent of each other [37]. If this is not possible, the design must make the interactions of control loops explicit and expose how these interactions are handled. Another typical scheme from control engineering is organizing multiple control loops in the form of a hierarchy where, due to the employed different time periods, unexpected interference between the levels can be excluded. This scheme seems to be of particular interest if we distinguish different forms of adaptation such as change management and goal management [28]. There is a rich history of control theory in all branches of engineering. Mining the experiences in these fields and applying them to software-intensive adaptive systems is a most worthwhile next step.

Specific Control Loop Models

Another key observation that we made is that different application areas introduce different nomenclature and architectural diagrams for their realization of the generic feedback loop depicted in Figure 1. It is useful to investigate how different application areas realize this generic feedback loop and to point out the commonalities in order to compare and leverage self-adaptive systems research from different application areas.For example, control engineering leverages the Model Reference Adaptive Control (MRAC) solution to describe many kinds of feedback-based systems (e.g., flight control) [16]. Because of the separation of concerns (i.e., model reference, adaptive algorithm, controller and process), this solution is a solid starting point for the design of self-adaptive software-intensive systems. In fact, many participants of the Seminar [43] presented self-adaptive systems that could be expressed in terms of MRAC (e.g., self-adaptive flight control [33], autonomous shuttles [5], AGV transportation [50], and the Rainbow system [21]). One of the most recognized derivatives of the MRAC solution for the computing domain is the autonomic element and its architectural blueprint developed by IBM for autonomic, self-managed systems [9]. The IBM architectural blueprint identifies the autonomic element as a fundamental building block for designing self-configuring, self-healing, self-protecting and self-optimizing systems.

In contrast to engineered self-adaptive systems, biologically inspired systems do not often have clearly visible control loops. Further, 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 behavior that yields the global goal as an emergent property. An example of a self-organizing biologically inspired software system is a distributed computational system built using the tile architectural style from [3]. In such a system, components distributed around the Internet come together to "self-assemble" into a solution to an NP-complete problem. This system can self-adapt to exhibit properties of fault and adversary tolerance [4]. The self-adaptation control-loop is not evident, but it does exist, and it resembles that of Figure 1. In an attempt to unify the self-adaptive (top-down) and self-organising (bottom-up) views, [12] propose a software architecture based on the use of metadata and policies where adaptation properties and feedback loop reasoning are considered explicitly both at design-time and run-time.

4.2 Challenges Ahead

We have argued that the control loop should be a first-class entity when thinking about the engineering of self-adaptive systems. We believe that understanding and reasoning about the control loop is key for advancing the construction of self-adaptive systems from an ad-hoc, trial-and-error endeavor towards a more disciplined approach. To achieve this goal the following issues, possibly among others, have to be addressed.

Modelling. There should be modelling support to make the control loop explicit and to expose self-adaptive properties so that the designer can reason about the system. The models have to capture what can be observed and what can be influenced. It would be desirable to have a widely agreed upon reference model of self-adaptive systems including the control loop. Another challenge is characterizing the control loop for self-organizing systems that are biologically inspired such as swarms. For such systems the control loop seems implicitly present.

The nature of self-adaptive system require to reify properties that would otherwise be encoded implicitly. These reified properties need to be modelled appropriately so that they can be queried and modified during run-time. Examples of such properties are system state that is used to reason about the system's behavior, and policies and business goals that govern and constrain how the system will and can adapt.

Architecture. Reference architectures for adaptive systems should highlight key aspects of feedback loops, including number of control loops, structural arrangements of control loops (e.g., sequence, 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. It is highly desirable that such architectures can be used to reason about the properties of the system and its control loop. In other words, is it possible to come up with Attribute-Based Architectural Styles (ABASs) for control loops in self-adaptive systems? Control engineering and existing approaches to cope with control loops assume a static control loop at the instance level. While this assumption eases the modelling and architecting of self-adaptive systems, it is by no means sufficient for software systems where dynamic structures are the default and

not the exception. Therefore, it can be expected that also control loops which can only be studied at the type level or are established at run-time become a major issue.

Design. For (multiple) control loops there should be a design catalog of common forms along with associated obligations and reasoning. We may have patterns for specific kinds of interacting control loops, for manual vs. automatic control, for making control loops independent, etc. As a first step, we should identify canonical self-adaptive systems from different domains and document their control loops. This should also help us to better understand the differences in control loops for self-adaptive and self-organizing systems and the resulting differences in design goals. For example, a design goal for self-adaptive systems is the independence of control loops. However, independence of control loops is a contradictory requirement for engineered self-organizing systems. In such systems loops overlap, local and global control loops interfere with each other. Individual components obey their own control loop (positive and negative feedback rules); it is the combined local interactions among these components that provide global control loops at the level of the system. It is the combined work of ants (laying down pheromone trails) that leads them to find the best source of food, and to find alternative ones when it is exhausted (changing the pheromone trails).

System-level support. Good system-level support should "allow researchers with different motivations and experiences to put their ideas in practice, free from the painful details of low-level system implementation" [2]. Such middleware support should allow for flexibility and rapid prototyping, supporting different heterogeneous platforms. This could be achieved with a framework and standardized interfaces for self-adaptive functionality. Importantly, can there be a common generic middleware, which can be then instantiated to realize both self-adaptive and self-organizing systems?

Human-computer interaction. Even though self-adaptive systems act autonomously in many respects, they have to keep the user in the loop. Providin.g the user with feedback about the system state is crucial to establish and keep users' trust. To that effect, a self-adaptive system needs to expose aspects of its control loop to the user. For example, if a web server is reconfigured in response to a load change, the human administrator needs (visual) feedback that the performed adaptation has a positive effect. Similarly, in a self-adaptive flight control system, the pilot of a plane that is about to spin out of control due to a damaged part would like to get feedback from the system that it is converging towards a stable state in response to that incident. Also, users should be given the option to disable self-adaptive features and the system should take care not to contradict explicit choices made by users [32]. Furthermore, users might want feedback from the system about the information collected by sensors and how this information is used to adapt the system. In fact, if the collected information is personal data there might be even a legal obligation to do so [41].

5. ASSURANCES

The goal of system assurance is simple. Developers need to provide evidence that the set of stated functional and nonfunctional 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 (V&V) 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 V&V methods implied by the presented framework.

5.1 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 behavioral 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.

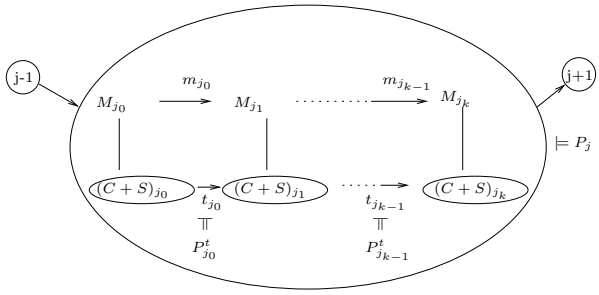

Figure 2: V & V model.

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 run-time. We abstract from the subtle differences between goals and requirements 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 fulfillment. 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 or at least updated 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, although delays in model definition may be inevitable.

5.2 Challenges

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 tradeoffs 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. Models allow the application of V&V 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. 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 runtime 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 runtime 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. Proof-Carrying Code (PCC) is a technique by which a host platform can verify that code provided for execution adheres to a predefined set of safety rules. The limitation of the PCP 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 a proof for code segments which are parameterized and undergo changes is a challenge. It is unclear whether defining such proofs for emerging system properties is even feasible.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 behavior 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. 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 [33, 23]. 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 [25]. 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 disabling 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 defense against liability claims.

6. LESSONS AND CHALLENGES

In this section, we present the overall conclusions of the road map paper in the context of lessons learned and what were the major challenges identified. First and foremost,

this exercise had no intention of being exhaustive. There is a lot of material yet to be covered that barely has been mentioned, or not at all mentioned, in this paper. The choice has been made to focus on four major issues that were identified to be key in the software engineering of self-adaptive systems, and in particular, those that were identified to be essential from the software engineering perspective: requirements, modelling, engineering, and assurances.

In the presentations of each of the four views, the intent was not to cover all the aspects related to them, instead very simple theses were considered as a means for identifying the challenges associated with each of the views. The four identified theses were: requirements - "the need to define a new requirements language for handling uncertainty to give self-adaptive systems the required freedom to do adaptation", modelling - "the need to enumerate and classify modelling dimensions for obtaining precise models to support run-time reasoning and decision making for achieving self-adaptability", engineering - "the need to consider feedback control loops as first-class entities during engineering to rule self-adaptive systems of different flavours appropriately", and assurances - "the need to define novel verification and validation methods for the provision of assurances which cover the adaptation of self-adaptive systems". In the following, for each of the four views, we presented some of the identified challenges.

Requirements. The major challenge here is the definition of a new requirements language that would be able to capture uncertainty at a more abstract level. If uncertainty is consider at the requirements level, means have to be found to manage this uncertainty. Thus there is the need to represent the tradeoffs between the flexibility provided by the uncertainty, and the assurances required by the application. Considering requirements might vary at run-time, systems should be made aware of their own requirements, hence the need of "requirements reflection" and online goal refinement. However, requirements should not be consider in isolation, techniques for mapping requirements into architecture are needed, in addition to new operators that are able support traceability from requirements to implementation.

Modelling. A major challenge here is the definition of models that are able to 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 simple, otherwise synthesis might become unfeasible. The definition of utility functions for supporting decision making is a challenging task, and practical techniques are needed to specify and generate these utility functions. Alternative approaches based on more light weight techniques are needed that might be able to be generated during run-time without human intervention.

Engineering. In order consider the feedback control loop as a first-class entity there is the need for modelling support to make its role more explicit. This should also be desirable for self-organizing systems where loops are not clearly visible. Once loops become more explicit, it becomes much easier to reify properties, so they can be queried and modified at run-time. For facilitating the reasoning between system properties and its control loops, reference architectures should be defined that highlight key aspects of these loops, such as, number, structural arrangements, interactions and stability conditions. From the user perspective it also important to expose certain aspects of the control in order to establish and keep user's trust. Considering the different types of application and system properties, it is expected a wide variety of control loops, hence the need for a design catalog of common forms, which should also be helpful in understanding their differences.

Assurances. The major challenge here is to supplement traditional methods applied at requirements and design stages of development with run time assurances. Since system context may change, there is the need to identify new contexts dynamically. There are uncertainties associated with this process, hence probabilistic approaches is a promising research direction. Models in context of self-adaptability will play a key role. If models support self-adaptation during run-time, it should be possible to perform adaptation analysis as a verification activity. This might be achieved with adaptation-specific model driven environments. However, a key requirement for run-time verification is the existence of efficient agile solution algorithms which do not require high space/time complexity. One approach would be to relax the correctness of the adaptation, and instead use the notion of "desirable" adaptation because we may not know what a correct adaptation is in an unforeseen context.

There are several aspects related to the software engineering of self-adaptive systems that were not covered. One of them are processes, which are an integral part of software engineering. Software engineering processes are essentially associated with design time, however, the engineering of self-adaptive systems will also require run-time processes that would provide means for handling change. This may require re-evaluating how software should be developed for self-adaptive systems. Instead of a single process, two complementary processes may be required for coordinating the design time and run-time activities of building software, and this might lead to a whole new way of developing software. Technology is another aspect that should enable and influence the development of self-adaptive systems. Technologies like, model driven development, aspect-oriented programming, and software product lines might offer new opportunities in the development of self-adaptive systems, and change the processes by which these systems are developed.

One thing that we have learned from this exercise is that the area of self-adaptive systems is vast, it is multidisciplinary, and it involves a wide range of systems. Thus it is important for software engineering to learn from other fields of knowledge that are working, or have being working, in the development of similar systems, or have already contributed with solutions that fit the purpose of self-adaptive systems. Some of the fields have been mentioned in this paper, like, control theory, but other fields from which software engineering might get some inspiration for the development of self-adaptive systems are, decision theory, non-classic computation, and computer networks. Another thing that we have learned from the fact that self-adaptability might be associated with a wide range of systems, is that the idea of finding a solution that should be able to fit all the purposes might be remote. For that reason, exemplars are needed from a wide range of applications that would enable to benchmark the different techniques, methods and tools that will be emerging to solve the different challenges

associated with the engineering of software for self-adaptive systems.

We can conclude that all four theses refer to new challenges the software engineering of self-adaptive systems has to face which result from the dynamics of adaptation. This dynamics requires that well proven principles and techniques valid for standard software engineering have to be questioned and new solutions have to be considered.

7. REFERENCES

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, January-March 2004.

[2] Ö. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, and A. P. A. van Moorsel. The self-star vision. In O. Babaoglu, M. Jelasity, A. Montresor, C. Fetzer, S. Leonardi, and A. van Moorsel, editors, *Proceedings Conference on Self-Star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 1–20, 2005.

[3] Y. Brun and N. Medvidovic. An architectural style for solving computationally intensive problems on large networks. In *Proceedings of Software Engineering for Adaptive and Self-Managing Systems (SEAMS07)*, Minneapolis, MN, USA, May 2007.

[4] Y. Brun and N. Medvidovic. 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 (EFTS07)*, pages 38–43, Dubrovnik, Croatia, September 2007.

[5] S. Burmester, H. Giese, E. Münch, O. Oberschelp, F. Klein, and P. Scheideler. 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).

[6] R. Burns. *Advanced Control Engineering*. Butterworth-Heinemann, 2001.

[7] B. H. C. Cheng and J. M. Atlee. Research directions in requirements engineering. In *FOSE '07: 2007 Future of Software Engineering*, pages 285–303. IEEE Computer Society, Minneapolis, MN, USA, May 2007.

[8] S.-W. Cheng, D. Garlan, and B. Schmerl. Making self-adaptation an engineering reality. In *Proceedings Conference on Self-Star Properties in Complex Information Systems*, volume 3460 of *Lecture Notes in Computer Science*, pages 158–173, 2005.

[9] I. Corporation. An architectural blueprint for autonomic computing. White Paper 4th Ed., IBM Corporation, June 2006. `http://www-03.ibm.com/autonomic/pdfs/AC_BlueprintWhite_Paper_4th.pdf`.

[10] G. Coulson, G. Blair, P. Grace, A. Joolia, K. Lee, and J. Ueyama. A generic component model for building systems software. *ACM Transactions on Computer Systems*, February 2008.

[11] A. Dardenne, A. van Lamsweerde, and S. Fickas. Goal directed requirements acquisition. In *Selected Papers of the Sixth International Workshop on Software Specification and Design (IWSSD)*, pages 3 – 50, 1993.

[12] G. Di Marzo-Serugendo, J. Fitzgerald, A. Romanovsky, and N. Guelfi. A generic framework for the engineering of self-adaptive and self-organising systems. Technical report, School of Computing Science, University of Newcastle, Newcastle, UK, 2007.

[13] G. Di Marzo-Serugendo, M.-P. Gleizes, and A. Karageorgos. Self-organisation in mas. *Knowledge Engineering Review*, 20(2):165–189, 2005.

[14] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Transactions Autonomous Adaptive Systems (TAAS)*, 1(2):223–259, December 2006.

[15] R. C. Dorf and R. H. Bishop. *Modern Control Systems*. Prentice Hall, 10^{th} edition, 2005.

[16] G. Dumont and M. Huzmezan. Concepts, methods and techniques in adaptive control. In *Proceedings American Control Conference (ACC 2002)*, volume 2, pages 1137–1150, Anchorage, AK, USA, 2002.

[17] S. Easterbrook and M. Chechik. A framework for multi-valued reasoning over inconsistent viewpoints. In *Proceedings of International Conference on Software Engineering (ICSE'01)*, pages 411–420, 2001.

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

[19] S. Fickas and M. S. Feather. Requirements monitoring in dynamic environments. In *IEEE International Symposium on Requirements Engineering (RE95)*, pages 140 – 147, 1995.

[20] A. Finkelstein. Requirements reflection. Dagstuhl Presentation, 2008.

[21] D. Garlan, S.-W. Cheng, and B. Schmerl. Increasing system dependability through architecture-based self-repair. In R. D. Lemos, C. Gacek, and A. Romanovsky, editors, *Architecting Dependable Systems*. Springer-Verlag, 2003.

[22] H. J. Goldsby, P. Sawyer, N. Bencomo, D. Hughes, and B. H.C.Cheng. Goal-based modeling of dynamically adaptive system requirements. In *15th Annual IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, 2008.

[23] J. J. Hageman, M. S. Smith, and S. Stachowiak. Integration of online parameter identification and neural network for in-flight adaptive control. Technical Report NASA/TM-2003-212028, NASA, 2003.

[24] D. Harel and R. Marelly. *Come Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer, 2005.

[25] C. Kaner. Software liability. *Software QA*, 4(6), 1997.

[26] M. M. Kokar, K. Baclawski, and Y. A. Eracar. Control theory-based foundations of self-controlling software. *IEEE Intelligent Systems*, 14(3):37–45, 1999.

[27] F. Kon, F. Costa, G. Blair, and R. H. Campbell. The case for reflective middleware. *Communications of the ACM*, 45(6):33–38, 2002.

[28] J. Kramer and J. Magee. Self-managed systems: an architectural challenge. In *FOSE '07: 2007 Future of Software Engineering*, pages 259–268, Minneapolis, MN, USA, May 2007. IEEE Computer Society.

[29] A. Lapouchnian, Y. Yu, S. Liaskos, and J. Mylopoulos. Requirements-driven design of autonomic application software. In *CASCON'06: Proceedings of the 2006 Conference of the Center for Advanced Studies on Collaborative Research*, page 7, New York, NY, USA, 2006. ACM.

[30] M. M. Lehman. Software's future: Managing evolution. *IEEE Software*, 15(1):40–44, 1998.

[31] S. Liaskos, A. Lapouchnian, Y. Wang, Y. Yu, and S. Easterbrook. Configuring common personal software: a requirements-driven approach. In *13th IEEE International Conference on Requirements Engineering (RE05),*, pages 9–18. IEEE Computer Society, 2005.

[32] S. Lightstone. Seven software engineering principles for autonomic computing development. *Innovations in Systems and Software Engineering*, 3(1):71–74, March 2007.

[33] Y. Liu, B. Cukic, E. Fuller, S. Yerramalla, and S. Gururajan. Monitoring techniques for an online neuro-adaptive controller. *Journal of Systems and Software (JSS)*, 79(11):1527–1540, 2006.

[34] P. Maes. *Computional reflection*. PhD thesis, Vrije Universiteit, 1987.

[35] P. K. McKinley, M. Sadjadi, E. P. Kasten, and B. H. Cheng. Composing adaptive software. *IEEE Computer*, 37(7):56–64, July 2004.

[36] H. A. Müller, M. Pezzè, , and M. Shaw. Visibility of control in adaptive systems. In *Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), ICSE 2008 Workshop*, May 2008.

[37] C. Perrow. *Normal Accidents: Living with High-Risk Technologies*. Princeton University Press, 1999.

[38] W. Robinson. A requirements monitoring framework for enterprise systems. *Requirements engineering*, 11:17–4, 2006.

[39] W. N. Robinson. Monitoring web service requirements. In *Proceedings of International Requirements Engineering Conference (RE03)*, pages 65–74, 2003.

[40] M. Sabetzadeh and S. Easterbrook. View merging in the presence of incompleteness and inconsistency. *Requirements Engineering Journal*, 11(3):174–193, 2006.

[41] S. Sackmann, J. Strüker, and R. Accorsi. Personalization in

privacy-aware highly dynamic systems. *Communications of the ACM (CACM)*, 49(9):32–38, September 2006.

[42] T. Savor and R. Seviora. An approach to automatic detection of software failures in realtime systems. In *IEEE Real-Time Technology and Applications Symposium*, pages 136–147, 1997.

[43] Schloss Dagstuhl Seminar 08031, Wadern, Germany. *Software Engineering for Self-Adaptive Systems*, January 2008. `http://www.dagstuhl.de/08031/`.

[44] G. Seetharaman, A. Lakhotia, and E. P. Blasch. Unmanned Vehicles Come of Age: The DARPA Grand Challenge. *Computer*, 39(12):26–29, 2006.

[45] M. Shaw. Beyond objects. *ACM SIGSOFT Software Engineering Notes (SEN)*, 20(1):27–38, January 1995.

[46] H. A. Simon. *The Sciences of the Artificial*. MIT Press. Cambridge, MA, USA, 1981.

[47] A. Sutcliffe, S. Fickas, and M. M. Sohlberg. PC-RE a method for personal and context requirements engineering with some experience. *Requirements Engineering Journal*, 11(3):1–17, 2006.

[48] M. Svahnberg, J. van Gurp, and J. Bosch. A taxonomy of variability realization techniques. *Software: Practice and Experience*, 35(8):705 – 754, 2005.

[49] J. A. Tanner. Feedback control in living prototypes: A new vista in control engineering. *Medical and Biological Engineering and Computing*, 1(3):333–351, August 1963. `http://www.springerlink.com/content/rh7wx0675k5mx544/`.

[50] D. Weyns. *An architecture-centric approach for software engineering with situated multiagent systems*. PhD thesis, Department of Computer Science, K.U. Leuven, Leuven, Belgium, October 2006.

[51] E. S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *3rd IEEE International Symposium on Requirements Engineering (RES97)*, page 226, Washington, DC, USA, 1997.

[52] J. Zhang and B. H. C. Cheng. Model-based development of dynamically adaptive software. In *Proceedings of International Conference on Software Engineering (ICSE'06)*, Shanghai,China, May 2006.

[53] J. Zhang and B. H. C. Cheng. Using temporal logic to specify adaptive program semantics. *Journal of Systems and Software (JSS), Architecting Dependable Systems*, 79(10):1361–1369, 2006.